Voblint

A Verified Goblint-Style Static Analysis Pipeline in Isabelle/HOL

Manuel Lerchner

October 28, 2026

Abstract

Voblint formalizes an executable interprocedural static analysis pipeline in Isabelle/HOL, following Goblint's D/G architecture. The development connects programs in VIMP, a procedural imperative language, to sound abstract results through a verified control-flow graph compiler, an equation generator, and a verified side-effecting top-down solver. The soundness argument applies to the result computed by the analyzer.

The framework separates program semantics, abstract domains, transfer operations, and solver integration. Its D/G interface distinguishes local analysis states from shared information communicated through side effects. Reusable soundness results support context-insensitive and context-sensitive analyses. Concrete instances include sign, interval, parity, and congruence analyses, together with a reduced product of these four domains.

The main soundness theorem relates every finite source-execution prefix to a corresponding control-flow graph state, establishes coverage by the computed abstract result, and proves the correctness of reported check verdicts. It applies to accepted programs and configurations under a solver-termination assumption; source-program termination is unnecessary. The verified pipeline begins with a VIMP abstract syntax tree. Parsing, code generation, and the external compilation and runtime toolchain remain outside the proved guarantees. This document presents the formalization by session, together with an executable end-to-end certificate and the code-export boundary.

The source code and build instructions are available in the Voblint GitHub repository.

Contents

1 Voblint_VIMP **1**
- 1.1 VIMP_Syntax . . . 2
- 1.2 VIMP_Globals . . . 3
- 1.3 VIMP_Expr . . . 4
- 1.4 VIMP_Special . . . 7
- 1.5 VIMP_Proc . . . 8
- 1.6 VIMP_Program . . . 16
- 1.7 VIMP_Grammar_Generated . . . 18
- 1.8 VIMP_Notation . . . 22

2 Voblint_Domain **25**
- 2.1 Abstract_Domain . . . 26
- 2.2 Reachability_Lift . . . 29
- 2.3 Three_Valued . . . 36
- 2.4 Query_Lift . . . 37
- 2.5 State_Concretization . . . 39
- 2.6 Nonrelational_State . . . 40
- 2.7 Nonrelational_Reachability . . . 43
- 2.8 Forward_Domain . . . 46
- 2.9 Backward_Domain . . . 47
- 2.10 Backward_Domain_Mono . . . 56
- 2.11 Numeric_Queries . . . 64
- 2.12 Backward_Numeric_Queries . . . 66
- 2.13 Interval_Lattice . . . 68
- 2.14 Sign_Lattice . . . 76
- 2.15 Parity_Lattice . . . 80
- 2.16 Congruence_Lattice . . . 83
- 2.17 Int_Lattice . . . 97
- 2.18 Order_Lattice . . . 108

3 Voblint_Solver **111**
- 3.1 Strategy_Program_Fold . . . 112
- 3.2 Strategy_Tree_Program . . . 117
- 3.3 Strategy_Tree_Side_Buffering . . . 122
- 3.4 Strategy_Tree_Pair . . . 128
- 3.5 TD_Solver_Bridge . . . 130
- 3.6 Globals_Rule . . . 131
- 3.7 Solver_Trace . . . 134

4 Voblint_CFG **138**
- 4.1 CFG_Def . . . 139
- 4.2 CFG_Transfer . . . 144
- 4.3 CFG_Prune . . . 146
- 4.4 CFG_Exec . . . 148
- 4.5 Activation_Trace_Def . . . 150
- 4.6 Activation_Trace_Context . . . 155
- 4.7 Activation_Trace_Collect . . . 161
- 4.8 Activation_Trace_Abstract . . . 165

5 Voblint_Compile **170**
5.1 VIMP_Proc_to_CFG . . . 171
5.2 Compile_Wellformed . . . 178
5.3 Compile_Invariants . . . 188
5.4 Residual_Location . . . 194
5.5 Residual_Edges . . . 196
5.6 Simulation_Relation . . . 208
5.7 Simulation_Preservation . . . 216
5.8 Source_Progress . . . 228
5.9 Procedure_Ownership . . . 232
5.10 Live_Nodes . . . 241
5.11 Source_To_Trace . . . 248
5.12 Elaborated_Compilation . . . 254

6 Voblint_Framework **265**
6.1 CFG_Enumeration . . . 266
6.2 DG_State . . . 274
6.3 DG_Constraint_Programs . . . 276
6.4 DG_Manager . . . 281
6.5 DG_Spec . . . 284
6.6 DG_Spec_Sound . . . 291
6.7 Transfer_Algebra . . . 297
6.8 State_Restriction . . . 300
6.9 DG_Ownership_Split_Spec . . . 302
6.10 DG_Indexed_Generator . . . 306
6.11 DG_Reader_Transport . . . 321
6.12 Activation_Backbone . . . 328
6.13 DG_Ctx_Activation . . . 329
6.14 Routed_Call_Programs . . . 333
6.15 Routed_Context . . . 341
6.16 Routed_Context_Unit . . . 352
6.17 DG_Local_State_Spec . . . 353
6.18 DG_Keyed_Split_Spec . . . 357
6.19 Analysis_Query . . . 363
6.20 Local_Spec_Product . . . 365
6.21 MCP_Spec . . . 367
6.22 Channel_Wrappers . . . 382
6.23 Call_String_Context . . . 384
6.24 Solved_Table . . . 386
6.25 Checks . . . 390
6.26 Check_Result . . . 391
6.27 Abstract_Checks . . . 392
6.28 Check_Report . . . 395
6.29 Contextual_Check_Report . . . 397
6.30 Check_Answer . . . 402
6.31 DG_Analysis_Adapter . . . 404
6.32 Routed_Analysis_Sound . . . 409

7 Voblint_Exec **411**
7.1 Default_St_Base . . . 412
7.2 Default_St_Algebra . . . 417
7.3 Default_St_Transfer . . . 421
7.4 Default_St_Reachability . . . 429
7.5 Default_St_Restriction_Refinement . . . 434
7.6 Exec_DG_State . . . 435
7.7 Ownership_Split_Exec . . . 436
7.8 DG_Local_State_Exec . . . 437
7.9 DG_Local_State_Exec_Refinement . . . 439
7.10 Routed_Exec_Refinement . . . 443

8 Voblint_Routing **446**
8.1 Compiled_Routed_Equations . . . 447
8.2 Call_String_Routed_Context . . . 449
8.3 Entry_State_Routed_Context . . . 452
8.4 Context_Space_Finite . . . 455

9 Voblint_Result **457**
9.1 Source_Activation_Sound . . . 458
9.2 DG_Result_Construction . . . 462
9.3 Analysis_Surface . . . 463
9.4 DG_Analysis . . . 465
9.5 DG_Live_Unknowns . . . 487

10 Voblint_Nonrelational **498**
10.1 Abstract_Arithmetic . . . 499
10.2 Special_Ops . . . 503
10.3 Nonrelational_Transfer . . . 505
10.4 Nonrelational_Ops . . . 510
10.5 Exec_Backward . . . 513
10.6 Nonrel_Registration . . . 523

11 Voblint_Analysis_Sign **525**
11.1 Sign_Warrowing . . . 526
11.2 Sign_Arithmetic . . . 527
11.3 Sign_Backward . . . 532
11.4 Sign_Special . . . 536
11.5 Sign_Numeric_Queries . . . 538
11.6 Sign_Transfer . . . 540
11.7 Sign_Exec . . . 542
11.8 Sign_Sound . . . 543
11.9 Sign_Classify . . . 544
11.10 Sign_Analyses . . . 545

12 Voblint_Analysis_Interval **546**
12.1 Interval_Warrowing . . . 547
12.2 Interval_Arithmetic . . . 551
12.3 Interval_Backward . . . 562
12.4 Interval_Transfer . . . 568
12.5 Interval_Domain . . . 570
12.6 Interval_Exec . . . 571
12.7 Interval_Special . . . 572
12.8 Interval_Numeric_Queries . . . 573
12.9 Interval_Point_Digest . . . 577
12.10 Interval_Sound . . . 578
12.11 Interval_Classify . . . 579
12.12 Interval_Analyses . . . 580

13 Voblint_Analysis_Parity **581**
13.1 Parity_Warrowing . . . 582
13.2 Parity_Domain . . . 583
13.3 Parity_Special . . . 587
13.4 Parity_Backward . . . 589
13.5 Parity_Transfer . . . 591
13.6 Parity_Exec . . . 593
13.7 Parity_Numeric_Queries . . . 594
13.8 Parity_Sound . . . 596
13.9 Parity_Classify . . . 597
13.10 Parity_Analyses . . . 598

14 Voblint_Analysis_Congruence **599**
14.1 Congruence_Warrowing . . . 600
14.2 Congruence_Arithmetic . . . 602
14.3 Congruence_Backward . . . 614
14.4 Congruence_Special . . . 624
14.5 Congruence_Transfer . . . 626
14.6 Congruence_Exec . . . 628
14.7 Congruence_Sound . . . 629
14.8 Congruence_Classify . . . 630
14.9 Congruence_Analyses . . . 631

15 Voblint_Analysis_Relational **632**
15.1 Rel_Order_Domain . . . 633
15.2 Rel_Order_Local . . . 639

16 Voblint_Analysis_Int **641**
16.1 Int_Warrowing . . . 642
16.2 Int_Refinement . . . 645
16.3 Int_Refinement_Control . . . 663
16.4 Int_Arithmetic . . . 668
16.5 Int_Backward . . . 678
16.6 Int_Transfer . . . 685
16.7 Int_Exec . . . 688
16.8 Int_Sound . . . 689
16.9 Int_Classify . . . 690
16.10 Int_Fixpoint_Analyses . . . 691
16.11 Int_Once_Analyses . . . 692
16.12 Int_Never_Analyses . . . 693

17 Voblint_CLI **694**
17.1 Analysis_Config . . . 695
17.2 Dispatch_Carrier . . . 696
17.3 MCP_Field . . . 698
17.4 MCP_Carrier . . . 699
17.5 MCP_Analyses . . . 706
17.6 MCP_Split . . . 711
17.7 Analysis_Run . . . 720
17.8 Analysis_Render . . . 726
17.9 Analysis_Run_Sound . . . 731
17.10 Analysis_Run_Ctx_Sound . . . 735
17.11 Analysis_Report . . . 737
17.12 Analysis_Certified . . . 751
17.13 Trace_Run . . . 757
17.14 Arithmetic_Diagnostics . . . 760

18 Voblint_Codegen **764**
18.1 Voblint_Codegen . . . 765

A End-to-end certificate **768**
A.1 Example_End_To_End_Certificate . . . 769

Chapter 1

Voblint_VIMP

1.1 VIMP_Syntax

```
theory VIMP_Syntax
  imports Main "HOL-Library.Countable" "Deriving.Compare_Order_Instances"
begin
```

Source-language expressions

One integer-valued expression language, as in C: comparisons and logical operators are ordinary constructors evaluating to `0`/`1`, and source `true`/`false` lower to `N 1`/`N 0` in the grammar.

```
type_synonym vname = String.literal
type_synonym store = "vname ⇒ int"

datatype exp =
    N (exp_lit: int)
  | V (exp_var: vname)
  | Plus  (exp_lhs: exp) (exp_rhs: exp)
  | Minus (exp_lhs: exp) (exp_rhs: exp)
  | Times (exp_lhs: exp) (exp_rhs: exp)
  | Div (exp_lhs: exp) (exp_rhs: exp)
  | Mod (exp_lhs: exp) (exp_rhs: exp)
  | Less (cmp_lhs: exp) (cmp_rhs: exp)
  | LessEq (cmp_lhs: exp) (cmp_rhs: exp)
  | Greater (cmp_lhs: exp) (cmp_rhs: exp)
  | GreaterEq (cmp_lhs: exp) (cmp_rhs: exp)
  | NotEq (cmp_lhs: exp) (cmp_rhs: exp)
  | Eq (cmp_lhs: exp) (cmp_rhs: exp)
  | Not (exp_arg: exp)
  | And (exp_lhs: exp) (exp_rhs: exp)
  | Or (exp_lhs: exp) (exp_rhs: exp)

instance exp :: countable
  by countable_datatype
```

The executable linear order gives `sorted_list_of_set` a deterministic representation of CFG edge sets; `compare_order` on `String.literal` makes it a usable leaf for the derivation.

```
derive (linorder) compare_order String.literal
derive linorder exp

end
```

1.2 VIMP_Globals

```
theory VIMP_Globals
  imports VIMP_Syntax
begin
```

Locals and globals over the scalar store

The store is one `vname ⇒ int` function; the locals/globals split is a classifier `𝒢 :: vname ⇒ bool` on top of it. A call needs both halves of that split: `enter_state` keeps the globals and resets the locals on entry, `combine_env caller callee` restores the caller's locals and keeps the callee's globals on exit.

`combine_env` is the single pointwise selector, generic in both key and codomain. `enter_frame` specializes it with a constant reset map. The concrete semantics uses these operations at `store = vname ⇒ int`, while abstract states reuse them at their own codomain.

```
type_synonym pname = String.literal

definition combine_env ::
    "('k ⇒ bool) ⇒ ('k ⇒ 'a) ⇒ ('k ⇒ 'a) ⇒ ('k ⇒ 'a)" where
  "combine_env 𝒢 s t = (λn. if 𝒢 n then t n else s n)"

definition enter_frame :: "('k ⇒ bool) ⇒ 'a ⇒ ('k ⇒ 'a) ⇒ ('k ⇒ 'a)" where
  "enter_frame 𝒢 reset_val s = combine_env 𝒢 (λ_. reset_val) s"

definition enter_state :: "(vname ⇒ bool) ⇒ store ⇒ store" where
  "enter_state 𝒢 s = enter_frame 𝒢 0 s"

lemma combine_query [simp]:
  "combine_env 𝒢 s t n = (if 𝒢 n then t n else s n)"
  unfolding combine_env_def by simp

lemma enter_frame_apply [simp]:
  "enter_frame 𝒢 reset_val s n = (if 𝒢 n then s n else reset_val)"
  unfolding enter_frame_def by simp

lemma enter_state_apply [simp]:
  "enter_state 𝒢 s n = (if 𝒢 n then s n else 0)"
  unfolding enter_state_def by simp

lemma combine_collapse [simp]:
  "combine_env 𝒢 s s = s"
  by (rule ext) simp

lemma combine_nest_left [simp]:
  "combine_env 𝒢 (combine_env 𝒢 s t) u = combine_env 𝒢 s u"
  by (rule ext) simp

lemma combine_nest_right [simp]:
  "combine_env 𝒢 s (combine_env 𝒢 t u) = combine_env 𝒢 s u"
  by (rule ext) simp
```

C-faithful initial store set

VIMP approximates C startup by initializing every global with no explicit initializer to 0 and leaving every other name of the initial store unconstrained. `cinit_stores` is the corresponding set of concrete stores: those where every global is 0, so `main`'s locals may hold any integer. It constrains only the initial store. A callee's locals do not come from it: `enter_state` resets them to 0 on every entry, which defines what C leaves indeterminate, as `c_div` defines division by zero. Neither convention is a model of C's indeterminate-value or undefined-behaviour rules, and VIMP has no global initializer syntax for a declaration such as `int x = 42;` to diverge from.

```
definition cinit_stores :: "(vname ⇒ bool) ⇒ store set" where
  "cinit_stores 𝒢 = {s. ∀x. 𝒢 x ⟶ s x = 0}"

end
```

1.3 VIMP_Expr

```
theory VIMP_Expr
  imports VIMP_Syntax
begin
```

Expression evaluation

`aval` evaluates an expression in a store. Every VIMP expression is integer-valued, so there is no separate Boolean type. Conditions use C-style truthiness (non-zero), while comparisons and logical operators yield `1` or `0`. Evaluation is total and has no effects, which is why a guard can be re-evaluated freely by the transfer functions.

Division and remainder truncate toward zero for non-zero divisors. VIMP keeps expressions total by defining division by zero as zero and remainder by zero as the dividend. These zero-divisor cases are VIMP conventions; they do not model C's undefined behavior.

```
definition c_div :: "int ⇒ int ⇒ int" where
  "c_div a b =
    (if b = 0 then 0 else sgn a * sgn b * (abs a div abs b))"

definition c_mod :: "int ⇒ int ⇒ int" where
  "c_mod a b = (if b = 0 then a else a - c_div a b * b)"

lemma c_div_zero [simp]: "c_div a 0 = 0" "c_div 0 b = 0"
  by (simp_all add: c_div_def)

lemma c_mod_zero [simp]: "c_mod a 0 = a" "c_mod 0 b = 0"
  by (simp_all add: c_mod_def)

lemma c_div_one [simp]: "c_div a 1 = a" "c_div a (-1) = -a"
  by (simp_all add: c_div_def sgn_mult_abs)

lemma c_mod_one [simp]: "c_mod a 1 = 0" "c_mod a (-1) = 0"
  by (simp_all add: c_mod_def)

lemma c_div_mod_reconstruct:
  "c_div a b * b + c_mod a b = a"
  by (simp add: c_mod_def)

lemma c_mod_abs:
  "c_mod a b = sgn a * (abs a mod abs b)"
proof (cases "b = 0")
  case True
  then show ?thesis by (simp add: sgn_mult_abs)
next
  case False
  have product: "c_div a b * b = sgn a * (abs a div abs b * abs b)"
    using False unfolding c_div_def
    by (simp add: abs_sgn mult_ac)
  have "sgn a * (abs a div abs b * abs b) + sgn a * (abs a mod abs b) = a"
    by (simp only: distrib_left[symmetric] div_mult_mod_eq sgn_mult_abs)
  with False product show ?thesis unfolding c_mod_def by (simp only: False if_False; linarith)
qed

lemma c_div_exact:
  assumes "b dvd a"
  shows "c_div a b = a div b"
proof (cases "b = 0")
  case True
  with assms show ?thesis by simp
next
  case False
  have rem: "c_mod a b = 0"
    using assms by (simp add: c_mod_abs)
  have product: "c_div a b * b = a"
    using c_div_mod_reconstruct[of a b] rem by simp
  have "c_div a b = (c_div a b * b) div b"
    using False by simp
  also have "... = a div b" using product by simp
  finally show ?thesis .
qed

lemma c_div_nonneg:
  "0 ≤ a ⟹ 0 ≤ b ⟹ 0 ≤ c_div a b"
```

```
  "a ≤ 0 ⟹ b ≤ 0 ⟹ 0 ≤ c_div a b"
  by (auto simp: c_div_def sgn_if pos_imp_zdiv_nonneg_iff neg_imp_zdiv_nonneg_iff)

lemma c_div_nonpos:
  "a ≤ 0 ⟹ 0 ≤ b ⟹ c_div a b ≤ 0"
  "0 ≤ a ⟹ b ≤ 0 ⟹ c_div a b ≤ 0"
  by (auto simp: c_div_def sgn_if pos_imp_zdiv_nonneg_iff neg_imp_zdiv_nonneg_iff)

lemma c_mod_nonneg:
  "0 ≤ a ⟹ 0 ≤ c_mod a b"
  by (cases "b = 0") (auto simp: c_mod_abs sgn_if intro: pos_mod_sign)

lemma c_mod_nonpos:
  "a ≤ 0 ⟹ c_mod a b ≤ 0"
  by (cases "b = 0") (auto simp: c_mod_abs sgn_if intro: pos_mod_sign)

lemma c_div_neg_divisor:
  "c_div a (-b) = - c_div a b"
  by (simp add: c_div_def)

lemma c_div_neg_dividend:
  "c_div (-a) b = - c_div a b"
  by (simp add: c_div_def)

definition truthy :: "int ⇒ bool" where
  [simp]: "truthy n ⟷ n ≠ 0"

fun aval :: "exp ⇒ store ⇒ int" ("⟦_⟧e") where
    "⟦N n⟧e     s  = n"
  | "⟦V x⟧e     s  = s x"
  | "⟦Plus  a b⟧e s  = ⟦a⟧e s + ⟦b⟧e s"
  | "⟦Minus a b⟧e s  = ⟦a⟧e s - ⟦b⟧e s"
  | "⟦Times a b⟧e s  = ⟦a⟧e s * ⟦b⟧e s"
  | "⟦Div a b⟧e s = c_div (⟦a⟧e s) (⟦b⟧e s)"
  | "⟦Mod a b⟧e s = c_mod (⟦a⟧e s) (⟦b⟧e s)"
  | "⟦Less a b⟧e  s  = (if ⟦a⟧e s < ⟦b⟧e s then 1 else 0)"
  | "⟦LessEq a b⟧e s = (if ⟦a⟧e s ≤ ⟦b⟧e s then 1 else 0)"
  | "⟦Greater a b⟧e s = (if ⟦a⟧e s > ⟦b⟧e s then 1 else 0)"
  | "⟦GreaterEq a b⟧e s = (if ⟦a⟧e s ≥ ⟦b⟧e s then 1 else 0)"
  | "⟦NotEq a b⟧e s = (if ⟦a⟧e s ≠ ⟦b⟧e s then 1 else 0)"
  | "⟦Eq   a b⟧e  s  = (if ⟦a⟧e s = ⟦b⟧e s then 1 else 0)"
  | "⟦Not b⟧e     s  = (if truthy (⟦b⟧e s) then 0 else 1)"
  | "⟦And b1 b2⟧e s  = (if truthy (⟦b1⟧e s) ∧ truthy (⟦b2⟧e s) then 1 else 0)"
  | "⟦Or  b1 b2⟧e s  = (if truthy (⟦b1⟧e s) ∨ truthy (⟦b2⟧e s) then 1 else 0)"
```

`truthy` of a compiled comparison or Boolean expression restated in plain Boolean form, so a caller never has to re-derive it from `aval.simps` and `truthy`'s own $\neq$ `0` encoding through an explicit `if`-split.

```
lemma truthy_aval_Less [simp]:
  "truthy (⟦Less a b⟧e s) ⟷ ⟦a⟧e s < ⟦b⟧e s"
  by simp

lemma truthy_aval_LessEq [simp]:
  "truthy (⟦LessEq a b⟧e s) ⟷ ⟦a⟧e s ≤ ⟦b⟧e s"
  by simp

lemma truthy_aval_Greater [simp]:
  "truthy (⟦Greater a b⟧e s) ⟷ ⟦a⟧e s > ⟦b⟧e s"
  by simp

lemma truthy_aval_GreaterEq [simp]:
  "truthy (⟦GreaterEq a b⟧e s) ⟷ ⟦a⟧e s ≥ ⟦b⟧e s"
  by simp

lemma truthy_aval_NotEq [simp]:
  "truthy (⟦NotEq a b⟧e s) ⟷ ⟦a⟧e s ≠ ⟦b⟧e s"
  by simp

lemma truthy_aval_Eq [simp]:
  "truthy (⟦Eq a b⟧e s) ⟷ ⟦a⟧e s = ⟦b⟧e s"
  by simp

lemma truthy_aval_Not [simp]:
  "truthy (⟦Not b⟧e s) ⟷ ¬ truthy (⟦b⟧e s)"
  by simp

lemma truthy_aval_And [simp]:
```

```
  "truthy (⟦And b1 b2⟧e s) ⟷ truthy (⟦b1⟧e s) ∧ truthy (⟦b2⟧e s)"
  by simp

lemma truthy_aval_Or [simp]:
  "truthy (⟦Or b1 b2⟧e s) ⟷ truthy (⟦b1⟧e s) ∨ truthy (⟦b2⟧e s)"
  by simp
```

Finite expression footprints

The variables an expression mentions. A transfer that evaluates the expression needs no other variable, which is what lets a keyed analysis read only the globals an edge mentions.

```
fun exp_vnames :: "exp ⇒ vname set" where
  "exp_vnames (N _) = {}"
| "exp_vnames (V x) = {x}"
| "exp_vnames (Plus a b) = exp_vnames a ∪ exp_vnames b"
| "exp_vnames (Minus a b) = exp_vnames a ∪ exp_vnames b"
| "exp_vnames (Times a b) = exp_vnames a ∪ exp_vnames b"
| "exp_vnames (Div a b) = exp_vnames a ∪ exp_vnames b"
| "exp_vnames (Mod a b) = exp_vnames a ∪ exp_vnames b"
| "exp_vnames (Less a b) = exp_vnames a ∪ exp_vnames b"
| "exp_vnames (LessEq a b) = exp_vnames a ∪ exp_vnames b"
| "exp_vnames (Greater a b) = exp_vnames a ∪ exp_vnames b"
| "exp_vnames (GreaterEq a b) = exp_vnames a ∪ exp_vnames b"
| "exp_vnames (NotEq a b) = exp_vnames a ∪ exp_vnames b"
| "exp_vnames (Eq a b) = exp_vnames a ∪ exp_vnames b"
| "exp_vnames (Not b) = exp_vnames b"
| "exp_vnames (And b1 b2) = exp_vnames b1 ∪ exp_vnames b2"
| "exp_vnames (Or b1 b2) = exp_vnames b1 ∪ exp_vnames b2"

lemma finite_exp_vnames [simp]: "finite (exp_vnames a)"
  by (induction a) auto

end
```

1.4 VIMP_Special

```
theory VIMP_Special
  imports VIMP_Expr VIMP_Globals
begin
```

Special calls

VIMP's analogue of Goblint's library-function dispatch, as a closed enumeration rather than an open table. Classification is two-level, as in Goblint: `special_table` resolves a callee name to a shape descriptor and `classify_special` applies the descriptor to the actuals, so no consumer re-checks arity. A special call never enters an activation.

```
datatype special_desc = SD_Nondet_Int | SD_Min | SD_Max

datatype special_call =
    Nondet_Int
  | Min exp exp
  | Max exp exp

instance special_call :: countable
  by countable_datatype

fun classify_special :: "special_desc ⇒ exp list ⇒ special_call option" where
  "classify_special SD_Nondet_Int [] = Some Nondet_Int"
| "classify_special SD_Min [a, b] = Some (Min a b)"
| "classify_special SD_Max [a, b] = Some (Max a b)"
| "classify_special _ _ = None"
```

Shared by `pstep`'s `Special` rule and the CFG's `special_step`, so the two concrete semantics cannot drift. `Nondet_Int` admits every integer.

```
fun special_result :: "special_call ⇒ store ⇒ int ⇒ bool" where
  "special_result Nondet_Int s v = True"
| "special_result (Min a b) s v = (v = min ([[a]]e s) ([[b]]e s))"
| "special_result (Max a b) s v = (v = max ([[a]]e s) ([[b]]e s))"

lemma special_result_ex [simp]: "∃v. special_result sc s v"
  by (cases sc) auto
```

Every `special_pname_*` is an ordinary identifier, so a source program could declare a colliding procedure; `wf_source_program` rejects that instead of letting special-call semantics shadow the declaration.

```
definition special_pname_nondet_int :: pname where
  "special_pname_nondet_int = STR ''__voblint_nondet_int''"

definition special_pname_min :: pname where
  "special_pname_min = STR ''min''"

definition special_pname_max :: pname where
  "special_pname_max = STR ''max''"

definition special_table :: "pname ⇒ special_desc option" where
  "special_table p =
     (if      p = special_pname_nondet_int then Some SD_Nondet_Int
      else if p = special_pname_min       then Some SD_Min
      else if p = special_pname_max       then Some SD_Max
      else None)"

end
```

1.5 VIMP_Proc

```
theory VIMP_Proc
  imports VIMP_Special "HOL-IMP.Star"
begin
```

Procedure commands and activation frames

A call evaluates its actuals in the caller store, binds them in a fresh activation, and pushes a frame holding the caller store and the optional destination. `Restore` and `Unwind` never occur in source programs: `Restore` marks the activation boundary inside sequential syntax, and `Unwind` is the state after a `Return` has published its value through `ret_var`, discarding pending commands up to the nearest `Restore`.

A check carries a label naming where it was written. The semantics never reads it; compilation copies it onto the check's edge, and the analysis result lists each check under it, so a report can be matched to its source check by label instead of by position in a list. The CLI parser writes the line and column of the check's `__voblint_check` keyword; a program written in Isabelle notation carries the placeholder `(0, 0)`.

```
type_synonym check_label = "nat × nat"
```

Every statement carries an annotation `'a`. Source programs written by hand or produced by the parser carry `()`; elaboration replaces it with a statement identity. The semantics never reads it: a step copies the annotation of the command it reduces onto what it leaves behind, so a residual still names the statement it came from. The variable and expression payloads are parameters too, so a later resolved or typed front end instantiates the same command syntax.

```
datatype ('a, 'v, 'e) gcom =
    is_SKIP: SKIP 'a
  | Assign 'a (assign_var: 'v) (assign_rhs: 'e)
  | Check  'a check_label (check_cond: 'e)
  | Seq    (seq_first: "('a, 'v, 'e) gcom") (seq_second: "('a, 'v, 'e) gcom")
  | If     'a (if_cond: 'e) (if_then: "('a, 'v, 'e) gcom") (if_else: "('a, 'v, 'e) gcom")
  | While  'a (while_cond: 'e) (while_body: "('a, 'v, 'e) gcom")
  | Call   'a (call_dest: "'v option") (call_proc: pname) (call_args: "'e list")
  | Return 'a (return_val: "'e option")
  | is_Restore: Restore 'a
  | Unwind 'a

type_synonym 'a acom = "('a, vname, exp) gcom"
type_synonym com = "unit acom"
```

The discriminators read as the equations they replace, so automation that used to substitute `c = SKIP` still finds the constructor.

```
lemma is_SKIP_iff [simp]: "is_SKIP c ⟷ (∃i. c = SKIP i)"
  by (cases c) auto

lemma is_Restore_iff [simp]: "is_Restore c ⟷ (∃i. c = Restore i)"
  by (cases c) auto
```

Activation – fresh locals and formal binding – happens at the call site, so a declaration records only what a call site looks up by name.

```
record 'a gproc_decl =
  formals :: "vname list"
  body    :: "'a acom"

type_synonym proc_decl = "unit gproc_decl"
```

The reserved local `ret_var` carries a published result to the restore boundary. It is absent from source programs, so a value-less completion leaves it at `enter_state`'s zero.

```
definition ret_var :: vname where
  "ret_var = STR ''#ret''"

type_synonym 'a ptable = "pname ⇒ 'a gproc_decl option"
type_synonym proc_table = "unit ptable"

datatype frame = Frame (frame_store: store) (frame_dest: "vname option")
```

Formal binding is the same fold over the concrete store and over every abstract state, so it is one polymorphic abbreviation.

```
abbreviation bind_formals :: "vname list ⇒ 'v list ⇒ (vname ⇒ 'v) ⇒ vname ⇒ 'v" where
  "bind_formals xs vs s ≡ fold (λ(x, v) st. st(x := v)) (zip xs vs) s"
```

```
lemma bind_formals_other:
  "x ∉ set xs ⟹ bind_formals xs vs s x = s x"
proof (induction xs arbitrary: vs s)
  case Nil then show ?case by simp
next
  case (Cons y ys)
  then show ?case by (cases vs) simp_all
qed
```

Whole procedure entry, for the same reason: reset every local to a chosen value, keep the globals, then bind the formals to the actuals evaluated in the caller's own state. Which state and which evaluator is all that separates the concrete entry from an abstract one – the concrete semantics runs it at `aval` and the reset value `0`, an abstract domain at its own evaluator and its own `top` – so it is one definition, not a concrete one with an `_abs` copy beside it.

```
definition enter_binding ::
  "(vname ⇒ bool) ⇒ 'a ⇒ (exp ⇒ (vname ⇒ 'a) ⇒ 'a)
   ⇒ vname list ⇒ exp list ⇒ (vname ⇒ 'a) ⇒ (vname ⇒ 'a)"
where
  "enter_binding 𝒢 reset_val ev xs es s =
     bind_formals xs (map (λe. ev e s) es) (enter_frame 𝒢 reset_val s)"
```

The concrete instance, spelled out: this is the equation that lets a soundness statement about the shared constant be read as one about the entry store the source semantics actually builds.

```
lemma enter_binding_concrete:
  "enter_binding 𝒢 0 aval xs es s
     = bind_formals xs (map (λe. ⟦e⟧e s) es) (enter_state 𝒢 s)"
  by (simp add: enter_binding_def enter_state_def)
```

The return-slot write is a pure per-variable update, generic in the codomain: the concrete VIMP semantics uses it at `store = vname ⇒ int`, and every abstract domain's own `vname ⇒ 'a` state reuses the same definition rather than restating an `_abs` copy of it.

```
fun combine_assign :: "vname option ⇒ 'a ⇒ (vname ⇒ 'a) ⇒ (vname ⇒ 'a)" where
  "combine_assign None _ s = s"
| "combine_assign (Some x) v s = s(x := v)"
```

Frame-stack small-step semantics

The reference semantics every soundness claim is ultimately about. A configuration is a command, a store, and a stack of caller frames; a call pushes one frame and continues in the callee body followed by `Restore`, a return writes the return slot and hands control to `Unwind`, which discards the rest of the callee body until that `Restore` pops the frame. `Restore` and `Unwind` exist only here, in configurations the semantics builds; no source program contains them.

```
inductive
  pstep :: "(vname ⇒ bool) ⇒ 'a ptable ⇒ 'a acom × store × frame list
                           ⇒ 'a acom × store × frame list ⇒ bool"
    ("(_,_ ⊢/ _ →p/ _)" [51, 51, 51, 51] 50)
  for 𝒢 :: "vname ⇒ bool" and Π :: "'a ptable"
where
  Assign:  "𝒢, Π ⊢ (Assign i x a, s, frs) →p (SKIP i, s(x := ⟦a⟧e s), frs)"
| Check:   "𝒢, Π ⊢ (Check i l c, s, frs) →p (SKIP i, s, frs)"
| Seq1:    "𝒢, Π ⊢ (Seq (SKIP i) c2, s, frs) →p (c2, s, frs)"
| Seq2:    "𝒢, Π ⊢ (c1, s, frs) →p (c1', s', frs')
             ⟹ 𝒢, Π ⊢ (Seq c1 c2, s, frs) →p (Seq c1' c2, s', frs')"
| IfTrue:  "truthy (⟦b⟧e s) ⟹ 𝒢, Π ⊢ (If i b c1 c2, s, frs) →p (c1, s, frs)"
| IfFalse: "¬ truthy (⟦b⟧e s) ⟹ 𝒢, Π ⊢ (If i b c1 c2, s, frs) →p (c2, s, frs)"
| While:   "𝒢, Π ⊢ (While i b c, s, frs)
                      →p (If i b (Seq c (While i b c)) (SKIP i), s, frs)"
| Call:    "Π p = Some decl
             ⟹ length actuals = length (formals decl)
             ⟹ distinct (formals decl)
             ⟹ vals = map (λe. ⟦e⟧e s) actuals
             ⟹ callee = bind_formals (formals decl) vals (enter_state 𝒢 s)
             ⟹ 𝒢, Π ⊢ (Call i dst p actuals, s, frs)
                 →p (Seq (body decl) (Restore i),
                     callee,
                     Frame s dst # frs)"
| Special: "special_table p = Some desc
             ⟹ classify_special desc actuals = Some sc
             ⟹ special_result sc s v
             ⟹ 𝒢, Π ⊢ (Call i (Some x) p actuals, s, frs) →p (SKIP i, s(x := v), frs)"
```

```
| RestoreStep:
    "𝒢, Π ⊢ (Restore i, s, Frame fr dst # frs)
       →p (SKIP i, combine_assign dst (s ret_var) (combine_env 𝒢 fr s), frs)"
| ReturnSome:
    "𝒢, Π ⊢ (Return i (Some e), s, frs)
       →p (Unwind i, s(ret_var := ⟦e⟧e s), frs)"
| ReturnNone:
    "𝒢, Π ⊢ (Return i None, s, frs) →p (Unwind i, s, frs)"
| UnwindDead:
    "¬ is_Restore c2
     ⟹ 𝒢, Π ⊢ (Seq (Unwind i) c2, s, frs) →p (Unwind i, s, frs)"
| UnwindAct:
    "𝒢, Π ⊢ (Seq (Unwind j) (Restore i), s, Frame fr dst # frs)
       →p (SKIP i, combine_assign dst (s ret_var) (combine_env 𝒢 fr s), frs)"

abbreviation
  psteps :: "(vname ⇒ bool) ⇒ 'a ptable ⇒ 'a acom × store × frame list
                        ⇒ 'a acom × store × frame list ⇒ bool"
    ("(_,_ ⊢/ _ →p*/ _)" [51, 51, 51, 51] 50)
where "psteps 𝒢 Π x y ≡ star (pstep 𝒢 Π) x y"

declare pstep.intros [simp, intro]

inductive_cases SkipSE[elim!]:
  "𝒢, Π ⊢ (SKIP i, s, frs) →p cfg"
inductive_cases AssignSE[elim!]:
  "𝒢, Π ⊢ (Assign i x a, s, frs) →p cfg"
inductive_cases CheckSE[elim!]:
  "𝒢, Π ⊢ (Check i l c, s, frs) →p cfg"
inductive_cases SeqSE[elim]:
  "𝒢, Π ⊢ (Seq c1 c2, s, frs) →p cfg"
inductive_cases IfSE[elim!]:
  "𝒢, Π ⊢ (If i b c1 c2, s, frs) →p cfg"
inductive_cases WhileSE[elim]:
  "𝒢, Π ⊢ (While i b c, s, frs) →p cfg"
inductive_cases CallSE[elim]:
  "𝒢, Π ⊢ (Call i dst p actuals, s, frs) →p cfg"
inductive_cases RestoreSE[elim!]:
  "𝒢, Π ⊢ (Restore i, s, frs) →p cfg"
inductive_cases ReturnSE[elim!]:
  "𝒢, Π ⊢ (Return i e, s, frs) →p cfg"
inductive_cases UnwindSE[elim!]:
  "𝒢, Π ⊢ (Unwind i, s, frs) →p cfg"

lemmas star_pstep_induct =
  star.induct[of "pstep 𝒢 Π", split_format(complete), case_names refl step]
```

Completing runs

A run completes when it reaches `SKIP` with the frame stack it started with; `SKIP` under a non-empty stack is stuck, not finished.

```
abbreviation pcompletes :: "(vname ⇒ bool) ⇒ proc_table ⇒ com ⇒ store ⇒ store ⇒ bool" where
  "pcompletes 𝒢 Π c s t ≡ 𝒢, Π ⊢ (c, s, []) →p* (SKIP (), t, [])"

lemma pcompletes_skip: "pcompletes 𝒢 Π (SKIP ()) s s"
  by (rule star.refl)

lemma pcompletes_assign: "pcompletes 𝒢 Π (Assign () x a) s (s(x := ⟦a⟧e s))"
  by simp

lemma pcompletes_special_nondet_int:
  "pcompletes 𝒢 Π (Call () (Some x) special_pname_nondet_int []) s (s(x := v))"
  by (simp add: special_table_def)

lemma psteps_Seq2:
  "𝒢, Π ⊢ (c1, s, frs) →p* (c1', s', frs')
   ⟹ 𝒢, Π ⊢ (Seq c1 c2, s, frs) →p* (Seq c1' c2, s', frs')"
  by (induction rule: star_pstep_induct) (auto intro: star.step)

lemma pcompletes_Seq:
  assumes "pcompletes 𝒢 Π c1 s s2" and "pcompletes 𝒢 Π c2 s2 t"
  shows "pcompletes 𝒢 Π (Seq c1 c2) s t"
  using psteps_Seq2[OF assms(1)] assms(2) by (meson Seq1 star.step star_trans)

lemma pcompletes_IfTrue:
```

```
  "truthy (⟦b⟧e s) ⟹ pcompletes 𝒢 Π c1 s t ⟹ pcompletes 𝒢 Π (If () b c1 c2) s t"
  by (meson IfTrue star.step)

lemma pcompletes_IfFalse:
  "¬ truthy (⟦b⟧e s) ⟹ pcompletes 𝒢 Π c2 s t ⟹ pcompletes 𝒢 Π (If () b c1 c2) s t"
  by (meson IfFalse star.step)
```

Frame-stack extension

A step never inspects the frames below the one it works on, so any run stays valid with more callers underneath. This is what lets a proof about a procedure in isolation be reused at an arbitrary call depth.

```
lemma pstep_frame_extend:
  "𝒢, Π ⊢ (c, s, frs) →p (c', s', frs') ⟹
   𝒢, Π ⊢ (c, s, frs @ extra) →p (c', s', frs' @ extra)"
  by (induction "(c, s, frs)" "(c', s', frs')"
        arbitrary: c s frs c' s' frs' rule: pstep.induct)
     auto

lemma psteps_frame_extend:
  "𝒢, Π ⊢ (c, s, frs) →p* (c', s', frs') ⟹
   𝒢, Π ⊢ (c, s, frs @ extra) →p* (c', s', frs' @ extra)"
  by (induction rule: star_pstep_induct) (auto intro: pstep_frame_extend star.step)

lemma psteps_frame_mono:
  "𝒢, Π ⊢ (c, s, []) →p* (SKIP i, t, []) ⟹
   𝒢, Π ⊢ (c, s, extra) →p* (SKIP i, t, extra)"
  using psteps_frame_extend[where frs = "[]" and frs' = "[]" and extra = extra]
  by simp
```

Call completion

Running a whole call in one step of reasoning: a body that finishes under its own frame lets the caller resume with the frame popped and the return slot written. Callers use these instead of replaying the `Restore` bookkeeping at every call site.

```
lemma combine_env_ret_var_irrelevant [simp]:
  "¬ 𝒢 ret_var ⟹ combine_env 𝒢 fr (t(ret_var := v)) = combine_env 𝒢 fr t"
  by (rule ext) simp

lemma psteps_Seq_Restore_body:
  assumes "𝒢, Π ⊢ (c, s0, [Frame fr dst]) →p* (SKIP i, t', [Frame fr dst])"
  shows "𝒢, Π ⊢ (Seq c (Restore j), s0, [Frame fr dst])
           →p* (SKIP j, combine_assign dst (t' ret_var) (combine_env 𝒢 fr t'), [])"
  using psteps_Seq2[OF assms] by (meson Seq1 RestoreStep star.refl star.step star_trans)

lemma pstep_Call:
  assumes "Π p = Some decl"
      and "length actuals = length (formals decl)"
      and "distinct (formals decl)"
  shows "𝒢, Π ⊢ (Call i dst p actuals, s, frs)
           →p (Seq (body decl) (Restore i),
            bind_formals (formals decl) (map (λe. ⟦e⟧e s) actuals) (enter_state 𝒢 s),
            Frame s dst # frs)"
  using assms by (rule Call) (rule refl)+

lemma pstep_Call_parameterless [intro]:
  assumes "Π p = Some (⦇formals = [], body = c⦈)"
  shows
    "𝒢, Π ⊢ (Call i dst p [], s, frs) →p (Seq c (Restore i), enter_state 𝒢 s, Frame s dst # frs)"
proof -
  have "𝒢, Π ⊢ (Call i dst p [], s, frs)
          →p (Seq (body (⦇formals = [], body = c⦈)) (Restore i),
           bind_formals (formals (⦇formals = [], body = c⦈))
             (map (λe. ⟦e⟧e s) []) (enter_state 𝒢 s),
           Frame s dst # frs)"
    using assms by (rule pstep_Call) simp_all
  then show ?thesis by simp
qed
```

A value reaches the destination only through an explicit `Return`; otherwise the destination receives `ret_var`'s `enter_state` value `0`.

```
lemma pcompletes_Call:
  assumes p: "Π p = Some decl"
      and arity: "length actuals = length (formals decl)"
```

```
    and distinct_formals: "distinct (formals decl)"
    and body: "pcompletes 𝒢 Π (body decl)
                (bind_formals (formals decl) (map (λe. ⟦e⟧e s) actuals) (enter_state 𝒢 s)) t'"
  shows "pcompletes 𝒢 Π (Call () dst p actuals) s
         (combine_assign dst (t' ret_var) (combine_env 𝒢 s t'))"
  by (rule star.step, rule pstep_Call[where Π = Π and p = p, OF p arity distinct_formals])
     (rule psteps_Seq_Restore_body[OF psteps_frame_mono[OF body]])
```

Source-program well-formedness

Source syntax excludes the runtime markers `Restore` and `Unwind`. The reserved `ret_var` belongs to the call mechanism, so source expressions, assignments, destinations, and formal parameters cannot mention it.

Finite syntactic variable sets

```
fun com_vnames :: "'a acom ⇒ vname set" where
  "com_vnames (SKIP _) = {}"
| "com_vnames (Assign _ x a) = insert x (exp_vnames a)"
| "com_vnames (Check _ l c) = exp_vnames c"
| "com_vnames (Seq c1 c2) = com_vnames c1 ∪ com_vnames c2"
| "com_vnames (If _ b c1 c2) =
    exp_vnames b ∪ com_vnames c1 ∪ com_vnames c2"
| "com_vnames (While _ b c) = exp_vnames b ∪ com_vnames c"
| "com_vnames (Call _ dst _ actuals) =
    (case dst of None ⇒ {} | Some x ⇒ {x}) ∪
    ⋃ (set (map exp_vnames actuals))"
| "com_vnames (Return _ e) = (case e of None ⇒ {} | Some a ⇒ exp_vnames a)"
| "com_vnames (Restore _) = {}"
| "com_vnames (Unwind _) = {}"

lemma finite_com_vnames [simp]: "finite (com_vnames c)"
  by (induction c) (auto split: option.splits)

fun source_com :: "'a acom ⇒ bool" where
  "source_com (SKIP _) = True"
| "source_com (Assign _ x a) = True"
| "source_com (Check _ l c) = True"
| "source_com (Seq c1 c2) = (source_com c1 ∧ source_com c2)"
| "source_com (If _ b c1 c2) = (source_com c1 ∧ source_com c2)"
| "source_com (While _ b c) = source_com c"
| "source_com (Call _ dst p actuals) = True"
| "source_com (Return _ e) = True"
| "source_com (Restore _) = False"
| "source_com (Unwind _) = False"

definition source_pi :: "'a ptable ⇒ bool" where
  "source_pi Π = (∀p decl. Π p = Some decl ⟶ source_com (body decl))"

definition source_exp :: "exp ⇒ bool" where
  "source_exp a ⟷ ret_var ∉ exp_vnames a"

definition valid_formal :: "(vname ⇒ bool) ⇒ vname ⇒ bool" where
  "valid_formal 𝒢 x ⟷ ¬ 𝒢 x ∧ x ≠ ret_var"
```

Procedure result contract

Declarations carry no return-kind annotation. A body is a value provider when every syntactic way to finish the body reaches `Return` with a value. The analysis is conservative: a loop may fall through because its guard may be false, and a call resumes normally after its callee completes.

```
fun may_fallthrough :: "'a acom ⇒ bool" where
  "may_fallthrough (SKIP _) = True"
| "may_fallthrough (Assign _ _ _) = True"
| "may_fallthrough (Check _ _ _) = True"
| "may_fallthrough (Seq c1 c2) = (may_fallthrough c1 ∧ may_fallthrough c2)"
| "may_fallthrough (If _ _ c1 c2) = (may_fallthrough c1 ∨ may_fallthrough c2)"
| "may_fallthrough (While _ _ _) = True"
| "may_fallthrough (Call _ _ _ _) = True"
| "may_fallthrough (Return _ _) = False"
| "may_fallthrough (Restore _) = False"
| "may_fallthrough (Unwind _) = False"

fun may_return_none :: "'a acom ⇒ bool" where
  "may_return_none (Seq c1 c2) =
```

```
     (may_return_none c1 ∨ (may_fallthrough c1 ∧ may_return_none c2))"
| "may_return_none (If _ _ c1 c2) = (may_return_none c1 ∨ may_return_none c2)"
| "may_return_none (While _ _ c) = may_return_none c"
| "may_return_none (Return _ e) = (e = None)"
| "may_return_none _ = False"

fun may_return_value :: "'a acom ⇒ bool" where
  "may_return_value (Seq c1 c2) =
     (may_return_value c1 ∨ (may_fallthrough c1 ∧ may_return_value c2))"
| "may_return_value (If _ _ c1 c2) = (may_return_value c1 ∨ may_return_value c2)"
| "may_return_value (While _ _ c) = may_return_value c"
| "may_return_value (Return _ e) = (e ≠ None)"
| "may_return_value _ = False"

definition value_providing :: "'a acom ⇒ bool" where
  "value_providing c ⟷
     source_com c ∧ ¬ may_fallthrough c ∧
     ¬ may_return_none c ∧ may_return_value c"
```

Commands, declarations, and programs

Every call names a declared procedure and supplies one actual per formal. A destination requires a value-providing callee; a destination-less call may discard a published value. Procedure formals are distinct local names, and the distinguished root command contains no return.

```
fun wf_source_com :: "'a ptable ⇒ 'a acom ⇒ bool" where
  "wf_source_com Π (SKIP _) = True"
| "wf_source_com Π (Assign _ x a) = (x ≠ ret_var ∧ source_exp a)"
| "wf_source_com Π (Check _ l c) = source_exp c"
| "wf_source_com Π (Seq c1 c2) = (wf_source_com Π c1 ∧ wf_source_com Π c2)"
| "wf_source_com Π (If _ b c1 c2) =
     (source_exp b ∧ wf_source_com Π c1 ∧ wf_source_com Π c2)"
| "wf_source_com Π (While _ b c) = (source_exp b ∧ wf_source_com Π c)"
| "wf_source_com Π (Call _ dst p actuals) =
     (case special_table p of
        Some desc ⇒
          classify_special desc actuals ≠ None ∧ list_all source_exp actuals ∧
          (case dst of None ⇒ False | Some x ⇒ x ≠ ret_var)
      | None ⇒
          (case Π p of
             None ⇒ False
           | Some decl ⇒
               length actuals = length (formals decl) ∧
               list_all source_exp actuals ∧
               (case dst of
                  None ⇒ True
                | Some x ⇒ x ≠ ret_var ∧ value_providing (body decl))))"
| "wf_source_com Π (Return _ e) = (case e of None ⇒ True | Some a ⇒ source_exp a)"
| "wf_source_com Π (Restore _) = False"
| "wf_source_com Π (Unwind _) = False"

fun no_return :: "'a acom ⇒ bool" where
  "no_return (Seq c1 c2) = (no_return c1 ∧ no_return c2)"
| "no_return (If _ _ c1 c2) = (no_return c1 ∧ no_return c2)"
| "no_return (While _ _ c) = no_return c"
| "no_return (Return _ _) = False"
| "no_return _ = True"

definition wf_proc_decl :: "(vname ⇒ bool) ⇒ 'a ptable ⇒ 'a gproc_decl ⇒ bool" where
  "wf_proc_decl 𝒢 Π decl ⟷
     distinct (formals decl) ∧
     list_all (valid_formal 𝒢) (formals decl) ∧
     wf_source_com Π (body decl)"

definition reserved_ret_var :: "(vname ⇒ bool) ⇒ bool" where
  "reserved_ret_var 𝒢 ⟷ ¬ 𝒢 ret_var"
```

The entry procedure is fixed: a program has exactly one, it is called `main`, and the parser rejects formals on it. Naming it here rather than threading it as a parameter means the compiler and its contract take a table and a callee list and nothing else.

```
definition prog_main_name :: pname where
  "prog_main_name = STR ''main''"
```

The entry body, looked up rather than passed alongside the table. The lookup cannot fail where `wf_source_program`

holds — its second conjunct is exactly that the entry is declared — so an undeclared entry is an invariant violation, and aborts in generated code rather than compiling to a plausible empty program.

```
definition main_body :: "'a ptable ⇒ 'a acom" where
  "main_body Π =
     (case Π prog_main_name of
        Some decl ⇒ body decl
      | None ⇒ Code.abort (STR ''main_body: entry procedure not declared'') (λ_. SKIP undefined))"
```

Deliberately not `[simp]`: `wf_source_program`'s entry conjunct has the shape `Π prog_main_name = Some (⦇formals = [], body = main_body Π⦈)`, against which this rule would rewrite `main_body Π` to itself.

```
definition wf_source_program :: "(vname ⇒ bool) ⇒ 'a ptable ⇒ bool" where
  "wf_source_program 𝒢 Π ⟷
     reserved_ret_var 𝒢 ∧
     Π prog_main_name = Some (⦇formals = [], body = main_body Π⦈) ∧
     wf_source_com Π (main_body Π) ∧ no_return (main_body Π) ∧
     (∀p decl. Π p = Some decl ⟶ wf_proc_decl 𝒢 Π decl) ∧
     (∀p. Π p ≠ None ⟶ special_table p = None)"
```

The compiler-input well-formedness defined in the compiler session unfolds through this pair before reaching `wf_source_com`; every site discharging that obligation shares the unfold skeleton, so it is collected here and the downstream definition adds itself to the same collection.

```
named_theorems wf_compile_input_simps

declare
  wf_proc_decl_def [wf_compile_input_simps]
  wf_source_program_def [wf_compile_input_simps]

lemma wf_source_com_source_com:
  "wf_source_com Π c ⟹ source_com c"
  by (induction c) (auto split: option.splits)

lemma wf_source_programD:
  assumes "wf_source_program 𝒢 Π"
  shows "reserved_ret_var 𝒢"
    and "Π prog_main_name = Some (⦇formals = [], body = main_body Π⦈)"
    and "wf_source_com Π (main_body Π)"
    and "no_return (main_body Π)"
    and "Π p = Some decl ⟹ wf_proc_decl 𝒢 Π decl"
    and "Π p = Some decl ⟹ special_table p = None"
    and "source_pi Π"
    and "source_com (main_body Π)"
  using assms wf_source_com_source_com
  unfolding wf_source_program_def source_pi_def wf_proc_decl_def by blast+
```

The semantics is inhabited

Every theorem above is conditional on a step or a run existing, so this theory ends by exhibiting one that exercises the parts most easily got wrong: a procedure call that pushes a frame, returns a value, and pops the frame again. Globals are empty here, so nothing the callee wrote survives except the returned value — which is what has to reach `x`.

```
definition witness_ret1_pi :: proc_table where
  "witness_ret1_pi p =
     (if p = STR ''ret1'' then Some (⦇formals = [], body = Return () (Some (N 1))⦈) else None)"

theorem pcompletes_witness:
  "∃t. pcompletes (λ_. False) witness_ret1_pi
         (Call () (Some (STR ''x'')) (STR ''ret1'') []) s t
     ∧ t (STR ''x'') = 1"
proof -
  let ?𝒢 = "λ_ :: vname. False"
  let ?fr = "[Frame s (Some (STR ''x''))]"
  let ?en = "enter_state ?𝒢 s"
  let ?s' = "?en(ret_var := ⟦N 1⟧e ?en)"
  let ?t  = "(combine_env ?𝒢 s ?s')(STR ''x'' := ⟦N 1⟧e ?en)"
  have q: "witness_ret1_pi (STR ''ret1'') = Some (⦇formals = [], body = Return () (Some (N 1))⦈)"
    by (simp add: witness_ret1_pi_def)
  have s1: "?𝒢, witness_ret1_pi ⊢ (Call () (Some (STR ''x'')) (STR ''ret1'') [], s, [])
              →p (Seq (Return () (Some (N 1))) (Restore ()), ?en, ?fr)"
    using q by (rule pstep_Call_parameterless)
  have s2: "?𝒢, witness_ret1_pi ⊢ (Seq (Return () (Some (N 1))) (Restore ()), ?en, ?fr)
              →p (Seq (Unwind ()) (Restore ()), ?s', ?fr)"
    by (intro Seq2 ReturnSome)
```

```
  have s3: "?𝒢, witness_ret1_pi ⊢ (Seq (Unwind ()) (Restore ()), ?s', ?fr) →p (SKIP (), ?t, [])"
    using UnwindAct[of ?𝒢 witness_ret1_pi "()" "()" ?s' s "Some (STR ''x'')" "[]"] by simp
  from s1 s2 s3
  have "pcompletes ?𝒢 witness_ret1_pi (Call () (Some (STR ''x'')) (STR ''ret1'') []) s ?t"
    by (meson star.refl star.step)
  moreover have "?t (STR ''x'') = 1" by simp
  ultimately show ?thesis by blast
qed

end
```

1.6 VIMP_Program

```
theory VIMP_Program
  imports VIMP_Proc
begin
```

Whole programs

`declared_global_vars` is the declared-global list in source order; a list, so finiteness is by type. `declared_global` is the derived classifier every program-level operation uses; procedure-level interfaces take the classifier as an explicit `vname ⇒ bool` argument instead.

```
record imp_prog =
  proc_rep :: "(pname * proc_decl) list"
  declared_global_vars :: "vname list"

definition declared_global :: "imp_prog ⇒ vname ⇒ bool" where
  "declared_global p x ⟷ x ∈ set (declared_global_vars p)"

lemma declared_global_iff [simp]:
  "declared_global p x ⟷ x ∈ set (declared_global_vars p)"
  by (simp add: declared_global_def)
```

The `program` parser fixes the entry name and rejects formals on it, so `main` is an ordinary `proc_rep` entry. `prog_main` is total through `main_body`'s aborting fallback; `wf_source_program`'s entry conjunct is what makes the lookup meaningful.

```
definition prog_table :: "imp_prog ⇒ proc_table" where
  "prog_table p = map_of (proc_rep p)"

definition prog_procs :: "imp_prog ⇒ pname list" where
  "prog_procs p = filter (λn. n ≠ prog_main_name) (map fst (proc_rep p))"

definition prog_main :: "imp_prog ⇒ com" where
  "prog_main p = main_body (prog_table p)"

lemma prog_main_eq_main_body [simp]: "main_body (prog_table p) = prog_main p"
  by (simp add: prog_main_def)
```

`mk_program` conses the entry onto `proc_rep`, so a program built through it satisfies `wf_source_program`'s entry conjunct by construction.

```
definition mk_program :: "(pname * proc_decl) list ⇒ com ⇒ vname list ⇒ imp_prog" where
  "mk_program ps m gv = imp_prog.make ((prog_main_name, (|formals = [], body = m|)) # ps) gv"

lemma mk_program_simps [simp]:
  "prog_table (mk_program ps m gv) = (map_of ps)(prog_main_name ↦ (|formals = [], body = m|))"
  "prog_main (mk_program ps m gv) = m"
  "declared_global_vars (mk_program ps m gv) = gv"
  "prog_main_name ∉ set (map fst ps) ⟹ prog_procs (mk_program ps m gv) = map fst ps"
  by (force simp: prog_table_def prog_main_def main_body_def prog_procs_def mk_program_def
      imp_prog.make_def filter_id_conv)+
```

The finite variable scope of an activation: every declared global, the activation's formals and body occurrences, and the reserved return location.

```
definition scope_vnames :: "imp_prog ⇒ pname ⇒ vname set" where
  "scope_vnames p owner =
    set (declared_global_vars p) ∪ {ret_var} ∪
    (case prog_table p owner of
      None ⇒ {} |
      Some decl ⇒ set (formals decl) ∪ com_vnames (body decl))"

lemma finite_scope_vnames [simp]: "finite (scope_vnames p owner)"
  unfolding scope_vnames_def by (auto split: option.splits)

definition scope_vnames_list :: "imp_prog ⇒ pname ⇒ vname list" where
  "scope_vnames_list p owner = sorted_list_of_set (scope_vnames p owner)"

lemma set_scope_vnames_list [simp]:
  "set (scope_vnames_list p owner) = scope_vnames p owner"
  unfolding scope_vnames_list_def by simp
```

Every variable some procedure of the program declares: a program-wide superset of any one activation's scope.

```
definition program_vars :: "imp_prog ⇒ vname list" where
```

```
  "program_vars p =
     remdups (concat (map (scope_vnames_list p) (prog_main_name # prog_procs p)))"

end
```

1.7 VIMP_Grammar_Generated

```
theory VIMP_Grammar_Generated
  imports VIMP_Proc
begin
```

GENERATED FILE. Source: `manifests/vimp-grammar.yaml`; generator: `scripts/gen_vimp_isabelle.py`. Regenerate with the generator rather than hand-editing; a CI drift check compares regenerated output against this file.

Owns the grammar-level nonterminals, syntax, and lowering (`Vimp_Grammar_Tr`) shared by every VIMP quotation: expressions, statements, statement lists, actuals, and the two identifier-list shapes (`formals`/`ids`). Does not register `imp ⟦ ... ⟧` or the whole-program `program { ... }` quotation itself, and has no notion of `imp_prog` – those are `VIMP_Notation`'s concern (main-selection, `proc_rep` construction, and everything else specific to that record shape), which imports this theory and calls into `Vimp_Grammar_Tr` for the pieces that are the same regardless of what the surrounding quotation ultimately builds.

Two lowering rules stay hand-written inside `Vimp_Grammar_Tr` rather than being generated from `manifests/vimp-grammar.yaml` directly: Isabelle's binary `Num` literal decoding (`dest_num`/`read_num_const`, no Menhir analogue) and the compositional unary-minus rule (folds a numeral operand into a negative `N`, otherwise `Minus (N 0) x` – deliberately does NOT cancel a nested unary minus; `--x` lowers to `Minus (N 0) (Minus (N 0) (V x))`, matching the shipped CLI frontend, not to `x`). `_exp_zero`/`_exp_one` and `_stmt_call0`/`_stmt_callret0` are Isabelle-target realizations of a canonical rule that Isabelle's own mixfix grammar cannot express directly: `num_const` has no derivation for the literal `0` or `1` (`Num.num`'s `One`/`Bit0`/`Bit1` encoding has no zero, and `1` is the unwrapped base case), and mixfix lists have no empty derivation at all – both confirmed empirically, not just by inspection: an isolation load test with only the generic `actuals` production failed to parse `f()`.

```
nonterminal imp2_exp
nonterminal imp2_stmt
nonterminal imp2_if_stmt
nonterminal imp2_stmts
nonterminal imp2_stmts_opt
nonterminal imp2_actuals
nonterminal imp2_formals
nonterminal imp2_ids

syntax
  "_exp_var" :: "id_position => imp2_exp" ("_" 1000)
  "_exp_num" :: "num_const => imp2_exp" ("_" 1000)
  "_exp_uminus" :: "imp2_exp => imp2_exp" ("- _" [90] 90)
  "_exp_plus" :: "imp2_exp => imp2_exp => imp2_exp" ("_ + _" [70, 71] 70)
  "_exp_minus" :: "imp2_exp => imp2_exp => imp2_exp" ("_ - _" [70, 71] 70)
  "_exp_times" :: "imp2_exp => imp2_exp => imp2_exp" ("_ * _" [80, 81] 80)
  "_exp_div" :: "imp2_exp => imp2_exp => imp2_exp" ("_ '/ _" [80, 81] 80)
  "_exp_mod" :: "imp2_exp => imp2_exp => imp2_exp" ("_ % _" [80, 81] 80)
  "_exp_true" :: imp2_exp ("true" 1000)
  "_exp_false" :: imp2_exp ("false" 1000)
  "_exp_less" :: "imp2_exp => imp2_exp => imp2_exp" ("_ < _" [61, 61] 60)
  "_exp_less_eq" :: "imp2_exp => imp2_exp => imp2_exp" ("_ <= _" [61, 61] 60)
  "_exp_greater" :: "imp2_exp => imp2_exp => imp2_exp" ("_ > _" [61, 61] 60)
  "_exp_greater_eq" :: "imp2_exp => imp2_exp => imp2_exp" ("_ >= _" [61, 61] 60)
  "_exp_eq" :: "imp2_exp => imp2_exp => imp2_exp" ("_ == _" [51, 51] 50)
  "_exp_not_eq" :: "imp2_exp => imp2_exp => imp2_exp" ("_ != _" [51, 51] 50)
  "_exp_not" :: "imp2_exp => imp2_exp" ("! _" [90] 90)
  "_exp_and" :: "imp2_exp => imp2_exp => imp2_exp" ("_ && _" [40, 41] 40)
  "_exp_or" :: "imp2_exp => imp2_exp => imp2_exp" ("_ || _" [30, 31] 30)
  "_exp_paren" :: "imp2_exp => imp2_exp" ("'( _ ')" [0] 1000)
  "_stmt_skip" :: imp2_stmt ("skip ;" 61)
  "_stmt_assign" :: "id_position => imp2_exp => imp2_stmt" ("_ = _ ;" [900, 0] 61)
  "_stmt_return" :: "imp2_exp => imp2_stmt" ("return _ ;" [0] 61)
  "_stmt_return0" :: imp2_stmt ("return ;" 61)
  "_stmt_check" :: "imp2_exp => imp2_stmt" ("'_'_voblint'_check '( _ ') ;" [0] 61)
  "_stmt_if" :: "imp2_exp => imp2_stmts_opt => imp2_stmts_opt => imp2_if_stmt" ("if '( _ ') { _ } else { _ }" [0, 0,
0] 61)
  "_stmt_if_then" :: "imp2_exp => imp2_stmts_opt => imp2_if_stmt" ("if '( _ ') { _ }" [0, 0] 61)
  "_stmt_if_chain" :: "imp2_exp => imp2_stmts_opt => imp2_if_stmt => imp2_if_stmt" ("if '( _ ') { _ } else _" [0, 0,
0] 61)
  "_stmt_conditional" :: "imp2_if_stmt => imp2_stmt" ("_" [0] 61)
  "_stmt_while" :: "imp2_exp => imp2_stmts_opt => imp2_stmt" ("while '( _ ') { _ }" [0, 0] 61)
  "_stmt_call" :: "id_position => imp2_actuals => imp2_stmt" ("_'( _ ') ;" [1000, 0] 61)
  "_stmt_callret" :: "id_position => id_position => imp2_actuals => imp2_stmt" ("_ = _'( _ ') ;" [900, 1000, 0] 61)
  "_stmts_one" :: "imp2_stmt => imp2_stmts" ("_" 61)
  "_stmts_seq" :: "imp2_stmts => imp2_stmt => imp2_stmts" ("_ _" [61, 61] 61)
  "_stmts_opt_none" :: imp2_stmts_opt ("")
```

```
  "_stmts_opt_some" :: "imp2_stmts => imp2_stmts_opt" ("_")
  "_actuals_one" :: "imp2_exp => imp2_actuals" ("_")
  "_actuals_cons" :: "imp2_exp => imp2_actuals => imp2_actuals" ("_, _")
  "_formals_one" :: "id_position => imp2_formals" ("_")
  "_formals_cons" :: "id_position => imp2_formals => imp2_formals" ("_, _")
  "_ids_one" :: "id_position => imp2_ids" ("_")
  "_ids_cons" :: "id_position => imp2_ids => imp2_ids" ("_, _")
  "_exp_zero" :: imp2_exp ("0" 1000)
  "_exp_one" :: imp2_exp ("1" 1000)
  "_stmt_call0" :: "id_position => imp2_stmt" ("_'(') ;" [1000] 61)
  "_stmt_callret0" :: "id_position => id_position => imp2_stmt" ("_ = _'(') ;" [900, 1000] 61)
```

ML ‹

```
structure Vimp_Grammar_Tr =
struct
  val c_N      = "VIMP_Syntax.N"
  val c_V      = "VIMP_Syntax.V"
  val c_Plus   = "VIMP_Syntax.exp.Plus"
  val c_Minus  = "VIMP_Syntax.exp.Minus"
  val c_Times  = "VIMP_Syntax.exp.Times"
  val c_Div = "VIMP_Syntax.exp.Div"
  val c_Mod = "VIMP_Syntax.exp.Mod"

  val c_Less   = "VIMP_Syntax.exp.Less"
  val c_LessEq = "VIMP_Syntax.exp.LessEq"
  val c_Greater = "VIMP_Syntax.exp.Greater"
  val c_GreaterEq = "VIMP_Syntax.exp.GreaterEq"
  val c_NotEq = "VIMP_Syntax.exp.NotEq"
  val c_Eq     = "VIMP_Syntax.exp.Eq"
  val c_Not    = "VIMP_Syntax.exp.Not"
  val c_And    = "VIMP_Syntax.exp.And"
  val c_Or     = "VIMP_Syntax.exp.Or"

  val c_SKIP   = "VIMP_Proc.gcom.SKIP"
  val c_Assign = "VIMP_Proc.gcom.Assign"
  val c_Seq    = "VIMP_Proc.gcom.Seq"
  val c_If     = "VIMP_Proc.gcom.If"
  val c_While  = "VIMP_Proc.gcom.While"
  val c_Call   = "VIMP_Proc.gcom.Call"
  val c_Return = "VIMP_Proc.gcom.Return"
  val c_Check  = "VIMP_Proc.gcom.Check"

  val c_None   = "Option.option.None"
  val c_Some   = "Option.option.Some"
  val c_Cons   = "List.list.Cons"
  val c_Nil    = "List.list.Nil"

  (* Parsed statements carry the unit annotation; elaboration assigns identities later. *)
  val annotated_ctors = [c_SKIP, c_Assign, c_If, c_While, c_Call, c_Return, c_Check]

  fun K name =
    if member (op =) annotated_ctors name
    then Const (name, dummyT) $ Const ("Product_Type.Unity", dummyT)
    else Const (name, dummyT)

  (* Decode Isabelle's Num binary structure: One=1, Bit0 n=2n, Bit1 n=2n+1.
     Leaf may also be a decimal-string Const (e.g. Const("20",_)) from the
     raw lexer. *)
  fun dest_num (Const (c, _)) =
        let val name = Long_Name.base_name c
        in if name = "One" then 1
           else case Int.fromString name of
                  SOME n => n
                | NONE => raise TERM ("Vimp_Grammar_Tr: not a num leaf", [Const (c, dummyT)])
        end
    | dest_num (Const (c, _) $ t) =
        let val name = Long_Name.base_name c
            val n    = dest_num t
        in if name = "Bit0" then 2 * n
           else if name = "Bit1" then 2 * n + 1
           else raise TERM ("Vimp_Grammar_Tr: not a num constructor", [Const (c, dummyT) $ t])
        end
    | dest_num t = raise TERM ("Vimp_Grammar_Tr: dest_num catchall", [t])

  fun read_num_const (Const ("_constify", _) $ t) = read_num_const t
    | read_num_const (Const ("_position", _) $ t) = read_num_const t
```

```
    | read_num_const ((Const ("_constrain", _) $ t) $ _) = read_num_const t
    | read_num_const (Free (s, _)) =
        (case Int.fromString s of
           SOME n => n
         | NONE => raise TERM ("Vimp_Grammar_Tr: not a numeral", [Free (s, dummyT)]))
    | read_num_const (Const (s, _)) =
        (case Int.fromString (Long_Name.base_name s) of
           SOME n => n
         | NONE => raise TERM ("Vimp_Grammar_Tr: not a numeral", [Const (s, dummyT)]))
    | read_num_const t = dest_num t

  fun neg_num n = K c_N $ HOLogic.mk_number HOLogic.intT (~ n)

  fun dest_id_position report_markup ctxt (Const ("_constrain", _) $ Free (s, _) $ m) =
        (case (report_markup, Term_Position.decode_position m) of
           (SOME markup, SOME (ps, _)) =>
             (List.app (fn p => Context_Position.report ctxt (#pos p) markup) ps; s)
         | _ => s)
    | dest_id_position _ _ (Free (s, _)) = s
    | dest_id_position _ _ t = raise TERM ("Vimp_Grammar_Tr: dest_id_position", [t])

  fun exp_tr ctxt t =
        (case Term.strip_comb t of
           (Const ("_exp_num", _), [n]) =>
             K c_N $ HOLogic.mk_number HOLogic.intT (read_num_const n)
         | (Const ("_exp_uminus", _), [a]) =>
             (case Term.strip_comb a of
                (Const ("_exp_num", _), [n]) => neg_num (read_num_const n)
              | (Const ("_exp_zero", _), []) =>
                  K c_N $ HOLogic.mk_number HOLogic.intT 0
              | (Const ("_exp_one", _), []) => neg_num 1
              | _ =>
                  K c_Minus $ (K c_N $ HOLogic.mk_number HOLogic.intT 0) $ exp_tr ctxt a)
         | (Const ("_exp_zero", _), []) => K c_N $ HOLogic.mk_number HOLogic.intT 0
         | (Const ("_exp_one", _), []) => K c_N $ HOLogic.mk_number HOLogic.intT 1
         | (Const ("_exp_var", _), [x0]) => K c_V $ (HOLogic.mk_literal (dest_id_position (SOME Markup.free) ctxt x0))
         | (Const ("_exp_plus", _), [a0, a2]) => K c_Plus $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_minus", _), [a0, a2]) => K c_Minus $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_times", _), [a0, a2]) => K c_Times $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_div", _), [a0, a2]) => K c_Div $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_mod", _), [a0, a2]) => K c_Mod $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_true", _), []) => K c_N $ (HOLogic.mk_number HOLogic.intT 1)
         | (Const ("_exp_false", _), []) => K c_N $ (HOLogic.mk_number HOLogic.intT 0)
         | (Const ("_exp_less", _), [a0, a2]) => K c_Less $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_less_eq", _), [a0, a2]) => K c_LessEq $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_greater", _), [a0, a2]) => K c_Greater $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_greater_eq", _), [a0, a2]) => K c_GreaterEq $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_eq", _), [a0, a2]) => K c_Eq $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_not_eq", _), [a0, a2]) => K c_NotEq $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_not", _), [a1]) => K c_Not $ (exp_tr ctxt a1)
         | (Const ("_exp_and", _), [a0, a2]) => K c_And $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_or", _), [a0, a2]) => K c_Or $ (exp_tr ctxt a0) $ (exp_tr ctxt a2)
         | (Const ("_exp_paren", _), [a1]) => exp_tr ctxt a1
         | _ => raise TERM ("Vimp_Grammar_Tr: exp_tr", [t]))

  fun actuals_tr ctxt (Const ("_actuals_one", _) $ x) = K c_Cons $ (exp_tr ctxt x) $ K c_Nil
    | actuals_tr ctxt (Const ("_actuals_cons", _) $ x $ rest) = K c_Cons $ (exp_tr ctxt x) $ (actuals_tr ctxt rest)
    | actuals_tr _ t = raise TERM ("Vimp_Grammar_Tr: actuals_tr", [t])

  fun stmts_tr ctxt (Const ("_stmts_one", _) $ x) = stmt_tr ctxt x
    | stmts_tr ctxt (Const ("_stmts_seq", _) $ xs $ x) = K c_Seq $ (stmts_tr ctxt xs) $ (stmt_tr ctxt x)
    | stmts_tr _ t = raise TERM ("Vimp_Grammar_Tr: stmts_tr", [t])
  and stmts_opt_tr ctxt (Const ("_stmts_opt_none", _)) = K c_SKIP
    | stmts_opt_tr ctxt (Const ("_stmts_opt_some", _) $ s) = stmts_tr ctxt s
    | stmts_opt_tr _ t = raise TERM ("Vimp_Grammar_Tr: stmts_opt_tr", [t])
  and stmt_tr ctxt t =
        (case Term.strip_comb t of
           (Const ("_stmt_call0", _), [x0]) => K c_Call $ (K c_None) $ (HOLogic.mk_literal (dest_id_position (SOME
Markup.skolem) ctxt x0)) $ K c_Nil
         | (Const ("_stmt_callret0", _), [x0, x2]) => K c_Call $ ((K c_Some $ (HOLogic.mk_literal (dest_id_position
(SOME Markup.free) ctxt x0)))) $ (HOLogic.mk_literal (dest_id_position (SOME Markup.skolem) ctxt x2)) $ K c_Nil
         | (Const ("_stmt_skip", _), []) => K c_SKIP
         | (Const ("_stmt_assign", _), [x0, a2]) => K c_Assign $ (HOLogic.mk_literal (dest_id_position (SOME Markup.free)
ctxt x0)) $ (exp_tr ctxt a2)
         | (Const ("_stmt_return", _), [a1]) => K c_Return $ ((K c_Some $ (exp_tr ctxt a1)))
         | (Const ("_stmt_return0", _), []) => K c_Return $ (K c_None)
```

```
        | (Const ("_stmt_check", _), [a2]) => K c_Check $ (HOLogic.mk_prod (HOLogic.mk_nat 0, HOLogic.mk_nat 0)) $
(exp_tr ctxt a2)
        | (Const ("_stmt_conditional", _), [a0]) => if_stmt_tr ctxt a0
        | (Const ("_stmt_while", _), [a2, a5]) => K c_While $ (exp_tr ctxt a2) $ (stmts_opt_tr ctxt a5)
        | (Const ("_stmt_call", _), [x0, a2]) => K c_Call $ (K c_None) $ (HOLogic.mk_literal (dest_id_position (SOME
Markup.skolem) ctxt x0)) $ (actuals_tr ctxt a2)
        | (Const ("_stmt_callret", _), [x0, x2, a4]) => K c_Call $ ((K c_Some $ (HOLogic.mk_literal (dest_id_position
(SOME Markup.free) ctxt x0)))) $ (HOLogic.mk_literal (dest_id_position (SOME Markup.skolem) ctxt x2)) $ (actuals_tr
ctxt a4)
        | _ => raise TERM ("Vimp_Grammar_Tr: stmt_tr", [t]))
  and if_stmt_tr ctxt t =
       (case Term.strip_comb t of
          (Const ("_stmt_if", _), [a2, a5, a9]) => K c_If $ (exp_tr ctxt a2) $ (stmts_opt_tr ctxt a5) $ (stmts_opt_tr
ctxt a9)
        | (Const ("_stmt_if_then", _), [a2, a5]) => K c_If $ (exp_tr ctxt a2) $ (stmts_opt_tr ctxt a5) $ (K c_SKIP)
        | (Const ("_stmt_if_chain", _), [a2, a5, a8]) => K c_If $ (exp_tr ctxt a2) $ (stmts_opt_tr ctxt a5) $ (if_stmt_tr
ctxt a8)
        | _ => raise TERM ("Vimp_Grammar_Tr: if_stmt_tr", [t]))

  fun formals_of ctxt (Const ("_formals_one", _) $ x) = [dest_id_position (SOME Markup.free) ctxt x]
    | formals_of ctxt (Const ("_formals_cons", _) $ x $ rest) = dest_id_position (SOME Markup.free) ctxt x :: formals_of
ctxt rest
    | formals_of _ t = raise TERM ("Vimp_Grammar_Tr: formals_of", [t])

  fun names_of ctxt (Const ("_ids_one", _) $ x) = [dest_id_position (SOME Markup.bound) ctxt x]
    | names_of ctxt (Const ("_ids_cons", _) $ x $ rest) = dest_id_position (SOME Markup.bound) ctxt x :: names_of ctxt
rest
    | names_of _ t = raise TERM ("Vimp_Grammar_Tr: names_of", [t])
end
›
```

end

1.8 VIMP_Notation

theory VIMP_Notation
  imports VIMP_Program VIMP_Grammar_Generated
begin

Concrete syntax

The `imp ⟦ ... ⟧` quotation produces an `com` without HOL string quotes or qualified constructor names. The `imp` keyword distinguishes command quotation from Pure premises; whole programs use the explicit `program` prefix.

Design is inspired by: https://awslabs.github.io/AutoCorrode/Unsorted/AutoCorrode/Micro_Rust_Examples.Basic_Micro_Rust.html and https://github.com/awslabs/AutoCorrode/blob/e234addc5e67f78cbff63defd24199578e8e1af3/Micro_Rust_Parsing_Frontend/Micro_Rust_Syntax.thy#L5

Inside the bracket: - bare identifiers become `V` literals (HOL string literals via `vname`) - numerals become `N` - arithmetic: +, -, *, /, `Div`, `Mod` unary minus `-n` on numerals becomes `N` with a negative int - comparisons: <, <=, >, >=, ==, != evaluate to an `int` 0/1 - constants: true/false lower to `N` 1/`N` 0

Example:

```
definition loop_prog :: VIMP_Proc.com where
      "loop_prog = imp \<lbrakk>
         x = 0;
         while (x < 20) { x = x + 1; }
       \<rbrakk>"
```

Whole-program syntax and its `imp_prog` lowering stay hand-written here, matching the grammar's `special: program_structure` boundary; every other nonterminal and lowering function comes from the generated `Vimp_Grammar_Tr` structure.

nonterminal imp2_funcs

syntax
```
  "_IMP2"        :: "imp2_stmts_opt ⇒ VIMP_Proc.com"        ("imp ⟦ _ ⟧" [0] 1000)

  "_PROGKW0"    :: "imp2_funcs ⇒ imp_prog"                ("program { _ }" [0] 1000)
  "_PROGKW"     :: "imp2_ids ⇒ imp2_funcs ⇒ imp_prog"      ("program { global _ ; _ }" [0, 0] 1000)
  "_funcs_nil"   :: imp2_funcs                                  ("")
  "_funcs_cons0" :: "id_position ⇒ imp2_stmts_opt ⇒ imp2_funcs ⇒ imp2_funcs"
    ("fun _'(') { _ } _" [1000, 0, 0] 1000)
  "_funcs_cons"  :: "id_position ⇒ imp2_formals ⇒ imp2_stmts_opt ⇒ imp2_funcs ⇒ imp2_funcs"
    ("fun _'( _ ') { _ } _" [1000, 0, 0, 0] 1000)
```

parse_translation ‹
```
  let
    val c_proc_decl_ext = "VIMP_Proc.gproc_decl.gproc_decl_ext"
    val c_Unity   = "Product_Type.Unity"
    val c_imp_prog = "VIMP_Program.mk_program"
    val c_Pair    = "Product_Type.Pair"

    val K = Vimp_Grammar_Tr.K
    val c_SKIP   = Vimp_Grammar_Tr.c_SKIP
    val c_Seq    = Vimp_Grammar_Tr.c_Seq
    val c_Return = Vimp_Grammar_Tr.c_Return
    val c_Some   = Vimp_Grammar_Tr.c_Some
    val c_None   = Vimp_Grammar_Tr.c_None
    val c_Cons   = Vimp_Grammar_Tr.c_Cons
    val c_Nil    = Vimp_Grammar_Tr.c_Nil

    (* Bodies stay raw parse trees until has_return has inspected them;
       mk_body_ret translates afterwards. The procedure name reports
       Markup.skolem, the same category a call site to it reports. *)
    fun funcs_tr ctxt (Const ("_funcs_nil", _)) = []
      | funcs_tr ctxt (Const ("_funcs_cons0", _) $ f $ pbody $ rest) =
          (Vimp_Grammar_Tr.dest_id_position (SOME Markup.skolem) ctxt f,
           [], SOME pbody, NONE) :: funcs_tr ctxt rest
      | funcs_tr ctxt (Const ("_funcs_cons", _) $ f $ formals $ pbody $ rest) =
          (Vimp_Grammar_Tr.dest_id_position (SOME Markup.skolem) ctxt f,
           Vimp_Grammar_Tr.formals_of ctxt formals, SOME pbody, NONE)
          :: funcs_tr ctxt rest
      | funcs_tr _ t = raise TERM ("VIMP_Notation: funcs_tr", [t])

    (* A bare "return;" is a Const without argument, matched by neither the
       _stmt_return case nor the application/abstraction fallbacks. *)
```

```
  fun has_return (Const ("_stmt_return", _) $ _) = true
    | has_return (Const ("_stmt_return0", _)) = true
    | has_return (t $ u) = has_return t orelse has_return u
    | has_return (Abs (_, _, b)) = has_return b
    | has_return _ = false

  fun check_distinct kind xs =
    (case duplicates (op =) xs of
       [] => ()
     | ds => error ("VIMP program: duplicate " ^ kind ^ ": " ^ commas_quote ds))

  (* A formal's per-call binding and a global's cross-call persistence are
     incompatible storage classes for one name. *)
  fun check_no_global_formal_collision decls funcs =
    List.app (fn (n, formals, _, _) =>
      case filter (member (op =) decls) formals of
        [] => ()
      | bad => error ("VIMP program: " ^ quote n ^ " declares global(s) as formal(s): "
                      ^ commas_quote bad)) funcs

  fun mk_names [] = K c_Nil
    | mk_names (n :: ns) = K c_Cons $ HOLogic.mk_literal n $ mk_names ns

  (* A trailing "return e" becomes an explicit Return appended to the body. *)
  fun mk_body_ret ctxt NONE NONE = K c_SKIP
    | mk_body_ret ctxt (SOME c) NONE = Vimp_Grammar_Tr.stmts_opt_tr ctxt c
    | mk_body_ret ctxt NONE (SOME e) = K c_Return $ (K c_Some $ Vimp_Grammar_Tr.exp_tr ctxt e)
    | mk_body_ret ctxt (SOME c) (SOME e) =
        K c_Seq $ Vimp_Grammar_Tr.stmts_opt_tr ctxt c
                $ (K c_Return $ (K c_Some $ Vimp_Grammar_Tr.exp_tr ctxt e))

  fun mk_proc_rep ctxt [] = K c_Nil
    | mk_proc_rep ctxt ((p, formals, body, result) :: rest) =
        K c_Cons
          $ (K c_Pair $ HOLogic.mk_literal p
              $ (K c_proc_decl_ext $ mk_names formals $ mk_body_ret ctxt body result $ K c_Unity))
          $ mk_proc_rep ctxt rest

  (* dummyT constructors only; type inference runs after the translation. *)
  fun prog_tr ctxt decls funcs_t =
    let
      val funcs = funcs_tr ctxt funcs_t
      val _ = check_distinct "procedure" (map (fn (n, _, _, _) => n) funcs)
      val _ = check_distinct "declared global" decls
      val _ = List.app (fn (n, formals, _, _) =>
                check_distinct ("formal parameter of " ^ quote n) formals) funcs
      val _ = check_no_global_formal_collision decls funcs
      val (mains, procs) = List.partition (fn (n, _, _, _) => n = "main") funcs
      val main_ast =
        (case mains of
           [("main", [], NONE, NONE)] => NONE
         | [("main", [], SOME body, NONE)] =>
             if has_return body then error "VIMP program: main may not return" else SOME body
         | [("main", _, _, SOME _)] => error "VIMP program: main may not return"
         | [("main", _, _, NONE)] => error "VIMP program: main must have no formals"
         | [] => error "VIMP program: missing 'fun main() { ... }'"
         | _  => error "VIMP program: more than one 'fun main()'")
      val proc_rep = mk_proc_rep ctxt procs
      val main = mk_body_ret ctxt main_ast NONE
      val decl_globals = mk_names decls
    in K c_imp_prog $ proc_rep $ main $ decl_globals end
in
  [("_IMP2", fn ctxt => fn [t] => Vimp_Grammar_Tr.stmts_opt_tr ctxt t | _ => raise Match),
   ("_PROGKW0", fn ctxt => fn [fs] => prog_tr ctxt [] fs | _ => raise Match),
   ("_PROGKW",
    fn ctxt => fn [g, fs] => prog_tr ctxt (Vimp_Grammar_Tr.names_of ctxt g) fs
             | _ => raise Match)]
end
›
```

Smoke tests

Four translations, each stated as the term it elaborates to: a branch and a check through the generated statement translation, and a program with a global and one with a procedure taking arguments through the whole-program translation hand-written here. The theory stops loading if a translation stops elaborating or elaborates to

something else, which no theorem elsewhere would notice – nothing in the proved pipeline reads concrete syntax.

```
lemma "imp ⟦ if (x < 10) { x = 0; } else { x = 1; } ⟧
  = gcom.If () (Less (V STR ''x'') (N 10)) (Assign () STR ''x'' (N 0))
    (Assign () STR ''x'' (N 1))"
  by eval
lemma "(imp ⟦ __voblint_check(!(x == 0)); ⟧ :: com)
  = Check () (0, 0) (exp.Not (exp.Eq (V STR ''x'') (N 0)))"
  by eval
lemma "(program { global Gx; fun ping() { Gx = Gx + 1; } fun main() { ping(); } } :: imp_prog)
  = ⦇proc_rep =
    [(STR ''main'', ⦇formals = [], body = Call () None STR ''ping'' []⦈),
    (STR ''ping'',
    ⦇formals = [],
    body = Assign () STR ''Gx'' (Plus (V STR ''Gx'') (N 1))⦈)],
    declared_global_vars = [STR ''Gx'']⦈"
  by eval
lemma "(program { fun add(a, b) { skip; return a + b; } fun main() { r = add(1, 2); } } :: imp_prog)
  = ⦇proc_rep =
    [(STR ''main'',
    ⦇formals = [], body = Call () (Some STR ''r'') STR ''add'' [N 1, N 2]⦈),
    (STR ''add'',
    ⦇formals = [STR ''a'', STR ''b''],
    body =
    Seq (SKIP ()) (Return () (Some (Plus (V STR ''a'') (V STR ''b''))))⦈)],
    declared_global_vars = []⦈"
  by eval

end
```

Chapter 2

Voblint_Domain

2.1 Abstract_Domain

```
theory Abstract_Domain
  imports "TD.Warrowing"
begin

unbundle lattice_syntax
```

Abstract value domains

An executable domain supplies the required lattice operations, the widening and narrowing the solver applies (the vendored `warrowing` class), and an exact emptiness test. Sound domains add a concretization into integers. Abstract states and control-flow reachability are separate modules.

Exact and sound emptiness

An emptiness test is *exact* for a concretization when it holds of exactly the values that describe nothing, and *sound* when every value it holds of describes nothing, with no converse. Each layer states which of the two its test is: a single domain's and a single state's test are exact, the combined state's only sound, because two analyses can each describe some value while their intersection is empty.

```
definition sound_emptiness :: "('s ⇒ bool) ⇒ ('s ⇒ 'c set) ⇒ bool" where
  "sound_emptiness e γ' ⟷ (∀x. e x ⟶ γ' x = {})"

definition exact_emptiness :: "('s ⇒ bool) ⇒ ('s ⇒ 'c set) ⇒ bool" where
  "exact_emptiness e γ' ⟷ (∀x. e x ⟷ γ' x = {})"

lemma sound_emptinessI [intro]: "(⋀x. e x ⟹ γ' x = {}) ⟹ sound_emptiness e γ'"
  unfolding sound_emptiness_def by blast

lemma sound_emptinessD [dest]: "sound_emptiness e γ' ⟹ e x ⟹ γ' x = {}"
  unfolding sound_emptiness_def by blast

lemma exact_emptinessI [intro]: "(⋀x. e x ⟷ γ' x = {}) ⟹ exact_emptiness e γ'"
  unfolding exact_emptiness_def by blast

lemma exact_emptinessD: "exact_emptiness e γ' ⟹ e x ⟷ γ' x = {}"
  unfolding exact_emptiness_def by blast

lemma exact_emptiness_sound: "exact_emptiness e γ' ⟹ sound_emptiness e γ'"
  unfolding exact_emptiness_def sound_emptiness_def by blast
```

Executable and sound domains

The vendored `widening` and `narrowing` classes constrain their type parameter (`'a::order`) instead of naming `order` as a superclass, so a class extending them starts from `order` and would reject $\bot$ and $\top$ in its axioms. The lattice classes therefore enter the same way, through the sort of the first parameter, and `numeric_domain` repeats that sort.

```
class executable_domain = warrowing +
  fixes is_empty :: "'a::{bounded_semilattice_sup_bot, order_top} ⇒ bool"
  fixes to_string :: "'a ⇒ String.literal"
```

A numeric domain adds a concretization `gamma` to an executable domain: bottom denotes no integer, top every integer, the order is sound for `gamma`, and `is_empty` decides emptiness exactly.

```
class numeric_domain = executable_domain +
  fixes gamma :: "'a::{bounded_semilattice_sup_bot, order_top} ⇒ int set" ("γ")
  assumes gamma_bot[simp]: "γ ⊥ = {}"
  assumes gamma_top[simp]: "γ ⊤ = UNIV"
  assumes gamma_mono: "a ≤ b ⟹ γ a ⊆ γ b"
  assumes is_empty_correct: "is_empty a ⟷ γ a = {}"

lemma exact_emptiness_is_empty: "exact_emptiness is_empty (γ :: 'a::numeric_domain ⇒ int set)"
  by (rule exact_emptinessI) (rule is_empty_correct)
```

`executable_domain` carries exactly the executable per-element operations a concrete domain's runtime representation needs: the lattice structure, the solver's `widen` and `narrow`, `is_empty` (a finite decision procedure on every real instance – Interval's bound comparison, Sign's constructor match, ...), and `to_string` for reporting a solved value back to a caller. `numeric_domain` extends it with `gamma`, which is not executable in general (an infinite `int set`) and exists purely to state and prove soundness. Splitting the class this way keeps `gamma` out of the

type-class dictionary that code generation must materialize for any constant that only needs `is_empty` (the finite witness-bottom tests over a `default_st`, in particular): requesting `'a::executable_domain` there never drags γ's code equation into the dependency closure, even though every `numeric_domain` instance is automatically a `executable_domain` instance too.

`is_empty` is a semantic classifier, not a structural equality test against `bot`: it mirrors Goblint's own `Lattice.Bot` signature (https://github.com/goblint/analyzer/blob/master/src/domain/lattice.ml): `val is_bot: t -> bool` is a per-domain operation there too, not a generic derived test. Goblint's own default implementation (`IntDomain0.Std` in https://github.com/goblint/analyzer/blob/master/src/cdomain/value/cdomains/intDomain0.ml) in fact defines it as structural equality against one canonical `bot_of` value, since most of its domains keep a single bottom representation (Interval's `bot () = None`, normalized on every operation). Voblint cannot take that shortcut: some Voblint domains admit representations with more than one empty-denoting value that are never normalized away (Interval's inverted bound pairs, e.g. `Ivl (Fin (5::'c)) (Fin (- 1))`, none of them favored over `bot` itself), so `a = bot` would silently miss some of them. Fixing `is_empty` as its own class operation, correct against γ rather than against $\bot$, makes every `numeric_domain` instance responsible for its own exact emptiness test, the same obligation every domain already carries for γ itself. The lattice constant `bot` stays the canonical representative; `is_empty` answers a different question (what a value denotes), and a proof that genuinely needs the canonical element still writes `a = bot` directly.

Concretization bounds

```
lemma gamma_sup_ub1[intro]: "γ a ⊆ γ (a ⊔ b)" for a b :: "'a::numeric_domain"
  by (rule gamma_mono[OF sup_ge1])

lemma gamma_sup_ub2[intro]: "γ b ⊆ γ (a ⊔ b)" for a b :: "'a::numeric_domain"
  by (rule gamma_mono[OF sup_ge2])
```

Emptiness is downward closed under the abstract order: it follows from `gamma_mono` alone, with no per-domain fact needed. This is what lets the generic transfer dispatcher's short-circuit stay monotone: a smaller input can only be witness-empty *more* often than a larger one, never less.

```
lemma is_empty_antimono:
  "a ≤ b ⟹ is_empty b ⟹ is_empty a" for a b :: "'a::numeric_domain"
  using gamma_mono unfolding is_empty_correct by blast
```

Domains with widening

`warrow` is deliberately not join-like: it takes the narrowing branch whenever `b` $\leq$ `a`, and narrowing moves the result down toward `b`, not up past `a`, so no `a` $\leq$ `a` $\nabla\Delta$ `b` counterpart to `warrowing_properties` holds in general. What does hold unconditionally in that regime is the matching upper bound on the `a` side, `warrow_le_when_le`, together with `warrowing_properties`' `b` $\leq$ `a` $\nabla\Delta$ `b` that gives the full sandwich `b` $\leq$ `a` $\nabla\Delta$ `b` $\leq$ `a`. `warrow_idem` is the fixpoint case of that sandwich.

Printing values

How a solved value is shown, following Goblint's notation (`1+3`$\mathbb{Z}$, $\top$, $\bot$). Nothing is proved about it: it is not part of any soundness claim.

A `String.literal` holds ASCII only, so a mathematical symbol is written as its Isabelle symbol name without the backslash — `<int>` for $\mathbb{Z}$ — and the CLI decodes these tokens into Unicode before rendering. No variable name, numeral or bracket a printer emits contains `<`, so the tokens are unambiguous.

```
abbreviation (input) sym_bottom :: String.literal where "sym_bottom ≡ STR ''<bottom>''"
abbreviation (input) sym_top :: String.literal where "sym_top ≡ STR ''<top>''"
abbreviation (input) sym_int :: String.literal where "sym_int ≡ STR ''<int>''"
abbreviation (input) sym_infinity :: String.literal where "sym_infinity ≡ STR ''<infinity>''"
abbreviation (input) sym_le :: String.literal where "sym_le ≡ STR ''<le>''"
abbreviation (input) sym_ge :: String.literal where "sym_ge ≡ STR ''<ge>''"
abbreviation (input) sym_and :: String.literal where "sym_and ≡ STR ''<and>''"

fun string_of_nat :: "nat ⇒ String.literal" where
  "string_of_nat n =
     (if n < 10 then String.implode [char_of (n + 48)]
      else string_of_nat (n div 10) + String.implode [char_of (n mod 10 + 48)])"

definition string_of_int :: "int ⇒ String.literal" where
  "string_of_int i =
     (if i < 0 then STR ''-'' + string_of_nat (nat (- i))
      else string_of_nat (nat i))"
```

end

2.2 Reachability_Lift

```
theory Reachability_Lift
  imports State_Concretization "HOL-Library.Monad_Syntax"
begin
```

Explicit reachability over an abstract carrier

The outer `Bot` represents an unreachable control-flow point; `Lifted a` represents a reachable point carrying `a`. The construction provides the order, joins, solver updates, concretization, and normalized transfer combinators independently of any particular abstract-state representation.

Mirrors Goblint's lifted-domain construction (https://github.com/goblint/analyzer/blob/master/src/framework/analyses.ml): `module Dom (LD: Lattice.S) = struct include Lattice.LiftConf (...) (LD) ... end`, whose outer `'Bot` means dead code and whose `unlift` only accepts `'Lifted x`. Reachability is carried as an explicit tag on top of the analysis-local state, not rediscovered from the state's own contents. `Bot` mirrors Goblint's lifted `'Bot`; `Lifted` mirrors `'Lifted`. Goblint's lift also carries an outer `'Top`, but not merely as a substitute for a missing payload `top`: `Lattice.LiftConf`'s `join`/`widen` fall back to `'Top` whenever the base domain's own `join`/`widen` raises `TopValue` or `Uncomparable`, i.e. whenever the base domain is partial enough that two of its values have no expressible join, and `unlift` rejects `'Top` exactly like `'Bot`. Every `numeric_domain` instance here is instead a total `bounded_semilattice_sup_bot` with its own `top`: $\sqcup$ and ∇ always return a value, so a lifted operation here can never fail the way Goblint's can, and `Lifted top` already says everything an outer `'Top` would. This lift therefore stays two-valued. At the raw lifted representation, `Bot` is strictly below every `Lifted a` regardless of `a` – including `Lifted bot`, which remains distinct from `Bot` even though `a` is itself locally bottom – since dead-by-construction and reachable-but-locally-bottom are the two concepts this lift exists to keep separate. The normalization discipline developed below (`normalize_lift`, `normalized_lift`) then layers a stronger, Voblint-specific invariant on top: every solver-facing payload an arbitrary caller-supplied predicate `empty_pred` classifies as bottom is additionally canonicalized to structural `Bot`, so `Lifted a` with an `empty_pred`-bottom `a` never actually escapes a value-producing step in practice, even though the raw datatype and order permit it. This theory does not know or assume that `empty_pred` means semantic emptiness – `Nonrelational_Reachability` is where `empty_pred` is instantiated to `is_empty_state` and proved to coincide with an empty concretization.

The lifted order and lattice structure

```
datatype (plugins del: quickcheck_narrowing) 'a lifted =
  Bot | Lifted 'a
```

The vendored TD solver stores values at sort `bounded_semilattice_sup_bot` (`TD_side`'s `T :: ('x,'g,'d::bounded_semilattice_s eqsT`), so `lifted` instantiates that combined structure directly in one step – order, bottom, and join together – rather than building it up through the three separate parent classes (`order`, `bot`, `semilattice_sup`) and bridging them with intermediate `instance` declarations. This intentionally gives up the weaker standalone arities that instantiating each parent separately would have provided (`'a::type ⇒ 'a lifted :: bot` and `'a::order ⇒ 'a lifted :: order`), for an arbitrary or merely ordered payload; consequently the generic lifted lemmas below (`gamma_lift_mono`, `map_lift_mono`, `normalize_lift_mono`, `transfer_lift_mono`, `transfer_lift2_mono`) and the standalone widening/narrowing instances are stated at `semilattice_sup` as well. A caller that only needs the weak standalone `bot` arity does not automatically satisfy this, and has to restate at the sort its real call sites already require.

```
instantiation lifted :: (semilattice_sup) bounded_semilattice_sup_bot
begin

fun less_eq_lifted :: "'a lifted ⇒ 'a lifted ⇒ bool" where
    "less_eq_lifted Bot _ = True"
  | "less_eq_lifted (Lifted _) Bot = False"
  | "less_eq_lifted (Lifted a) (Lifted b) = (a ≤ b)"

definition less_lifted :: "'a lifted ⇒ 'a lifted ⇒ bool" where
  "less_lifted x y = (x ≤ y ∧ ¬ y ≤ x)"

definition bot_lifted :: "'a lifted" where "bot_lifted = Bot"

fun sup_lifted :: "'a lifted ⇒ 'a lifted ⇒ 'a lifted" where
    "sup_lifted Bot y = y"
  | "sup_lifted x Bot = x"
  | "sup_lifted (Lifted a) (Lifted b) = Lifted (a ⊔ b)"

instance
proof intro_classes
```

```
  fix x y z :: "'a lifted"
  show "(x < y) = (x ≤ y ∧ ¬ y ≤ x)" unfolding less_lifted_def by (rule refl)
  show "x ≤ x" by (cases x) simp_all
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z" by (cases x; cases y; cases z) simp_all
  show "x ≤ y ⟹ y ≤ x ⟹ x = y" by (cases x; cases y) simp_all
  show "x ≤ x ⊔ y" by (cases x; cases y) simp_all
  show "y ≤ x ⊔ y" by (cases x; cases y) simp_all
  show "y ≤ x ⟹ z ≤ x ⟹ y ⊔ z ≤ x" by (cases x; cases y; cases z) simp_all
  show "bot ≤ x" by (cases x) (simp_all add: bot_lifted_def)
qed

end

lemma bot_lifted_eq [simp]: "(bot :: 'a::semilattice_sup lifted) = Bot"
  unfolding bot_lifted_def by (rule refl)
```

The warrowing update rule needs `'a lifted` itself to carry `widen`/`narrow` once the executable trees it runs store a lifted payload. `Bot` keeps the same role here as in the `sup` instance above: a structural placeholder, neutral for `widen`/`narrow` against a `Lifted` operand, with only the `Lifted`/`Lifted` case deferring to the base domain's own operation. `quickcheck_narrowing` is disabled on the `lifted` datatype declaration above: Isabelle/HOL's own built-in `narrowing` class (for Quickcheck's narrowing-based counterexample search) collides with the vendored solver's `narrowing` class of the same base name, and the datatype package's default Quickcheck plugin registers an arity for `lifted` under that colliding name before this instantiation gets a chance to.

```
instantiation lifted :: ("{semilattice_sup,widening}") widening
begin

fun widen_lifted :: "'a lifted ⇒ 'a lifted ⇒ 'a lifted" where
  "widen_lifted Bot y = y"
| "widen_lifted x Bot = x"
| "widen_lifted (Lifted a) (Lifted b) = Lifted (a ▽ b)"

instance proof intro_classes
  fix x y :: "'a lifted"
  show "x ≤ x ▽ y" by (cases x; cases y) (simp_all add: widen_ge1)
  show "y ≤ x ▽ y" by (cases x; cases y) (simp_all add: widen_ge2)
qed

end

instantiation lifted :: ("{semilattice_sup,narrowing}") narrowing
begin

fun narrow_lifted :: "'a lifted ⇒ 'a lifted ⇒ 'a lifted" where
  "narrow_lifted Bot y = Bot"
| "narrow_lifted (Lifted a) Bot = Bot"
| "narrow_lifted (Lifted a) (Lifted b) = Lifted (a △ b)"

instance proof intro_classes
  fix a b :: "'a lifted"
  show "b ≤ a ⟹ b ≤ a △ b"
    by (cases a; cases b) (simp_all add: narrow_ge)
  show "b ≤ a ⟹ a △ b ≤ a"
    by (cases a; cases b) (simp_all add: narrow_le)
qed

end
```

Generic concretization

Reading a lifted value back: `Bot` denotes no state at all, so it concretizes to the empty set rather than to whatever the underlying domain's own bottom denotes. That difference is the whole point of the lift – an unreachable point and a point whose every coordinate is bottom are distinguishable here, and only the former is empty by construction.

```
definition gamma_lift :: "('a ⇒ 'b set) ⇒ 'a lifted ⇒ 'b set" where
  "gamma_lift gam x = (case x of Bot ⇒ {} | Lifted a ⇒ gam a)"

lemma gamma_lift_Bot [simp]: "gamma_lift gam Bot = {}"
  unfolding gamma_lift_def by simp

lemma gamma_lift_Lifted [simp]: "gamma_lift gam (Lifted a) = gam a"
  unfolding gamma_lift_def by simp
```

```
lemma gamma_lift_mono:
  fixes x y :: "'a::semilattice_sup lifted"
  assumes gam_mono: "⋀a b. a ≤ b ⟹ gam a ⊆ gam b"
    and le: "x ≤ y"
  shows "gamma_lift gam x ⊆ gamma_lift gam y"
  using le by (cases x; cases y) (auto simp: gam_mono)
```

gamma_liftE recovers the Lifted payload a lifted membership witnesses: Bot concretizes to {}, so any s ∈ gamma_lift gam x already forces x = Lifted a for some a with s ∈ gam a. Named as an eliminator because its witness is exactly this structural case split, replacing the repeated manual obtain ... where "... = Lifted ..." "s ∈ ..." by (cases ...) auto pattern with one rule application.

```
lemma gamma_liftE [elim]:
  assumes "s ∈ gamma_lift gam x"
  obtains a where "x = Lifted a" and "s ∈ gam a"
  using assms by (cases x) auto
```

Generic reachability combinators

bind_lift is the reachability monad's Kleisli bind, exactly (≫=)'s shape with Bot for None and Lifted for Some: Bot in, Bot out, without ever calling f – matching Goblint's unlift being called before dispatch rather than inside each analysis's assign/branch. map_lift is its functor counterpart (Option.map's analogue), for pushing a total, never-re-collapsing operation through the lift. Both are representation-independent: neither one inspects whether a Lifted payload happens to be semantically empty. That is a separate concern, taken up below in *Canonical-bottom normalization*.

```
fun bind_lift :: "'a lifted ⇒ ('a ⇒ 'b lifted) ⇒ 'b lifted" where
  "bind_lift Bot f = Bot"
| "bind_lift (Lifted a) f = f a"

lemma bind_lift_left_identity: "bind_lift (Lifted a) f = f a" by simp

lemma bind_lift_right_identity [simp]: "bind_lift x Lifted = x"
  by (cases x) simp_all
```

gamma_lift_bindI follows bind_lift's own structure: a witness for xs together with a step that carries any such witness through f gives a witness for the bind. Targets the sequential And/Or recursions that chain one filtering step's result into the next, replacing their manual obtain d' where "... = Lifted d'" "s ∈ ..." destructuring with one cases-driven rule.

```
lemma gamma_lift_bindI [intro]:
  assumes xs: "s ∈ gamma_lift gam xs"
      and step: "⋀a. s ∈ gam a ⟹ s ∈ gamma_lift gam' (f a)"
  shows "s ∈ gamma_lift gam' (bind_lift xs f)"
  using xs by (cases xs) (auto intro: step)
```

Registers bind_lift under (≫=)'s ad hoc overloading, so x >>= f and do { a <- x; f a } both parse to exactly bind_lift x f – a pure syntax translation resolved at elaboration time, so every lemma about bind_lift applies unchanged to code written with do notation. Registered here, immediately after bind_lift's definition, so the reachability combinators below can use do notation too.

```
adhoc_overloading Monad_Syntax.bind == bind_lift

definition map_lift :: "('a ⇒ 'b) ⇒ 'a lifted ⇒ 'b lifted" where
  "map_lift f x = do { a <- x; Lifted (f a) }"

lemma map_lift_Bot [simp]: "map_lift f Bot = Bot"
  unfolding map_lift_def by simp

lemma map_lift_Lifted [simp]: "map_lift f (Lifted a) = Lifted (f a)"
  unfolding map_lift_def by simp
```

map_lift never invents or destroys the dead point, so the two carriers agree on the coarsest bottom test there is.

```
lemma map_lift_eq_Bot_iff [simp]: "map_lift f x = Bot ⟷ x = Bot"
  by (cases x) simp_all

lemma map_lift_mono:
  fixes x y :: "'a::semilattice_sup lifted"
  assumes f_mono: "⋀a b. a ≤ b ⟹ f a ≤ (f b :: 'b::semilattice_sup)"
    and le: "x ≤ y"
  shows "map_lift f x ≤ map_lift f y"
  using le by (cases x; cases y) (auto simp: f_mono)
```

```
lemma map_lift_sup:
  fixes x y :: "'a::semilattice_sup lifted"
  assumes f_sup: "⋀a b. f (a ⊔ b) = (f a ⊔ f b :: 'b::semilattice_sup)"
  shows "map_lift f (x ⊔ y) = map_lift f x ⊔ map_lift f y"
  by (cases x; cases y) (simp_all add: f_sup)

lemma map_lift_id [simp]: "map_lift id x = x"
  by (cases x) simp_all

lemma map_lift_comp [simp]: "map_lift g (map_lift f x) = map_lift (g ∘ f) x"
  by (cases x) simp_all
```

Collapsing back to the plain, bot-carrying type

`collapse_lift` goes the opposite direction from `normalize_lift`: where `normalize_lift` turns a domain-produced witness-bottom value into structural `Bot`, `collapse_lift` turns `Bot` back into the plain type's own `bot`, so a caller whose interface only has room for the plain type – not the `lifted` wrapper – can still receive a more precise lifted result. Unlike `normalize_lift`, no predicate is needed: `Bot` already says the value is empty, and `bot` is exactly the plain type's own representative of that.

```
fun collapse_lift :: "'a::bot lifted ⇒ 'a" where
    "collapse_lift Bot = bot"
  | "collapse_lift (Lifted a) = a"

lemma collapse_lift_mono:
  fixes x y :: "'a::bounded_semilattice_sup_bot lifted"
  assumes "x ≤ y"
  shows "collapse_lift x ≤ collapse_lift y"
  using assms by (cases x; cases y) simp_all
```

The soundness counterpart: a witness against `gam` through the lifted carrier survives collapsing, since `Bot` can only arise when there is no witness at all (`gam bot = {}` rules that reading out).

```
lemma gamma_collapse_lift:
  assumes "s ∈ gamma_lift gam x" and "gam bot = {}"
  shows "s ∈ gam (collapse_lift x)"
  using assms by (cases x) simp_all
```

`collapse_lift` commutes with `map_lift f` whenever `f` itself sends `bot` to `bot`: both sides land on `Bot`'s collapse, `bot`, when the carrier is `Bot`, and agree trivially on `Lifted a`. This is what lets the map from an executable state to its function (`f`) be applied either before or after collapsing a `lifted` result, so a commutation theorem proved for the `lifted` level transports to the collapsed, plain-type level for free.

```
lemma collapse_lift_map_lift:
  assumes "f bot = bot"
  shows "collapse_lift (map_lift f x) = f (collapse_lift x)"
  using assms by (cases x) simp_all
```

Canonical-bottom normalization

`normalize_lift` is the one Voblint-specific piece on top of the generic combinators above: it turns a domain-produced witness-bottom raw result back into the canonical structural `Bot`, the counterpart of Goblint's transfer functions raising `Deadcode` when their own computation lands on bottom. `transfer_lift`/`transfer_lift2` compose `bind_lift` with `normalize_lift`: the shape every generic edge/combine dispatcher actually calls.

```
definition normalize_lift :: "('a ⇒ bool) ⇒ 'a ⇒ 'a lifted" where
  "normalize_lift empty_pred a = (if empty_pred a then Bot else Lifted a)"

lemma normalize_lift_bot [simp]: "empty_pred a ⟹ normalize_lift empty_pred a = Bot"
  unfolding normalize_lift_def by simp

lemma normalize_lift_not_bot [simp]:
  "¬ empty_pred a ⟹ normalize_lift empty_pred a = Lifted a"
  unfolding normalize_lift_def by simp
```

When `empty_pred` exactly tracks `gam`'s own emptiness, normalizing never changes what a payload concretizes to: the `if` only chooses between `gam a` and `{}`, and those coincide exactly when `empty_pred a` does. This is the generic fact every `gamma_*_lift` correctness statement (`Nonrelational_Reachability`'s `is_empty_state_lift_iff`) reduces to, once `empty_pred`'s own iff-correctness is in hand.

```
lemma gamma_normalize_lift:
  assumes "empty_pred a ⟷ gam a = {}"
  shows "gamma_lift gam (normalize_lift empty_pred a) = gam a"
```

```
  using assms unfolding normalize_lift_def by auto

definition transfer_lift ::
  "('b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ 'a lifted ⇒ 'b lifted" where
  "transfer_lift empty_pred f x = do { a <- x; normalize_lift empty_pred (f a) }"

definition transfer_lift2 ::
  "('c ⇒ bool) ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'a lifted ⇒ 'b lifted ⇒ 'c lifted" where
  "transfer_lift2 empty_pred f x y =
     do { a <- x; b <- y; normalize_lift empty_pred (f a b) }"

lemma transfer_lift_Bot [simp]: "transfer_lift empty_pred f Bot = Bot"
  unfolding transfer_lift_def by simp

lemma transfer_lift_Lifted [simp]:
  "transfer_lift empty_pred f (Lifted a) = normalize_lift empty_pred (f a)"
  unfolding transfer_lift_def by simp

lemma transfer_lift2_Bot_left [simp]: "transfer_lift2 empty_pred f Bot y = Bot"
  unfolding transfer_lift2_def by simp

lemma transfer_lift2_Bot_right [simp]: "transfer_lift2 empty_pred f (Lifted a) Bot = Bot"
  unfolding transfer_lift2_def by simp

lemma transfer_lift2_Lifted [simp]:
  "transfer_lift2 empty_pred f (Lifted a) (Lifted b) = normalize_lift empty_pred (f a b)"
  unfolding transfer_lift2_def by simp
```

`normalized_lift` names the discipline every solver-facing lifted value is meant to keep: `Bot` is the *sole* representation this construction lets a caller-supplied `empty_pred` classification collapse to – a `Lifted` payload `empty_pred` still calls bottom is a representation the framework never lets escape a value-producing step, not a second, competing way to say the same thing. `normalize_lift`'s own `if` makes every one of its outputs self-normalized: the `Lifted` branch only fires when `empty_pred` already said no. `transfer_lift`/`transfer_lift2` route every domain transfer step through exactly this `if`, so no transfer step can ever hand back a `Lifted` payload `empty_pred` still calls bottom, regardless of what the pre-lift inputs looked like.

The lemmas below extend this from a single transfer step to every other value-producing primitive a solver combines lifted values with: $\sqcup$ (joining several contributions at a program point) and $\nabla\Delta$ (the warrowing update rule). Both need only that `empty_pred` itself is downward closed under the payload's order (`mono` below) – exactly `is_empty_antimono`'s shape at `numeric_domain`, or `default_st_is_bot_for`'s own monotonicity once bridged through `default_st_to_fun_mono` – not any fact specific to how `empty_pred` itself is computed.

```
definition normalized_lift :: "('a ⇒ bool) ⇒ 'a lifted ⇒ bool" where
  "normalized_lift empty_pred x = (case x of Bot ⇒ True | Lifted a ⇒ ¬ empty_pred a)"

lemma normalized_lift_Bot [simp]: "normalized_lift empty_pred Bot"
  unfolding normalized_lift_def by simp

lemma normalized_lift_Lifted [simp]:
  "normalized_lift empty_pred (Lifted a) ⟷ ¬ empty_pred a"
  unfolding normalized_lift_def by simp

lemma normalize_lift_normalized [simp]:
  "normalized_lift empty_pred (normalize_lift empty_pred a)"
  unfolding normalize_lift_def by (cases "empty_pred a") simp_all

lemma transfer_lift_normalized [simp]:
  "normalized_lift empty_pred (transfer_lift empty_pred f x)"
  by (cases x) (simp_all add: transfer_lift_def)

lemma transfer_lift2_normalized [simp]:
  "normalized_lift empty_pred (transfer_lift2 empty_pred f x y)"
  by (cases x; cases y) (simp_all add: transfer_lift2_def)
```

`canonicalize_lift` re-normalizes an already-lifted value against `empty_pred`: a `Lifted` payload that has since become witness-bottom (e.g. read from a representation that does not itself carry the `transfer_lift` discipline) collapses to `Bot`, exactly as if it had been produced by a transfer step in the first place. It is `transfer_lift empty_pred id`, not a new bottom test: reusing `transfer_lift` at the identity function is what makes `canonicalize_lift_normalized` immediate from `transfer_lift_normalized` rather than a second proof of the same fact.

```
definition canonicalize_lift :: "('a ⇒ bool) ⇒ 'a lifted ⇒ 'a lifted" where
  "canonicalize_lift empty_pred = transfer_lift empty_pred id"
```

```
lemma canonicalize_lift_Bot [simp]: "canonicalize_lift empty_pred Bot = Bot"
  unfolding canonicalize_lift_def by simp

lemma canonicalize_lift_Lifted [simp]:
  "canonicalize_lift empty_pred (Lifted a) = normalize_lift empty_pred a"
  unfolding canonicalize_lift_def by simp

lemma canonicalize_lift_normalized [simp]:
  "normalized_lift empty_pred (canonicalize_lift empty_pred x)"
  unfolding canonicalize_lift_def by (rule transfer_lift_normalized)
```

Canonicalization is idempotent: a value already collapsed against `empty_pred` is left unchanged by collapsing it again – the defining property of a normalization operation.

```
lemma canonicalize_lift_idem [simp]:
  "canonicalize_lift empty_pred (canonicalize_lift empty_pred x)
     = canonicalize_lift empty_pred x"
  by (cases x) (auto simp: normalize_lift_def)
```

$\sqcup$ preserves `normalized_lift` from two already-normalized operands: `Bot` is $\sqcup$'s identity on `'a lifted` (`sup_lifted.simps`), so the only case needing `mono` at all is `Lifted`/`Lifted`, where `sup_ge1`/`sup_ge2` place each operand below the join and `mono`'s contrapositive carries non-bottomness upward.

```
lemma normalized_lift_sup:
  fixes x y :: "'a::semilattice_sup lifted"
  assumes mono: "⋀a b::'a. a ≤ b ⟹ empty_pred b ⟹ empty_pred a"
    and nx: "normalized_lift empty_pred x"
    and ny: "normalized_lift empty_pred y"
  shows "normalized_lift empty_pred (x ⊔ y)"
  using nx ny by (cases x; cases y) (auto dest: mono[OF sup_ge1] mono[OF sup_ge2])
```

`normalized_lift_bind` lifts a per-payload `normalized_lift` guarantee on `f` through `bind_lift`: `Bot` is already normalized, and `Lifted a` hands off to `f a`, which the assumption covers uniformly. Left unattributed: its premise is a schematic function fact, too broad a shape for automatic intro search.

```
lemma normalized_lift_bind:
  assumes "⋀a. normalized_lift empty_pred (f a)"
  shows "normalized_lift empty_pred (bind x f)"
  using assms by (cases x) simp_all
```

Generic monotonicity for the reachability dispatch, proved once here instead of at every instantiation (`abs_state`, `default_st`, ...): whenever the underlying transfer is monotone and the bottom predicate is downward closed (as `is_empty_state` always is, via `is_empty_state_antimono`), `transfer_lift`/`transfer_lift2` are monotone.

```
lemma normalize_lift_mono:
  fixes a b :: "'a::semilattice_sup"
  assumes f_le: "a ≤ b"
    and bot_mono: "empty_pred b ⟹ empty_pred a"
  shows "normalize_lift empty_pred a ≤ normalize_lift empty_pred b"
  unfolding normalize_lift_def using f_le bot_mono by auto

lemma transfer_lift_mono:
  fixes x1 x2 :: "'a::semilattice_sup lifted"
  assumes f_mono: "⋀a b. a ≤ b ⟹ f a ≤ f b"
    and bot_mono: "⋀a b. a ≤ b ⟹ empty_pred b ⟹ empty_pred a"
    and le: "x1 ≤ x2"
  shows "transfer_lift empty_pred f x1
         ≤ (transfer_lift empty_pred f x2 :: 'b::semilattice_sup lifted)"
  using assms by (cases x1; cases x2) (auto intro!:normalize_lift_mono)

lemma transfer_lift2_mono:
  fixes x1 x2 :: "'a::semilattice_sup lifted" and y1 y2 :: "'b::semilattice_sup lifted"
  assumes f_mono: "⋀a1 a2 b1 b2. a1 ≤ a2 ⟹ b1 ≤ b2 ⟹ f a1 b1 ≤ f a2 b2"
    and bot_mono: "⋀a b. a ≤ b ⟹ empty_pred b ⟹ empty_pred a"
    and lex: "x1 ≤ x2" and ley: "y1 ≤ y2"
  shows "transfer_lift2 empty_pred f x1 y1
         ≤ (transfer_lift2 empty_pred f x2 y2 :: 'c::semilattice_sup lifted)"
  using assms by (cases x1; cases x2; cases y1; cases y2)
     (auto intro: normalize_lift_mono f_mono bot_mono)
```

`bind_lift`'s own monotonicity: a monotone step function carried through a monotone bind argument stays monotone. Left unattributed, like `normalized_lift_bind`: its `f`/`g` premise is schematic and broader than a typical `[intro]` shape, so call sites invoke it explicitly.

```
lemma bind_lift_mono2:
  fixes x y :: "'a::semilattice_sup lifted"
```

```
  assumes xy: "x ≤ y"
      and fg: "⋀a b. a ≤ b ⟹ f a ≤ (g b :: 'b::semilattice_sup lifted)"
  shows "bind x f ≤ bind y g"
  using xy by (cases x; cases y) (auto intro: fg)
```

Solver update integration

The solver's update rules need `'d::{bounded_semilattice_sup_bot, warrowing}`. `'a lifted` already carries `widening` and `narrowing` separately; this registers their combination `warrowing`, which Isabelle does not derive from the two arities by itself.

```
instance lifted :: ("{bounded_semilattice_sup_bot, warrowing}") warrowing ..
```

Transporting a lifted transfer along a representation map

A representation map that commutes with the underlying operation, and classifies emptiness the same way on both sides, commutes with the whole lifted transfer. Stated for `transfer_lift`/`transfer_lift2` themselves, so any pair of representations of one domain — an executable carrier and the mathematical one it stands for, say — gets the fact once rather than once per operation that uses it.

```
lemma transfer_lift_commute:
  assumes commute: "⋀s. phi (f s) = F (phi s)"
    and exact: "⋀s. empty_pred s = empty_pred' (phi s)"
  shows "map_lift phi (transfer_lift empty_pred f d) = transfer_lift empty_pred' F (map_lift phi d)"
  by (cases d) (simp_all add: transfer_lift_def normalize_lift_def commute exact)

lemma transfer_lift2_commute:
  assumes commute: "⋀s t. phi (f s t) = F (phi s) (phi t)"
    and exact: "⋀s. empty_pred s = empty_pred' (phi s)"
  shows "map_lift phi (transfer_lift2 empty_pred f d1 d2) =
          transfer_lift2 empty_pred' F (map_lift phi d1) (map_lift phi d2)"
  by (cases d1; cases d2) (simp_all add: transfer_lift2_def normalize_lift_def commute exact)

end
```

2.3 Three_Valued

```
theory Three_Valued
  imports Main
begin
```

Three-valued Boolean combinators

`bin_log f ann` lifts a Boolean connective `f` to `bool option` with the short-circuit reading `f` already has: one definite operand equal to the annihilator `ann` decides the result regardless of the other side, two definite operands are combined by `f`, and anything else is `None`. Goblint's `id_binary_log` (`base.ml`) has the same shape; `and_opt` and `or_opt` are its two instances.

Both expression evaluation and check classification combine three-valued answers this way, so the combinators sit below either of them.

```
definition bin_log ::
  "(bool ⇒ bool ⇒ bool) ⇒ bool ⇒ bool option ⇒ bool option ⇒ bool option" where
  "bin_log f ann x y =
     (if x = Some ann ∨ y = Some ann then Some ann
      else case (x, y) of (Some a, Some b) ⇒ Some (f a b) | _ ⇒ None)"

abbreviation and_opt :: "bool option ⇒ bool option ⇒ bool option" where
  "and_opt ≡ bin_log (∧) False"

abbreviation or_opt :: "bool option ⇒ bool option ⇒ bool option" where
  "or_opt ≡ bin_log (∨) True"
```

The semantic reading a caller actually wants: given what `x`/`y` mean (`px`/`py`, via the same Horn-clause shape an induction hypothesis already has), the combined answer agrees with `f` on those meanings, provided `ann` really annihilates `f`. Stated this way, a consumer never has to know `bin_log`'s own case split.

```
lemma bin_log_sound:
  assumes "bin_log f ann x y = Some r"
    and "⋀b. f ann b = ann" and "⋀b. f b ann = ann"
    and "⋀b. x = Some b ⟹ px = b"
    and "⋀b. y = Some b ⟹ py = b"
  shows "f px py = r"
  using assms unfolding bin_log_def
  by (cases x; cases y) (auto split: if_splits)

lemma and_opt_sound:
  assumes "and_opt x y = Some r"
    and "⋀b. x = Some b ⟹ px = b"
    and "⋀b. y = Some b ⟹ py = b"
  shows "(px ∧ py) = r"
  using bin_log_sound[OF assms(1) _ _ assms(2,3)] by simp

lemma or_opt_sound:
  assumes "or_opt x y = Some r"
    and "⋀b. x = Some b ⟹ px = b"
    and "⋀b. y = Some b ⟹ py = b"
  shows "(px ∨ py) = r"
  using bin_log_sound[OF assms(1) _ _ assms(2,3)] by simp
```

A definite answer survives widening the operands, so the combinators keep every definite answer of their wider inputs.

```
lemma bin_log_mono:
  assumes "⋀b. x2 = Some b ⟹ x1 = Some b" and "⋀b. y2 = Some b ⟹ y1 = Some b"
  shows "bin_log f ann x2 y2 = Some b ⟹ bin_log f ann x1 y1 = Some b"
  using assms unfolding bin_log_def
  by (cases x2; cases y2) (auto split: if_splits)

lemma and_opt_mono:
  assumes "⋀b. x2 = Some b ⟹ x1 = Some b" and "⋀b. y2 = Some b ⟹ y1 = Some b"
  shows "and_opt x2 y2 = Some b ⟹ and_opt x1 y1 = Some b"
  using bin_log_mono[OF assms] .

lemma or_opt_mono:
  assumes "⋀b. x2 = Some b ⟹ x1 = Some b" and "⋀b. y2 = Some b ⟹ y1 = Some b"
  shows "or_opt x2 y2 = Some b ⟹ or_opt x1 y1 = Some b"
  using bin_log_mono[OF assms] .

end
```

2.4 Query_Lift

```
theory Query_Lift
  imports Main
begin

unbundle lattice_syntax
```

Lifting a domain with a fresh bottom and top

Goblint answers `EvalInt` in `Lattice.Lift(IntDomTuple)` (`valueDomainQueries.ml` at `0dc12d355`): the integer domain with a fresh $\bot$ and $\top$ around it. The fresh top lets an analysis that does not understand a query decline it without knowing the value domain's own top, and the fresh bottom marks a query asked of an unreachable state. This is kept apart from the reachability lift of the solver's carrier: the two bottoms mean different things, and only this one has a fresh top.

```
datatype 'a query_lift = QBot | QLifted 'a | QTop

instantiation query_lift :: (ord) ord
begin

fun less_eq_query_lift :: "'a query_lift ⇒ 'a query_lift ⇒ bool" where
  "less_eq_query_lift QBot _ = True"
| "less_eq_query_lift _ QTop = True"
| "less_eq_query_lift (QLifted a) (QLifted b) = (a ≤ b)"
| "less_eq_query_lift _ _ = False"

definition less_query_lift :: "'a query_lift ⇒ 'a query_lift ⇒ bool" where
  "less_query_lift a b ⟷ a ≤ b ∧ ¬ b ≤ a"

instance ..

end

instance query_lift :: (order) order
proof
  fix x y z :: "'a query_lift"
  show "x < y ⟷ x ≤ y ∧ ¬ y ≤ x"
    by (simp add: less_query_lift_def)
  show "x ≤ x"
    by (cases x) simp_all
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
    by (cases x; cases y; cases z) auto
  show "x ≤ y ⟹ y ≤ x ⟹ x = y"
    by (cases x; cases y) auto
qed

instantiation query_lift :: (order) order_bot
begin
definition bot_query_lift :: "'a query_lift" where "bot_query_lift = QBot"
instance by standard (simp add: bot_query_lift_def)
end

instantiation query_lift :: (order) order_top
begin
definition top_query_lift :: "'a query_lift" where "top_query_lift = QTop"
instance
proof
  fix x :: "'a query_lift"
  show "x ≤ ⊤"
    by (cases x) (simp_all add: top_query_lift_def)
qed
end
```

`QTop` is the identity of the meet and `QBot` absorbs it; two lifted values meet in the underlying domain.

```
instantiation query_lift :: (inf) inf
begin

fun inf_query_lift :: "'a query_lift ⇒ 'a query_lift ⇒ 'a query_lift" where
  "inf_query_lift QBot _ = QBot"
| "inf_query_lift _ QBot = QBot"
| "inf_query_lift QTop b = b"
| "inf_query_lift a QTop = a"
| "inf_query_lift (QLifted a) (QLifted b) = QLifted (a ⊓ b)"
```

```
instance ..

end

instance query_lift :: (semilattice_inf) semilattice_inf
proof
  fix x y z :: "'a query_lift"
  show "x ⊓ y ≤ x"
    by (cases x; cases y) simp_all
  show "x ⊓ y ≤ y"
    by (cases x; cases y) simp_all
  show "x ≤ y ⟹ x ≤ z ⟹ x ≤ y ⊓ z"
    by (cases x; cases y; cases z) simp_all
qed
```

Concretization

A lifted value denotes what the underlying domain says; the fresh bottom denotes nothing and the fresh top everything. An exact underlying meet stays exact after lifting.

```
fun gamma_query_lift :: "('a ⇒ 'b set) ⇒ 'a query_lift ⇒ 'b set" where
  "gamma_query_lift γ QBot = {}"
| "gamma_query_lift γ (QLifted a) = γ a"
| "gamma_query_lift γ QTop = UNIV"

lemma gamma_query_lift_top [simp]: "gamma_query_lift γ ⊤ = UNIV"
  by (simp add: top_query_lift_def)

lemma gamma_query_lift_bot [simp]: "gamma_query_lift γ ⊥ = {}"
  by (simp add: bot_query_lift_def)

lemma gamma_query_lift_inf:
  assumes "⋀a b. γ (a ⊓ b) = γ a ∩ γ b"
  shows "gamma_query_lift γ (x ⊓ y) = gamma_query_lift γ x ∩ gamma_query_lift γ y"
  using assms by (cases x; cases y) simp_all

end
```

2.5 State_Concretization

```
theory State_Concretization
  imports "Voblint_VIMP.VIMP_Syntax" Abstract_Domain
begin
```

The stores an abstract state describes

Every kind of abstract state, pointwise or relational, denotes a set of stores. γ `d` is that set for every state type: each state theory registers its own concretization under this one constant with `adhoc_overloading`, and the type of `d` selects it.

The class parameter γ gives up its own notation here, so that γ has exactly one production in every theory that sees this one.

```
no_notation gamma ("γ")

consts gamma_S :: "'s ⇒ 'v set" ("γ")

adhoc_overloading gamma_S == gamma
```

An executable state is read back as the abstract state it represents, given the classifier that tells locals from globals. `readback` $\mathcal{G}$ `x` is that reading for every executable representation, registered the same way, so a concretization of an executable state is γ `(readback` $\mathcal{G}$ `x)` whatever its representation. Its notation $\varrho_{\mathcal{G}}$ `x` is opt-in, in the bundle `default_st_syntax`.

```
consts readback :: "(vname ⇒ bool) ⇒ 'e ⇒ 'a"

end
```

2.6 Nonrelational_State

```
theory Nonrelational_State
  imports State_Concretization
begin
```

What a store of one abstract value per variable denotes, and when it denotes nothing

A non-relational abstract state, `'a abs_state`, pairs every program variable with one abstract value, independently of every other variable – the "non-relational" in the name. `gamma_state` says what that pairing denotes: exactly the concrete stores where each variable's value lies in its own component's concretization, i.e. the product of the per-component concretizations. Because that product is a conjunction over every variable, one witness-bottom component – one variable whose abstract value denotes no concrete value at all – already makes the whole state denote nothing, regardless of what every other component holds. `is_empty_state` names that witness-bottom condition directly, so a generic transfer dispatcher can detect and canonicalize it without testing `gamma_state` against the empty set at run time.

```
type_synonym 'a abs_state = "vname => 'a"


instance "fun" :: (type, bounded_semilattice_sup_bot) bounded_semilattice_sup_bot ..
```

Pointwise join on abstract states is idempotent because the value-domain semilattice structure lifts pointwise. Finite folds can therefore use the standard idempotent-join laws without a separate state-level assumption.

State concretization

```
definition gamma_state :: "('a::numeric_domain) abs_state ⇒ store set" where
  "gamma_state d = {s. ∀x. s x ∈ γ (d x)}"

adhoc_overloading gamma_S == gamma_state

lemma gamma_stateI [intro]:
  "(⋀x. s x ∈ γ (d x)) ⟹ s ∈ γ d"
  unfolding gamma_state_def by simp
```

State concretization laws

Every value-level concretization law, lifted to states. All of them fall out pointwise, which is exactly what makes a nonrelational state cheap: order, bottom and join are read one variable at a time, and no law here needs a relation between variables.

```
lemma gamma_state_mono:
  "sigma1 ≤ sigma2 ⟹ γ sigma1 ⊆ γ sigma2"
  for sigma1 sigma2 :: "'a::numeric_domain abs_state"
  unfolding gamma_state_def le_fun_def
  using gamma_mono by blast

lemma gamma_state_bot [simp]:
  "γ (bot :: 'a::numeric_domain abs_state) = {}"
  unfolding gamma_state_def bot_fun_def using gamma_bot by auto

lemma gamma_state_sup_ub1 [intro]:
  "γ sigma1 ⊆ γ (sigma1 ⊔ sigma2)"
  for sigma1 sigma2 :: "'a::numeric_domain abs_state"
  unfolding gamma_state_def sup_fun_def
  using gamma_sup_ub1 by blast

lemma gamma_state_sup_ub2 [intro]:
  "γ sigma2 ⊆ γ (sigma1 ⊔ sigma2)"
  for sigma1 sigma2 :: "'a::numeric_domain abs_state"
  unfolding gamma_state_def sup_fun_def
  using gamma_sup_ub2 by blast
```

Membership counterparts of `gamma_state_sup_ub1`/`gamma_state_sup_ub2`, for citing at a witness `s` directly (`using subsetD[OF gamma_state_sup_ub1 h]` otherwise repeats the same `subsetD` wrapper at every call site). `[intro]`: their conclusion has the distinctive `s ∈ γ (sigma1 ⊔ sigma2)` shape a join-injection goal actually has, and a witness already in one side's concretization is exactly the natural way to discharge it.

```
lemma gamma_state_supI1 [intro]:
  "s ∈ γ sigma1 ⟹ s ∈ γ (sigma1 ⊔ sigma2)"
```

```
  for sigma1 sigma2 :: "'a::numeric_domain abs_state"
  using gamma_state_sup_ub1 by blast

lemma gamma_state_supI2 [intro]:
  "s ∈ γ sigma2 ⟹ s ∈ γ (sigma1 ⊔ sigma2)"
  for sigma1 sigma2 :: "'a::numeric_domain abs_state"
  using gamma_state_sup_ub2 by blast
```

Left bare rather than `[dest]`: `x` is unconstrained by the premise, so a global destruction rule here would let `auto` pick a schematic variable unrelated to the goal at hand (as with the `Voblint_Solver` session's `env_indep_depsD`). Cite it explicitly with the variable the goal needs.

```
lemma gamma_stateD:
  "s ∈ γ d ⟹ s x ∈ γ (d x)"
  for d :: "'a::numeric_domain abs_state"
  unfolding gamma_state_def by simp
```

Witness-bottom abstract states

Witness-bottom is not "the state has exactly one empty component": any number of a state's components can independently denote the empty set of concrete integers at once, and `is_empty_state` does not distinguish those cases from each other or count them. It only asks whether at least one witness exists (`∃x. is_empty (d x)`), because one witness already suffices – `gamma_state`'s product structure means a single empty component collapses the whole state's concretization to the empty set of stores, so a second, third, or every remaining empty component would only reconfirm what the first already decided.

`is_empty_state` is a logical characterization of that condition, not an executable test: `'a abs_state` is a raw function `vname ⇒ 'a`, and `vname` is infinite, so `∃x. is_empty (d x)` has no code equation – there is no finite witness search for code generation to compile. The finite executable bottom test a real dispatcher runs is `default_st_is_bot_for`, defined downstream in the `Voblint_Exec` session, which scans only the state's finitely many stored locations and is proved to agree with `is_empty_state` on every such state.

```
definition is_empty_state :: "('a::executable_domain) abs_state ⇒ bool" where
  "is_empty_state d = (∃x. is_empty (d x))"

lemma is_empty_stateI [intro]:
  "is_empty (d x) ⟹ is_empty_state d"
  unfolding is_empty_state_def by (rule exI)

lemma is_empty_stateE [elim]:
  assumes "is_empty_state d"
  obtains x where "is_empty (d x)"
  using assms unfolding is_empty_state_def by blast
```

Overwriting one variable of a state that is not already empty can only make it empty through the written element. This is what lets an incremental emptiness check inspect the freshly computed element alone instead of searching the infinite variable space again.

```
lemma is_empty_state_fun_upd_iff:
  assumes "¬ is_empty_state d"
  shows "is_empty_state (d(x := a)) ⟷ is_empty a"
proof
  assume "is_empty_state (d(x := a))"
  then obtain y where "is_empty ((d(x := a)) y)"
    by (rule is_empty_stateE)
  with assms show "is_empty a"
    by (cases "y = x") auto
next
  assume "is_empty a"
  then have "is_empty ((d(x := a)) x)" by simp
  then show "is_empty_state (d(x := a))"
    by (rule is_empty_stateI)
qed

lemma is_empty_state_gamma_state_empty:
  assumes "is_empty_state d"
  shows "γ d = {}"
proof -
  obtain x where "is_empty (d x)" using assms by (rule is_empty_stateE)
  then show ?thesis
    using gamma_stateD[of _ d x] unfolding is_empty_correct by fastforce
qed

lemma gamma_state_empty_is_empty_state:
```

```
  assumes "γ d = {}"
  shows "is_empty_state d"
proof (rule ccontr)
  assume "¬ is_empty_state d"
  then have nonempty: "⋀x. ∃v. v ∈ γ (d x)"
    unfolding is_empty_state_def using is_empty_correct by blast
  define s where "s = (λx. SOME v. v ∈ γ (d x))"
  have s_prop: "s x ∈ γ (d x)" for x
    unfolding s_def using nonempty[of x] by (rule someI_ex)
  then have "s ∈ γ d"
    unfolding gamma_state_def by simp
  with assms show False by simp
qed

lemma is_empty_state_iff_gamma_state_empty:
  "is_empty_state d ⟷ γ d = {}"
  using is_empty_state_gamma_state_empty gamma_state_empty_is_empty_state by blast

lemma exact_emptiness_is_empty_state:
  "exact_emptiness is_empty_state (gamma_state :: 'a::numeric_domain abs_state ⇒ store set)"
  by (rule exact_emptinessI) (rule is_empty_state_iff_gamma_state_empty)

lemma is_empty_state_bot [simp]:
  "is_empty_state (bot :: 'a::numeric_domain abs_state)"
  unfolding is_empty_state_def bot_fun_def
  using is_empty_correct gamma_bot by blast

lemma is_empty_state_antimono:
  "d1 ≤ d2 ⟹ is_empty_state d2 ⟹ is_empty_state d1"
  for d1 d2 :: "'a::numeric_domain abs_state"
  unfolding is_empty_state_def le_fun_def using is_empty_antimono by blast

end
```

2.7 Nonrelational_Reachability

```
theory Nonrelational_Reachability
  imports Nonrelational_State Reachability_Lift
begin
```

Reachability for pointwise abstract stores

Pointwise stores and structural reachability are independent domain constructions. This theory composes them: unreachable points denote no concrete store, and reachable payloads use γ. The logical witness-empty predicate agrees with that composed concretization; a concrete executable state representation supplies a finite implementation of it (`default_st_is_bot_for`, downstream in the `Voblint_Exec` session), since `is_empty_state_lift` below inherits `is_empty_state`'s non-executable quantifier over `vname` on its `Lifted` case.

Composed concretization

A lifted pointwise state is one more kind of state, so γ `d` covers it too. $\gamma_\bot$ `_` stays as input syntax that fixes the lifted type where nothing else does.

```
abbreviation (input) gamma_state_lift ::
  "'a::numeric_domain abs_state lifted ⇒ store set" ("γ⊥") where
  "gamma_state_lift ≡ gamma_lift gamma_state"

adhoc_overloading gamma_S == "gamma_lift gamma_state"

fun is_empty_state_lift ::
  "'a::executable_domain abs_state lifted ⇒ bool" where
  "is_empty_state_lift Bot = True"
| "is_empty_state_lift (Lifted d) = is_empty_state d"

lemma is_empty_state_lift_iff:
  "is_empty_state_lift s ⟷ γ⊥ s = {}"
  by (cases s) (simp_all add: is_empty_state_iff_gamma_state_empty)

lemma exact_emptiness_is_empty_state_lift:
  "exact_emptiness is_empty_state_lift
     (gamma_lift gamma_state :: 'a::numeric_domain abs_state lifted ⇒ store set)"
  by (rule exact_emptinessI) (rule is_empty_state_lift_iff)
```

`Reachability_Lift`'s generic `gamma_normalize_lift`/`normalize_lift_mono`/ `normalized_lift_sup` specialized to `is_empty_state`: each generic side condition (concretization agreement, downward closure) is discharged by `is_empty_state_iff_gamma_state_empty`/`is_empty_state_antimono`. The first is tagged `[simp]`, unlike the generic version, because its side condition is now a closed fact rather than a further proof obligation.

```
lemma gamma_state_normalize_lift [simp]:
  "γ⊥ (normalize_lift is_empty_state d) = γ d"
  by (rule gamma_normalize_lift) (rule is_empty_state_iff_gamma_state_empty)
```

Collapsing a witness-bottom payload changes what a value says about itself, never what it represents. Every public result adapter routes its solved values through `canonicalize_lift` before publishing them, so this is what lets such an adapter's soundness argument ignore the collapse entirely.

```
lemma gamma_state_canonicalize_lift [simp]:
  "γ⊥ (canonicalize_lift is_empty_state d) = γ⊥ d"
  by (cases d) simp_all

lemma normalize_state_lift_mono [intro]:
  fixes d1 d2 :: "'a::numeric_domain abs_state"
  assumes le: "d1 ≤ d2"
  shows "normalize_lift is_empty_state d1 ≤ normalize_lift is_empty_state d2"
  by (rule normalize_lift_mono[OF le]) (rule is_empty_state_antimono[OF le])

lemma normalized_state_lift_sup [intro]:
  fixes s1 s2 :: "'a::numeric_domain abs_state lifted"
  assumes n1: "normalized_lift is_empty_state s1" and n2: "normalized_lift is_empty_state s2"
  shows "normalized_lift is_empty_state (s1 ⊔ s2)"
  by (rule normalized_lift_sup[OF is_empty_state_antimono n1 n2])
```

The normalization invariant

After normalization, structural and semantic deadness coincide: a `normalized_lift`-disciplined value can no longer disagree with its own `is_empty_state_lift` reading, because normalization is exactly what rules out a live-looking `Lifted` payload that is secretly witness-bottom.

```
lemma normalized_state_lift_bot_iff [simp]:
  "normalized_lift is_empty_state s ⟹ is_empty_state_lift s ⟷ s = Bot"
  by (cases s) simp_all
```

Join

`gamma_state_supI1`/`gamma_state_supI2` lifted through `gamma_state_lift`: a witness on one side of a lifted join survives it, whether the other side is `Bot` (the join's identity) or a competing `Lifted` payload.

```
lemma gamma_state_lift_supI1 [intro]:
  "s ∈ γ⊥ x ⟹ s ∈ γ⊥ (x ⊔ y)"
  for x y :: "'a::numeric_domain abs_state lifted"
  by (cases x; cases y) auto

lemma gamma_state_lift_supI2 [intro]:
  "s ∈ γ⊥ y ⟹ s ∈ γ⊥ (x ⊔ y)"
  for x y :: "'a::numeric_domain abs_state lifted"
  by (cases x; cases y) auto
```

Carrying a transfer's soundness through the lift

A pure transfer's soundness fact carries over the `transfer_lift` wrapper by one case split: `Bot`'s concretization is empty, and a `normalize_lift` collapse to `Bot` only ever fires when `empty_pred` holds, whose concretization is empty by assumption. Three shapes cover every use — a collecting inclusion, a membership, and a binary membership for a combine — so no consumer repeats the bottom argument.

```
lemma transfer_lift_sound_collect:
  assumes step: "⋀d'. C (γ d') ⊆ γ (f d')"
    and Cempty: "C {} = {}"
    and empty_pred_sound: "⋀d'. empty_pred d' ⟹ γ d' = {}"
  shows "C (γ⊥ d) ⊆ γ⊥ (transfer_lift empty_pred f d)"
proof (cases d)
  case Bot
  then show ?thesis by (simp add: Cempty)
next
  case (Lifted d')
  have "C (γ d') ⊆ γ (f d')" by (rule step)
  then show ?thesis
    using Lifted empty_pred_sound[of "f d'"] by (auto simp: normalize_lift_def)
qed

lemma transfer_lift_sound_mem:
  assumes step: "⋀d'. s ∈ γ d' ⟹ h s ∈ γ (f d')"
    and empty_pred_sound: "⋀d'. empty_pred d' ⟹ γ d' = {}"
    and s: "s ∈ γ⊥ d"
  shows "h s ∈ γ⊥ (transfer_lift empty_pred f d)"
proof (cases d)
  case Bot
  then show ?thesis using s by simp
next
  case (Lifted d')
  with s have "s ∈ γ d'" by simp
  then have hs: "h s ∈ γ (f d')" by (rule step)
  then have "¬ empty_pred (f d')" using empty_pred_sound by auto
  with Lifted hs show ?thesis by simp
qed

lemma transfer_lift2_sound_mem:
  assumes step: "⋀d1' d2'. s ∈ γ d1' ⟹ t ∈ γ d2' ⟹ h s t ∈ γ (f d1' d2')"
    and empty_pred_sound: "⋀d'. empty_pred d' ⟹ γ d' = {}"
    and s: "s ∈ γ⊥ d1"
    and t: "t ∈ γ⊥ d2"
  shows "h s t ∈ γ⊥ (transfer_lift2 empty_pred f d1 d2)"
proof (cases d1)
  case Bot
  then show ?thesis using s by simp
next
  case (Lifted d1')
  show ?thesis
  proof (cases d2)
    case Bot
    then show ?thesis using t by simp
  next
    case (Lifted d2')
    with ‹d1 = Lifted d1'› s t have "s ∈ γ d1'" "t ∈ γ d2'" by simp_all
```

```
    then have hst: "h s t ∈ γ (f d1' d2')" by (rule step)
    then have "¬ empty_pred (f d1' d2')" using empty_pred_sound by auto
    with ‹d1 = Lifted d1'› Lifted hst show ?thesis by simp
  qed
qed

end
```

2.8 Forward_Domain

```
theory Forward_Domain
  imports State_Concretization "Voblint_VIMP.VIMP_Expr"
begin
```

Forward evaluation over a numeric domain

Every expression-level interface of a domain needs the same two operations: an evaluator of expressions over an abstract state and a truth test on abstract values. Guard refinement (`sound_refinement`), the special calls of the transfer functions (`sound_minmax_ops`), the arithmetic of an expression domain (`sound_arith_ops`) and the check layer (`sound_check_query`) all build on them. Stating the operations and their laws once lets a domain prove them once: the first interpretation registers them, and every later interface over the same operations inherits them instead of asking again.

The evaluator is stated over any state type `'d` together with the stores γ_S `d` a state describes. The pointwise abstract states of the numeric domains are the instance at their pointwise `gamma_state`; the check layer keeps the state type open.

```
locale sound_evaluator =
  fixes γ_S :: "'d ⇒ store set"
    and aval_abs :: "exp ⇒ 'd ⇒ 'a::numeric_domain" ("⟦_⟧♯")
  assumes aval_abs_sound[intro]:
    "s ∈ γ_S d ⟹ ⟦e⟧_e s ∈ γ (⟦e⟧♯ d)"
```

A truth test answers `Some b` only when every integer the abstract value concretizes to has truthiness `b`; `None` means the value cannot decide.

```
locale sound_truth_test =
  fixes tobool :: "'a::numeric_domain ⇒ bool option"
  assumes tobool_sound:
    "tobool p = Some b ⟹ i ∈ γ p ⟹ truthy i = b"
```

Monotonicity is not needed for soundness. The solver's least-solution theorem and the monotone transfer interfaces ask for it, so it is a separate layer. `tobool_mono` reads downward: a definite answer at a coarser value survives at a sharper non-empty one.

```
locale mono_evaluator = sound_evaluator γ_S aval_abs
  for γ_S :: "'d::order ⇒ store set"
    and aval_abs :: "exp ⇒ 'd ⇒ 'a::numeric_domain" ("⟦_⟧♯") +
  assumes aval_abs_mono[intro]:
    "d1 ≤ d2 ⟹ ⟦e⟧♯ d1 ≤ ⟦e⟧♯ d2"
```

`mono_truth_test` adds `tobool_mono` on top of the sound truth test, the downward monotonicity described above for non-empty values.

```
locale mono_truth_test = sound_truth_test +
  assumes tobool_mono:
    "¬ is_empty p1 ⟹ p1 ≤ p2 ⟹ tobool p2 = Some bv ⟹ tobool p1 = Some bv"

end
```

2.9 Backward_Domain

```
theory Backward_Domain
  imports Nonrelational_Reachability Forward_Domain
begin
```

Refining an abstract state against a guard

A forward transfer over-approximates what an assignment does; a backward one narrows a state against a condition that is known to hold, which is what a branch on `b` learns about the stores that pass it. `sound_refinement` is the interface a domain supplies for that: inverse operators for arithmetic and comparison (`inv_less`, `inv_plus`, ...), each sound in the sense that every concrete pair the operator admits before the operation is still admitted after narrowing, from which `afilter` and `bfilter` – the expression- and boolean-level filters – are derived once with their soundness and reductiveness. `mono_refinement` adds the monotone operators a domain can provide on top. This abstracts the backward-refinement operations Goblint's `BaseInvariant` implements concretely for its Base analysis; Goblint has no generic module signature this locale is a formalization of.

Backward-analysis locale

A semantic intersection preserves every concrete value shared by both operands and lies below both. It need not be the lattice infimum: a domain may normalize an empty result while its representation order still distinguishes several empty elements. Soundness of the filters needs only `intersect_sound`. The lower bounds make every refinement step reductive, so a state emptied by one step stays empty through the rest, which is what lets the executable filter stop at the first empty step (`Exec_Backward`). Every domain's intersection has both, so they form one contract; `mono_intersection` adds monotonicity as a separate strengthening.

```
locale sound_intersection =
  fixes intersect :: "'a::numeric_domain => 'a => 'a"
  assumes intersect_sound[intro]:
    "n ∈ γ a ⟹ n ∈ γ b ⟹ n ∈ γ (intersect a b)"
    and intersect_reductive1[intro]: "intersect a b ≤ a"
    and intersect_reductive2[intro]: "intersect a b ≤ b"
```

Monotone intersection is a separate layer for the same reason as in the forward interfaces: soundness needs only `sound_intersection`, while the solver's least-solution theorem asks `intersect` to preserve the order in both arguments.

```
locale mono_intersection = sound_intersection +
  assumes intersect_mono[intro]:
    "a1 ≤ a2 ⟹ b1 ≤ b2 ⟹ intersect a1 b1 ≤ intersect a2 b2"
```

The inverse operators work on abstract values alone: given the result an operation must have produced, each narrows its operands to values that still admit every concrete pair producing it. `sound_inverse_ops` states them and their soundness with no state in sight, which is all the derived numeric queries (`Backward_Numeric_Queries`) read.

```
locale sound_inverse_ops = sound_intersection intersect
  for intersect :: "'a::numeric_domain => 'a => 'a" +
  fixes
    inv_less  :: "bool => 'a => 'a => 'a * 'a"
    and inv_eq    :: "bool => 'a => 'a => 'a * 'a"
    and inv_plus  :: "'a => 'a => 'a => 'a * 'a"
    and inv_minus :: "'a => 'a => 'a => 'a * 'a"
    and inv_times :: "'a => 'a => 'a => 'a * 'a"
  assumes
      inv_less_sound:
      "n1 ∈ γ a1 ⟹ n2 ∈ γ a2 ⟹ (n1 < n2) = res
       ⟹ n1 ∈ γ (fst (inv_less res a1 a2)) ∧ n2 ∈ γ (snd (inv_less res a1 a2))"
    and inv_eq_sound:
      "n1 ∈ γ a1 ⟹ n2 ∈ γ a2 ⟹ (n1 = n2) = res
       ⟹ n1 ∈ γ (fst (inv_eq res a1 a2)) ∧ n2 ∈ γ (snd (inv_eq res a1 a2))"
    and inv_plus_sound:
      "n1 ∈ γ a1 ⟹ n2 ∈ γ a2 ⟹ n1 + n2 ∈ γ r
       ⟹ n1 ∈ γ (fst (inv_plus r a1 a2)) ∧ n2 ∈ γ (snd (inv_plus r a1 a2))"
    and inv_minus_sound:
      "n1 ∈ γ a1 ⟹ n2 ∈ γ a2 ⟹ n1 - n2 ∈ γ r
       ⟹ n1 ∈ γ (fst (inv_minus r a1 a2)) ∧ n2 ∈ γ (snd (inv_minus r a1 a2))"
    and inv_times_sound:
      "n1 ∈ γ a1 ⟹ n2 ∈ γ a2 ⟹ n1 * n2 ∈ γ r
       ⟹ n1 ∈ γ (fst (inv_times r a1 a2)) ∧ n2 ∈ γ (snd (inv_times r a1 a2))"
begin
```

Projections of the five `inv_*_sound` assumptions above, one membership fact per operand instead of the pair conjunction the raw assumption gives. These, not the raw assumptions, are the `[intro]` rules: their conclusion has a distinctive `fst (inv_* ...)`/`snd (inv_* ...)` head, so a goal of exactly that shape picks the matching rule directly, whereas the raw conjunction-valued assumption would leave automation to split a conjunction goal it did not ask for.

```
lemmas inv_less_sound_fst [intro] = inv_less_sound[THEN conjunct1]
lemmas inv_less_sound_snd [intro] = inv_less_sound[THEN conjunct2]

lemmas inv_eq_sound_fst [intro] = inv_eq_sound[THEN conjunct1]
lemmas inv_eq_sound_snd [intro] = inv_eq_sound[THEN conjunct2]

lemmas inv_plus_sound_fst [intro] = inv_plus_sound[THEN conjunct1]
lemmas inv_plus_sound_snd [intro] = inv_plus_sound[THEN conjunct2]

lemmas inv_minus_sound_fst [intro] = inv_minus_sound[THEN conjunct1]
lemmas inv_minus_sound_snd [intro] = inv_minus_sound[THEN conjunct2]

lemmas inv_times_sound_fst [intro] = inv_times_sound[THEN conjunct1]
lemmas inv_times_sound_snd [intro] = inv_times_sound[THEN conjunct2]

end
```

`sound_refinement` takes the inverse operators to pointwise states. With the forward operations of `sound_evaluator` and `sound_truth_test` it derives the guard filters `afilter` and `bfilter`, whose soundness follows by induction. A domain that already interpreted the forward locales (through its expression domain) is not asked for their laws again.

```
locale sound_refinement =
  sound_intersection intersect + sound_evaluator gamma_state aval_abs
    + sound_truth_test tobool
    + sound_inverse_ops intersect inv_less inv_eq inv_plus inv_minus inv_times
    for intersect :: "'a::numeric_domain => 'a => 'a"
    and aval_abs :: "exp => 'a abs_state => 'a" ("⟦_⟧♯")
    and tobool :: "'a => bool option"
    and inv_less  :: "bool => 'a => 'a => 'a * 'a"
    and inv_eq    :: "bool => 'a => 'a => 'a * 'a"
    and inv_plus  :: "'a => 'a => 'a => 'a * 'a"
    and inv_minus :: "'a => 'a => 'a => 'a * 'a"
    and inv_times :: "'a => 'a => 'a => 'a * 'a"
begin

fun afilter :: "exp => 'a => 'a abs_state => 'a abs_state" where
    "afilter (V x) a d = d(x := intersect a (d x))"
  | "afilter (Plus  e1 e2) a d =
       (let (a1, a2) = inv_plus a (⟦e1⟧♯ d) (⟦e2⟧♯ d)
        in afilter e1 a1 (afilter e2 a2 d))"
  | "afilter (Minus e1 e2) a d =
       (let (a1, a2) = inv_minus a (⟦e1⟧♯ d) (⟦e2⟧♯ d)
        in afilter e1 a1 (afilter e2 a2 d))"
  | "afilter (Times e1 e2) a d =
       (let (a1, a2) = inv_times a (⟦e1⟧♯ d) (⟦e2⟧♯ d)
        in afilter e1 a1 (afilter e2 a2 d))"
  | "afilter _ a d = d"
```

`feasible` is the forward half of Goblint's two-phase branch handling, as a predicate: evaluate `e` forward and ask whether the abstract value that yields leaves the selected polarity possible at all. A bottom value denotes no store, and a definite `tobool` answer disagreeing with `pol` rules the polarity out; every other case keeps it open. Backward narrowing cannot replace this test: a Boolean-valued subexpression in an operand position inverts to a target `afilter` has no rule to push through a comparison node, so the target is dropped and the state survives unrefined even where no state satisfies the condition.

The leading `is_empty` test is not redundant with the `tobool` one. An author's `tobool` is free to answer arbitrarily at its own domain's bottom (every answer is vacuously sound there, since `gamma bot = {}`), so it need not, and generally does not, agree across a bottom/non-bottom pair `d1` $\leq$ `d2` the way `tobool_mono` requires. Answering `is_empty` directly, ahead of `tobool`, sidesteps that disagreement instead of relying on it, and is what makes `feasible_mono` hold.

```
definition feasible :: "exp => bool => 'a abs_state => bool" where
  "feasible e pol d =
     (¬ is_empty (⟦e⟧♯ d) ∧ tobool (⟦e⟧♯ d) ≠ Some (¬ pol))"
```

Every state a concrete store witnesses is feasible for the polarity that store takes: `aval_abs_sound` rules out the bottom case and `tobool_sound` the disagreeing-answer case. This is what makes the gate's `bot` outcome sound

wherever it fires – it fires only when no represented store exists.

```
lemma feasible_of_concrete [intro]:
  assumes "s ∈ γ d" and "truthy (⟦e⟧ₑ s) = pol"
  shows "feasible e pol d"
  unfolding feasible_def using assms is_empty_correct tobool_sound by blast
```

`bfilter` narrows a state under an assumed truth value of `e`: `bfilter e True` is `assume e`, `bfilter e False` is `assume-not e`. `Not`/`And`/`Or` distribute structurally (De Morgan, over ⊔ for the disjunctive branch); `Less`/`Eq` go straight through `inv_less`/`inv_eq` on their two operands. Every other constructor – `N`, `V`, `Plus`, `Minus`, `Times` – has no Boolean-shaped narrowing operator of its own, so the fallback case reduces truthiness to the one comparison every domain already inverts: `truthy (⟦e⟧ₑ s) = res` iff `(⟦e⟧ₑ s = 0) = (¬ res)`, so `inv_eq (¬ res)` against the abstract constant `0` narrows `e`'s own target value, and `afilter` propagates that target through `e`'s structure. This reuses `inv_eq`/`afilter` rather than adding a new domain-author operator.

The two disjunctive cases – `Or _ _ True` and `And _ _ False` – gate each side on `feasible` before joining, matching Goblint's `inv_exp`, which refines the arms of a disjunction separately and drops one whose refinement contradicts. Without the gate, a side no state satisfies still contributes its own unrefined incoming state and the join discards what the other side established: the empty target such a side inverts to is dropped wherever `afilter` has no rule for the node carrying it, so the narrowing that would have signalled the contradiction never happens. `bot` is the unit of ⊔, so gating an infeasible side removes it from the join exactly.

`feasible` is only a necessary forward gate, though: a side it approves can still have its own `bfilter` recursion discover a stronger, backward-only contradiction, landing on a witness-bottom state (`is_empty_state`) that need not be the literal pointwise `bot` – e.g. empty at the one location the side's own condition constrains, but still carrying whatever `d` already held everywhere else. Joining that raw state *pointwise* then contributes those other, unrefined locations to the join exactly as if the side were live, silently discarding what the other side established there. Canonicalizing each gated side before the join (collapsing a witness-bottom side to the literal `bot`, the unit of ⊔) would close this, but `is_empty_state` has no code equation over an infinite `vname` domain, and embedding it inside `bfilter`'s own primitive-recursive equations would make the whole function – every constructor, not only `And`/`Or` – lose code-generatability with it. `bfilter` therefore stays exactly this pointwise join, and the correction lives one level up, in `bfilter_lifted`, which does not need to code-generate.

```
fun bfilter :: "exp => bool => 'a abs_state => 'a abs_state" where
    "bfilter (Less e1 e2) res d =
       (let (a1, a2) = inv_less res (⟦e1⟧♯ d) (⟦e2⟧♯ d)
        in afilter e1 a1 (afilter e2 a2 d))"
  | "bfilter (GreaterEq e1 e2) res d =
       (let (a1, a2) = inv_less (¬ res) (⟦e1⟧♯ d) (⟦e2⟧♯ d)
        in afilter e1 a1 (afilter e2 a2 d))"
  | "bfilter (Greater e1 e2) res d =
       (let (a1, a2) = inv_less res (⟦e2⟧♯ d) (⟦e1⟧♯ d)
        in afilter e2 a1 (afilter e1 a2 d))"
  | "bfilter (LessEq e1 e2) res d =
       (let (a1, a2) = inv_less (¬ res) (⟦e2⟧♯ d) (⟦e1⟧♯ d)
        in afilter e2 a1 (afilter e1 a2 d))"
  | "bfilter (Not b) res d = bfilter b (¬ res) d"
  | "bfilter (And b1 b2) True  d = bfilter b1 True  (bfilter b2 True  d)"
  | "bfilter (And b1 b2) False d =
       (if feasible b1 False d then bfilter b1 False d else bot)
       ⊔ (if feasible b2 False d then bfilter b2 False d else bot)"
  | "bfilter (Or  b1 b2) True  d =
       (if feasible b1 True d then bfilter b1 True d else bot)
       ⊔ (if feasible b2 True d then bfilter b2 True d else bot)"
  | "bfilter (Or  b1 b2) False d = bfilter b1 False (bfilter b2 False d)"
  | "bfilter (Eq  e1 e2) res  d =
       (let (a1, a2) = inv_eq res (⟦e1⟧♯ d) (⟦e2⟧♯ d)
        in afilter e1 a1 (afilter e2 a2 d))"
  | "bfilter (NotEq  e1 e2) res  d =
       (let (a1, a2) = inv_eq (¬ res) (⟦e1⟧♯ d) (⟦e2⟧♯ d)
        in afilter e1 a1 (afilter e2 a2 d))"
  | "bfilter e res d =
       (let (a1, a2) = inv_eq (¬ res) (⟦e⟧♯ d) (⟦N 0⟧♯ d)
        in afilter e a1 d)"
```

The state-level fact behind `afilter`'s `V` case: narrowing one location by `intersect` keeps every location represented, the narrowed one because `intersect_sound` says so and every other because it is untouched. Named once here so `afilter_sound`'s `V` case cites it instead of unfolding `gamma_state_def` and case-splitting on `y = x` inline.

```
lemma gamma_state_update_intersect [intro]:
  assumes "s ∈ γ d" and "s x ∈ γ a"
  shows "s ∈ γ (d(x := intersect a (d x)))"
  using assms by (simp add: gamma_stateD gamma_stateI intersect_sound)
```

```
lemma afilter_sound [intro]:
  assumes "s ∈ γ d" "⟦e⟧ₑ s ∈ γ a"
  shows "s ∈ γ (afilter e a d)"
using assms proof (induction e arbitrary: a d)
  case (V x)
  then show ?case
    unfolding afilter.simps aval.simps by (rule gamma_state_update_intersect)
next
  case (Plus e1 e2)
  have e1a: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ d)" and e2a: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ d)"
    using aval_abs_sound[OF Plus.prems(1)] by simp_all
  have asum: "⟦e1⟧ₑ s + ⟦e2⟧ₑ s ∈ γ a" using Plus.prems(2) by simp
  show ?case
    unfolding afilter.simps Let_def case_prod_beta
    using e1a e2a asum Plus.prems(1)
    by (blast intro: Plus.IH)
next
  case (Minus e1 e2)
  have e1a: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ d)" and e2a: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ d)"
    using aval_abs_sound[OF Minus.prems(1)] by simp_all
  have adiff: "⟦e1⟧ₑ s - ⟦e2⟧ₑ s ∈ γ a" using Minus.prems(2) by simp
  show ?case
    unfolding afilter.simps Let_def case_prod_beta
    using e1a e2a adiff Minus.prems(1)
    by (blast intro: Minus.IH)
next
  case (Times e1 e2)
  have e1a: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ d)" and e2a: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ d)"
    using aval_abs_sound[OF Times.prems(1)] by simp_all
  have aprod: "⟦e1⟧ₑ s * ⟦e2⟧ₑ s ∈ γ a" using Times.prems(2) by simp
  show ?case
    unfolding afilter.simps Let_def case_prod_beta
    using e1a e2a aprod Times.prems(1)
    by (blast intro: Times.IH)
qed simp_all
```

The two-operand filtering step `Less`/`Eq` each apply once `inv_less`/ `inv_eq` has narrowed their operand pair: filter the second operand by its narrowed target, then the first by its narrowed target against the result. Stated generically here, using the completed `afilter_sound` rather than an induction hypothesis, so `bfilter_sound`'s own `Less`/`Eq` cases apply it directly instead of repeating this chaining.

```
lemma afilter_pair_sound [intro]:
  assumes st: "s ∈ γ d"
      and fst: "⟦e1⟧ₑ s ∈ γ (fst p)"
      and snd: "⟦e2⟧ₑ s ∈ γ (snd p)"
  shows "s ∈ γ (afilter e1 (fst p) (afilter e2 (snd p) d))"
proof -
  have inner: "s ∈ γ (afilter e2 (snd p) d)" by (rule afilter_sound[OF st snd])
  show ?thesis by (rule afilter_sound[OF inner fst])
qed
```

Every constructor without its own Boolean-shaped narrowing operator – `N`, `V`, `Plus`, `Minus`, `Times` – reduces `bfilter`'s target truth value to an `inv_eq` narrowing against the abstract constant `0` (see `bfilter`'s own comment), then propagates the narrowed target through `afilter`. Proved once here, generically in `e`, so the induction below cites it instead of repeating the same `inv_eq`/`afilter_sound` chain per arithmetic constructor.

```
lemma bfilter_default_sound:
  assumes "s ∈ γ d" "truthy (⟦e⟧ₑ s) = res"
  shows "s ∈ γ (afilter e (fst (inv_eq (¬ res) (⟦e⟧♯ d) (⟦N 0⟧♯ d))) d)"
proof -
  have ea: "⟦e⟧ₑ s ∈ γ (⟦e⟧♯ d)"
    using aval_abs_sound[OF assms(1)] by simp
  have e0: "⟦N 0⟧ₑ s ∈ γ (⟦N 0⟧♯ d)"
    by (rule aval_abs_sound[of s d "N 0", OF assms(1)])
  have eq0: "(⟦e⟧ₑ s = ⟦N 0⟧ₑ s) = (¬ res)"
    using assms(2) by auto
  have "⟦e⟧ₑ s ∈ γ (fst (inv_eq (¬ res) (⟦e⟧♯ d) (⟦N 0⟧♯ d)))"
    using inv_eq_sound[OF ea e0 eq0] by simp
  then show ?thesis using afilter_sound[OF assms(1)]
    by simp
qed
```

Soundness of a feasibility-gated join, the shape of `bfilter`'s `And _ _ False` and `Or _ _ True` cases: a store taking the polarity on one side makes that side feasible, so the gate keeps it and the join admits the store through it.

```
lemma gated_join_sound:
  fixes f1 f2 :: "'a abs_state"
  assumes st: "s ∈ γ d"
    and "truthy (⟦b1⟧e s) = pol ∨ truthy (⟦b2⟧e s) = pol"
    and f1: "truthy (⟦b1⟧e s) = pol ⟹ s ∈ γ f1"
    and f2: "truthy (⟦b2⟧e s) = pol ⟹ s ∈ γ f2"
  shows "s ∈ γ ((if feasible b1 pol d then f1 else bot)
               ⊔ (if feasible b2 pol d then f2 else bot))"
  using assms(2)
proof
  assume h: "truthy (⟦b1⟧e s) = pol"
  have "feasible b1 pol d" by (rule feasible_of_concrete[OF st h])
  with f1[OF h] show ?thesis by (simp add: gamma_state_supI1)
next
  assume h: "truthy (⟦b2⟧e s) = pol"
  have "feasible b2 pol d" by (rule feasible_of_concrete[OF st h])
  with f2[OF h] show ?thesis by (simp add: gamma_state_supI2)
qed

lemma bfilter_sound [intro]:
  assumes "s ∈ γ d" "truthy (⟦e⟧e s) = res"
  shows "s ∈ γ (bfilter e res d)"
using assms proof (induction e arbitrary: res d)
  case (Not e)
  have bv': "truthy (⟦e⟧e s) = (¬ res)" using Not.prems(2) by (auto split: if_splits)
  from Not.IH[OF Not.prems(1) bv'] show ?case by simp
next
  case (And e1 e2)
  show ?case
  proof (cases res)
    case True
    have v1: "truthy (⟦e1⟧e s) = True" and v2: "truthy (⟦e2⟧e s) = True"
      using And.prems(2) True unfolding truthy_aval_And by simp_all
    show ?thesis
      using And.IH(1)[OF And.IH(2)[OF And.prems(1) v2] v1] by (simp add: True)
  next
    case False
    then have res: "res = False" by simp
    have "truthy (⟦e1⟧e s) = False ∨ truthy (⟦e2⟧e s) = False"
      using And.prems(2) res by (auto split: if_splits)
    then show ?thesis
      unfolding res bfilter.simps
      by (rule gated_join_sound[OF And.prems(1) _ And.IH[OF And.prems(1)]])
  qed
next
  case (Or e1 e2)
  show ?case
  proof (cases res)
    case True
    then have res: "res = True" by simp
    have "truthy (⟦e1⟧e s) = True ∨ truthy (⟦e2⟧e s) = True"
      using Or.prems(2) res by auto
    then show ?thesis
      unfolding res bfilter.simps
      by (rule gated_join_sound[OF Or.prems(1) _ Or.IH[OF Or.prems(1)]])
  next
    case False
    have v1: "truthy (⟦e1⟧e s) = False" and v2: "truthy (⟦e2⟧e s) = False"
      using Or.prems(2) False unfolding truthy_aval_Or by simp_all
    show ?thesis
      using Or.IH(1)[OF Or.IH(2)[OF Or.prems(1) v2] v1] by (simp add: False)
  qed
next
  case (Less e1 e2)
  have e1a: "⟦e1⟧e s ∈ γ (⟦e1⟧♯ d)" and e2a: "⟦e2⟧e s ∈ γ (⟦e2⟧♯ d)"
    using aval_abs_sound[OF Less.prems(1)] by simp_all
  have less: "(⟦e1⟧e s < ⟦e2⟧e s) = res" using Less.prems(2) by (auto split: if_splits)
  show ?case
    unfolding bfilter.simps Let_def case_prod_beta
    by (blast intro: Less.prems(1) inv_less_sound_fst[OF e1a e2a less]
                     inv_less_sound_snd[OF e1a e2a less])
next
  case (GreaterEq e1 e2)
  have e1a: "⟦e1⟧e s ∈ γ (⟦e1⟧♯ d)" and e2a: "⟦e2⟧e s ∈ γ (⟦e2⟧♯ d)"
    using aval_abs_sound[OF GreaterEq.prems(1)] by simp_all
  have less: "(⟦e1⟧e s < ⟦e2⟧e s) = (¬ res)" using GreaterEq.prems(2) by (auto split: if_splits)
```

```
  show ?case
    unfolding bfilter.simps Let_def case_prod_beta
    by (blast intro: GreaterEq.prems(1) inv_less_sound_fst[OF e1a e2a less]
                     inv_less_sound_snd[OF e1a e2a less])
next
  case (Greater e1 e2)
  have e1a: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ d)" and e2a: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ d)"
    using aval_abs_sound[OF Greater.prems(1)] by simp_all
  have less: "(⟦e2⟧ₑ s < ⟦e1⟧ₑ s) = res" using Greater.prems(2) by (auto split: if_splits)
  show ?case
    unfolding bfilter.simps Let_def case_prod_beta
    by (blast intro: Greater.prems(1) inv_less_sound_fst[OF e2a e1a less]
                     inv_less_sound_snd[OF e2a e1a less])
next
  case (LessEq e1 e2)
  have e1a: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ d)" and e2a: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ d)"
    using aval_abs_sound[OF LessEq.prems(1)] by simp_all
  have less: "(⟦e2⟧ₑ s < ⟦e1⟧ₑ s) = (¬ res)" using LessEq.prems(2) by (auto split: if_splits)
  show ?case
    unfolding bfilter.simps Let_def case_prod_beta
    by (blast intro: LessEq.prems(1) inv_less_sound_fst[OF e2a e1a less]
                     inv_less_sound_snd[OF e2a e1a less])
next
  case (Eq e1 e2)
  have e1a: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ d)" and e2a: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ d)"
    using aval_abs_sound[OF Eq.prems(1)] by simp_all
  have eq: "(⟦e1⟧ₑ s = ⟦e2⟧ₑ s) = res" using Eq.prems(2) by (auto split: if_splits)
  show ?case
    unfolding bfilter.simps Let_def case_prod_beta
    by (blast intro: Eq.prems(1) inv_eq_sound_fst[OF e1a e2a eq] inv_eq_sound_snd[OF e1a e2a eq])

next
  case (NotEq e1 e2)
  have e1a: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ d)" and e2a: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ d)"
    using aval_abs_sound[OF NotEq.prems(1)] by simp_all
  have eq: "(⟦e1⟧ₑ s = ⟦e2⟧ₑ s) = (¬ res)" using NotEq.prems(2) by (auto split: if_splits)
  show ?case
    unfolding bfilter.simps Let_def case_prod_beta
    by (blast intro: NotEq.prems(1) inv_eq_sound_fst[OF e1a e2a eq] inv_eq_sound_snd[OF e1a e2a eq])
qed (simp_all only: bfilter.simps Let_def case_prod_beta bfilter_default_sound)
```

`branch_lifted` is the Goblint-aligned branch semantics: the forward `feasible` gate on the whole condition ahead of `bfilter`, matching `Base.branch`'s `eval_rv` / `to_bool` dead-code gate ahead of `invariant`. An infeasible condition denotes `Bot` – no concrete successor, matching Goblint's `Deadcode` as an outer control-flow fact rather than a value of the domain – while every other case narrows via `bfilter` and returns `Lifted`. `tobool`'s definite answer, when present, is exactly `truthy` of every concrete value `aval_abs e d` represents; `truthy (⟦e⟧ₑ s) = pol` for a represented `s` then forces that answer to equal `pol`, so the `Bot` case below is exercised only when no represented `s` exists at all.

It is the same gate `bfilter` applies per disjunct inside its two join cases, here run on the whole condition – which is what makes it a control-flow fact rather than a narrowing.

A feasible condition's `bfilter` narrowing can itself still land on a witness-bottom store (`is_empty_state`, without being the raw `Lifted`'s structural `Bot`): `feasible` is only a necessary forward gate, and `bfilter`'s backward narrowing may subsequently discover a stronger contradiction. `branch_lifted` therefore re-normalizes `bfilter`'s result against `is_empty_state` (`normalize_lift`), matching Goblint's `BaseInvariant` raising `Deadcode` when its own refinement reaches bottom: `branch_lifted` is always `normalized_lift` (`branch_lifted_normalized`), so a caller can rely on structural `Bot` alone rather than re-testing `is_empty_state` on a `Lifted` result.

`bfilter_lifted` is not `bfilter`'s wrapper: a wrapper can only test the *whole* recursion's final result for emptiness, one collapse after every arm has already been joined pointwise, so it cannot undo pollution already baked into that pointwise result – the failure mode documented at `bfilter`'s own `And`/`Or` equations. `bfilter_lifted` is instead an independent primitive recursion, structurally mirroring `bfilter` one constructor at a time, that canonicalizes each `And`/`Or` arm (`Bot` on an infeasible or witness-bottom side, the join's own identity) *before* joining, at the `lifted` level, exactly where `bfilter` itself cannot afford to: `is_empty_state` has no code equation, so embedding it in `bfilter`'s own equations would carry that non-executability to every constructor, whereas `bfilter_lifted` is never code-generated. Every atomic case (`afilter`'s pointwise update has no pollution to fix) still reduces to plain `bfilter`, so the two recursions agree everywhere they overlap and cannot silently drift on those shared cases; all Boolean structure – `Not`, and both polarities of `And`/`Or` – is instead interpreted recursively in the `lifted` carrier: `Not` just flips polarity and recurses, the non-join polarity chains through `(≫=)` to propagate an arm's `Bot` immediately, and the two join equations are the deliberately independent pollution fix.

```
fun bfilter_lifted :: "exp => bool => 'a abs_state => 'a abs_state lifted" where
    "bfilter_lifted (Not b) res d = bfilter_lifted b (¬ res) d"
  | "bfilter_lifted (And b1 b2) True  d =
       bind (bfilter_lifted b2 True d) (bfilter_lifted b1 True)"
  | "bfilter_lifted (And b1 b2) False d =
       (if feasible b1 False d then bfilter_lifted b1 False d else Bot)
       ⊔ (if feasible b2 False d then bfilter_lifted b2 False d else Bot)"
  | "bfilter_lifted (Or  b1 b2) True  d =
       (if feasible b1 True d then bfilter_lifted b1 True d else Bot)
       ⊔ (if feasible b2 True d then bfilter_lifted b2 True d else Bot)"
  | "bfilter_lifted (Or  b1 b2) False d =
       bind (bfilter_lifted b2 False d) (bfilter_lifted b1 False)"
  | "bfilter_lifted (Eq  e1 e2) res  d = normalize_lift is_empty_state (bfilter (Eq e1 e2) res d)"
  | "bfilter_lifted e res d = normalize_lift is_empty_state (bfilter e res d)"

lemma bfilter_lifted_normalized [simp]:
  "normalized_lift is_empty_state (bfilter_lifted e pol d)"
proof (induction e arbitrary: pol d)
  case (And b1 b2)
  then show ?case
    by (cases pol) (auto intro: normalized_lift_bind)
next
  case (Or b1 b2)
  then show ?case
    by (cases pol) (auto intro: normalized_lift_bind)
qed simp_all
```

gated_join_sound for bfilter_lifted's join cases, against Bot.

```
lemma gated_join_lifted_sound:
  assumes st: "s ∈ γ d"
    and "truthy (⟦b1⟧e s) = pol ∨ truthy (⟦b2⟧e s) = pol"
    and f1: "truthy (⟦b1⟧e s) = pol ⟹ s ∈ γ⊥ f1"
    and f2: "truthy (⟦b2⟧e s) = pol ⟹ s ∈ γ⊥ f2"
  shows
    "s ∈ γ⊥ ((if feasible b1 pol d then f1 else Bot) ⊔ (if feasible b2 pol d then f2 else Bot))"
  using assms(2)
proof
  assume h: "truthy (⟦b1⟧e s) = pol"
  have "feasible b1 pol d" by (rule feasible_of_concrete[OF st h])
  with f1[OF h] show ?thesis by (simp add: gamma_state_lift_supI1)
next
  assume h: "truthy (⟦b2⟧e s) = pol"
  have "feasible b2 pol d" by (rule feasible_of_concrete[OF st h])
  with f2[OF h] show ?thesis by (simp add: gamma_state_lift_supI2)
qed

lemma bfilter_lifted_sound [intro]:
  assumes "s ∈ γ d" "truthy (⟦e⟧e s) = res"
  shows "s ∈ γ⊥ (bfilter_lifted e res d)"
using assms proof (induction e arbitrary: res d)
  case (Not e)
  have bv': "truthy (⟦e⟧e s) = (¬ res)" using Not.prems(2) by (auto split: if_splits)
  from Not.IH[OF Not.prems(1) bv'] show ?case by simp
next
  case (And e1 e2)
  show ?case
  proof (cases res)
    case True
    have v1: "truthy (⟦e1⟧e s) = True" and v2: "truthy (⟦e2⟧e s) = True"
      using And.prems(2) True unfolding truthy_aval_And by simp_all
    have h2: "s ∈ γ⊥ (bfilter_lifted e2 True d)"
      using And.IH(2)[OF And.prems(1) v2] .
    have res_eq: "res = True" using True by simp
    show ?thesis
      unfolding res_eq bfilter_lifted.simps
      using h2 v1 by (blast intro: And.IH(1))
  next
    case False
    have "truthy (⟦e1⟧e s) = False ∨ truthy (⟦e2⟧e s) = False"
      using And.prems(2) False by (auto split: if_splits)
    from gated_join_lifted_sound[OF And.prems(1) this And.IH[OF And.prems(1)]] False
    show ?thesis by (simp split del: if_split)
  qed
next
  case (Or e1 e2)
  show ?case
```

```
  proof (cases res)
    case True
    have "truthy (⟦e1⟧ₑ s) = True ∨ truthy (⟦e2⟧ₑ s) = True"
      using Or.prems(2) True by auto
    from gated_join_lifted_sound[OF Or.prems(1) this Or.IH[OF Or.prems(1)]] True
    show ?thesis by (simp split del: if_split)
  next
    case False
    have v1: "truthy (⟦e1⟧ₑ s) = False" and v2: "truthy (⟦e2⟧ₑ s) = False"
      using Or.prems(2) False unfolding truthy_aval_Or by simp_all
    have h2: "s ∈ γ⊥ (bfilter_lifted e2 False d)"
      using Or.IH(2)[OF Or.prems(1) v2] .
    have res_eq: "res = False" using False by simp
    show ?thesis
      unfolding res_eq bfilter_lifted.simps
      using h2 v1 by (blast intro: Or.IH(1))
  qed
qed (simp_all only: bfilter_lifted.simps gamma_state_normalize_lift bfilter_sound)
```

`branch_lifted` is `branch`'s reachability wrapper, now routed through `bfilter_lifted` rather than raw `bfilter`: the top-level `feasible` gate is unchanged (that is `branch`'s own concern, not the `And`/`Or` pollution `bfilter_lifted` fixes), but the filtering it gates is the corrected recursion, so `branch_lifted` inherits the improved precision without restating it.

```
definition branch_lifted :: "exp => bool => 'a abs_state => 'a abs_state lifted" where
  "branch_lifted e pol d = (if feasible e pol d then bfilter_lifted e pol d else Bot)"

lemma branch_lifted_normalized [simp]:
  "normalized_lift is_empty_state (branch_lifted e pol d)"
  unfolding branch_lifted_def by simp

lemma branch_lifted_sound [intro]:
  assumes "s ∈ γ d" "truthy (⟦e⟧ₑ s) = pol"
  shows "s ∈ γ⊥ (branch_lifted e pol d)"
proof -
  have g: "feasible e pol d" by (rule feasible_of_concrete[OF assms])
  have "s ∈ γ⊥ (bfilter_lifted e pol d)" by (rule bfilter_lifted_sound[OF assms])
  with g show ?thesis unfolding branch_lifted_def by simp
qed
```

A domain's branch operation has no room for `lifted`: it is one plain-state-to-plain-state operation among several, dispatched uniformly by the edge-action dispatcher with no `Bot`/`Lifted` case at that layer. `branch` is `branch_lifted` collapsed back through `collapse_lift`, so a domain can use the pollution-fixed recursion while still returning a plain `abs_state`: an infeasible or witness-bottom result becomes the plain type's own `bot`, exactly the value the branch operation's callers already treat as a dead program point. This replaces the old feasible-gated-raw- `bfilter` `branch`: that definition is strictly less precise on `And`/ `Or` conditions (the join-arm-pollution `bfilter`'s own comment documents), and this project keeps exactly one public plain-state branch operation rather than two of different precision.

```
definition branch :: "exp => bool => 'a abs_state => 'a abs_state" where
  "branch e pol d = collapse_lift (branch_lifted e pol d)"

lemma branch_sound [intro]:
  assumes "s ∈ γ d" "truthy (⟦e⟧ₑ s) = pol"
  shows "s ∈ γ (branch e pol d)"
  unfolding branch_def
  by (rule gamma_collapse_lift[where gam = gamma_state,
        OF branch_lifted_sound[OF assms] gamma_state_bot])
```

`branch` and `branch_lifted` denote the same concrete stores: `branch` only adapts the carrier to the plain-state shape a domain's branch operation needs, it does not lose precision relative to `branch_lifted`. Left bare rather than `[simp]`: its right-hand side is a strictly more complex normal form than its left, so declaring it a default rewrite would silently block every proof that reasons about `branch` directly (`branch_sound` among them) – cite it explicitly where the lifted view is actually needed.

```
end
```

Conservative inverse operator

A domain too coarse to invert an operator (e.g. sign, which cannot narrow either operand of a plus/minus/times from its result) instantiates that inverse with this shared no-op: both operands pass through unchanged. The first argument is the known result or truth value, so the same no-op serves the arithmetic inverses and the

comparison inverses. Its soundness obligation unfolds to the operands' own membership, so each instance discharges it with `inv_conservative_def`.

```
definition inv_conservative :: "'r => 'a => 'a => 'a * 'a" where
  "inv_conservative r a1 a2 = (a1, a2)"

end
```

2.10 Backward_Domain_Mono

```
theory Backward_Domain_Mono
  imports Backward_Domain
begin
```

Monotone and reductive backward filtering

Split out of `Voblint_Domain.Backward_Domain`: this theory proves the filters of `sound_refinement` reductive, from its reductive intersection, and adds `mono_refinement` for monotone operations on top of it.

Pairwise order for reductive/monotone inverse operators

A minimal componentwise order on `'a * 'a`, used only to state each inverse operator's monotonicity and reductiveness as a single fact about its returned pair rather than as two separately-assumed projections. Deliberately not `'a * 'a`'s own $\leq$: some already-transitively-imported theory instantiates `('a,'b) prod :: order` here with a non-componentwise (order-with-a-<-disjunct) definition, so relying on the ambient $\leq$ would silently pick up the wrong relation. This local, self-contained $\leq$`_pair` avoids that entirely.

```
abbreviation le_pair :: "'a::order × 'a ⇒ 'a × 'a ⇒ bool" where
  "le_pair p q ≡ fst p ≤ fst q ∧ snd p ≤ snd q"

lemma le_pair_fst: "le_pair p q ⟹ fst p ≤ fst q"
  by simp

lemma le_pair_snd: "le_pair p q ⟹ snd p ≤ snd q"
  by simp
```

Refined backward-analysis locale

Extends `sound_refinement` with two orthogonal strengthenings of the domain-author operators, bundled into one locale so a concrete domain proves both against a single interpretation rather than reproving `sound_refinement`'s base soundness once per strengthening:

- Monotonicity: the generic `afilter` / `bfilter` are then monotone in the abstract state (and target value) by the same induction that proves their soundness.

- Reductiveness: `intersect`'s result is below both input operands – refinement narrows, never enlarges. This is the only reductiveness this locale assumes: `afilter_reductive`/`bfilter_reductive` need it solely at `afilter`'s `V` case, where `intersect` is the sole narrowing step; the `inv_*` operators have no reductiveness assumption of their own, because none of the reductiveness proofs need one – each recursive `afilter`/`bfilter` step is bounded by its own induction hypothesis, not by a per-operator shrink fact.

Each `inv_*` operator's monotonicity contract is one `le_pair` fact about its returned pair rather than two separately-assumed `fst`/`snd` projections; the componentwise facts `afilter_mono`/`bfilter_mono`'s own induction needs are derived from these once, inside this locale.

Reductiveness is part of every backward domain, and monotonicity is a separate locale because a domain can be reductive without being monotone, and the executable lifted filtering needs only reductiveness. Int's `Refine_Fixpoint` mode is exactly that case: `refine` is reductive at every mode but monotone only off `Refine_Fixpoint` (`Int_Arithmetic`'s `refine_nonfixpoint_mono`), so it interprets `sound_refinement` and gets the precise, dead-arm-eliminating filtering, while `branch_mono` and the rest of the monotonicity layer stay out of its reach.

```
context sound_refinement
begin
```

Reductiveness of the whole recursion, not just its individual `intersect` steps: `afilter e a d`/`bfilter b res d` are always $\leq$ their input state, so a compound expression's re-narrowing of an already-settled location never revives it – once a location is `is_empty` it stays `is_empty` through every later step (⟦`?d1.0` $\leq$ `?d2.0; is_empty_state ?d2.0`⟧ ⟹ `is_empty_state ?d1.0`). Neither proof needs any monotonicity: `afilter`'s `V` case is bounded by `intersect_reductive2`, every compound case by its own induction hypothesis.

```
lemma afilter_Plus_unfold:
  "afilter (Plus e1 e2) a d =
     afilter e1 (fst (inv_plus a (⟦e1⟧♯ d) (⟦e2⟧♯ d)))
               (afilter e2 (snd (inv_plus a (⟦e1⟧♯ d) (⟦e2⟧♯ d))) d)"
  by (simp add: Let_def case_prod_beta)

lemma afilter_Minus_unfold:
  "afilter (Minus e1 e2) a d =
     afilter e1 (fst (inv_minus a (⟦e1⟧♯ d) (⟦e2⟧♯ d)))
```

```
              (afilter e2 (snd (inv_minus a ([[e1]]♯ d) ([[e2]]♯ d))) d)"
  by (simp add: Let_def case_prod_beta)

lemma afilter_Times_unfold:
  "afilter (Times e1 e2) a d =
     afilter e1 (fst (inv_times a ([[e1]]♯ d) ([[e2]]♯ d)))
              (afilter e2 (snd (inv_times a ([[e1]]♯ d) ([[e2]]♯ d))) d)"
  by (simp add: Let_def case_prod_beta)

lemma bfilter_Less_unfold:
  "bfilter (Less e1 e2) res d =
     afilter e1 (fst (inv_less res ([[e1]]♯ d) ([[e2]]♯ d)))
              (afilter e2 (snd (inv_less res ([[e1]]♯ d) ([[e2]]♯ d))) d)"
  by (simp add: Let_def case_prod_beta)

lemma bfilter_GreaterEq_unfold:
  "bfilter (GreaterEq e1 e2) res d =
     afilter e1 (fst (inv_less (¬ res) ([[e1]]♯ d) ([[e2]]♯ d)))
              (afilter e2 (snd (inv_less (¬ res) ([[e1]]♯ d) ([[e2]]♯ d))) d)"
  by (simp add: Let_def case_prod_beta)

lemma bfilter_Greater_unfold:
  "bfilter (Greater e1 e2) res d =
     afilter e2 (fst (inv_less res ([[e2]]♯ d) ([[e1]]♯ d)))
              (afilter e1 (snd (inv_less res ([[e2]]♯ d) ([[e1]]♯ d))) d)"
  by (simp add: Let_def case_prod_beta)

lemma bfilter_LessEq_unfold:
  "bfilter (LessEq e1 e2) res d =
     afilter e2 (fst (inv_less (¬ res) ([[e2]]♯ d) ([[e1]]♯ d)))
              (afilter e1 (snd (inv_less (¬ res) ([[e2]]♯ d) ([[e1]]♯ d))) d)"
  by (simp add: Let_def case_prod_beta)

lemma bfilter_Eq_unfold:
  "bfilter (Eq e1 e2) res d =
     afilter e1 (fst (inv_eq res ([[e1]]♯ d) ([[e2]]♯ d)))
              (afilter e2 (snd (inv_eq res ([[e1]]♯ d) ([[e2]]♯ d))) d)"
  by (simp add: Let_def case_prod_beta)

lemma bfilter_NotEq_unfold:
  "bfilter (NotEq e1 e2) res d =
     afilter e1 (fst (inv_eq (¬ res) ([[e1]]♯ d) ([[e2]]♯ d)))
              (afilter e2 (snd (inv_eq (¬ res) ([[e1]]♯ d) ([[e2]]♯ d))) d)"
  by (simp add: Let_def case_prod_beta)

lemma afilter_reductive: "afilter e a d ≤ d"
proof (induction e arbitrary: a d)
  case (V x) then show ?case by (auto simp: le_fun_def)
next
  case (Plus e1 e2) then show ?case
    unfolding afilter_Plus_unfold split_beta by (blast intro: order_trans)
next
  case (Minus e1 e2) then show ?case
    unfolding afilter_Minus_unfold split_beta by (blast intro: order_trans)
next
  case (Times e1 e2) then show ?case
    unfolding afilter_Times_unfold split_beta by (blast intro: order_trans)
qed simp_all

lemma bfilter_reductive: "bfilter b res d ≤ d"
proof (induction b arbitrary: res d)
  case (Not b)
  show ?case unfolding bfilter.simps by (rule Not.IH)
next
  case (And b1 b2)
  show ?case
  proof (cases res)
    case True
    have "bfilter b1 True (bfilter b2 True d) ≤ d"
      by (rule order_trans[OF And.IH(1) And.IH(2)])
    then show ?thesis using True by simp
  next
    case False
    then have res: "res = False" by simp
    show ?thesis unfolding res bfilter.simps
      by (intro sup_least) (simp_all add: And.IH)
```

```
  qed
next
  case (Or b1 b2)
  show ?case
  proof (cases res)
    case True
    then have res: "res = True" by simp
    show ?thesis unfolding res bfilter.simps
      by (intro sup_least) (simp_all add: Or.IH)
  next
    case False
    have "bfilter b1 False (bfilter b2 False d) ≤ d"
      by (rule order_trans[OF Or.IH(1) Or.IH(2)])
    then show ?thesis using False by simp
  qed
qed (simp_all only: bfilter.simps Let_def split_beta afilter_reductive
       order_trans[OF afilter_reductive afilter_reductive])
```

A witness-bottom incoming state forces `bfilter_lifted` to `Bot` regardless of what `feasible` answers: narrowing (`bfilter_reductive`) can only shrink a location, never repopulate one already empty, so whichever location witnessed `sigma`'s emptiness survives every leaf update and every join arm untouched. This is what lets an executable caller skip `bfilter_st_lift`'s `live_default_st` precondition entirely on a dead state: the answer is `Bot` either way, forward-gate false positive or not.

```
lemma bfilter_lifted_witness_bottom:
  assumes "is_empty_state sigma"
  shows "bfilter_lifted e pol sigma = Bot"
using assms proof (induction e arbitrary: pol)
  case (And b1 b2) then show ?case by (cases pol) simp_all
next
  case (Or b1 b2) then show ?case by (cases pol) simp_all
qed (simp_all add: is_empty_state_antimono[OF bfilter_reductive] del: bfilter.simps)

lemma branch_lifted_witness_bottom:
  assumes "is_empty_state sigma"
  shows "branch_lifted e pol sigma = Bot"
  unfolding branch_lifted_def using bfilter_lifted_witness_bottom[OF assms] by simp

lemma branch_witness_bottom:
  assumes "is_empty_state sigma"
  shows "branch e pol sigma = bot"
  unfolding branch_def using branch_lifted_witness_bottom[OF assms] by simp

end
```

The monotonicity half of the strengthening. Each inverse operator is assumed monotone in every argument, which is what carries `bfilter` and `branch` through their structural induction. `tobool_mono` is the one assumption that does not read that way: a definite truth value found at the coarser value must survive at the sharper one, and only a non-empty sharper value can be asked – an empty one decides everything vacuously.

```
locale mono_refinement =
  sound_refinement + mono_intersection intersect
    + mono_evaluator gamma_state aval_abs + mono_truth_test tobool +
  assumes inv_less_mono:
      "x1 ≤ x2 ⟹ y1 ≤ y2 ⟹ le_pair (inv_less res x1 y1) (inv_less res x2 y2)"
    and inv_eq_mono:
      "x1 ≤ x2 ⟹ y1 ≤ y2 ⟹ le_pair (inv_eq res x1 y1) (inv_eq res x2 y2)"
    and inv_plus_mono:
      "r1 ≤ r2 ⟹ x1 ≤ x2 ⟹ y1 ≤ y2 ⟹ le_pair (inv_plus r1 x1 y1) (inv_plus r2 x2 y2)"
    and inv_minus_mono:
      "r1 ≤ r2 ⟹ x1 ≤ x2 ⟹ y1 ≤ y2 ⟹ le_pair (inv_minus r1 x1 y1) (inv_minus r2 x2 y2)"
    and inv_times_mono:
      "r1 ≤ r2 ⟹ x1 ≤ x2 ⟹ y1 ≤ y2 ⟹ le_pair (inv_times r1 x1 y1) (inv_times r2 x2 y2)"
begin

lemmas inv_less_mono_fst [intro] = inv_less_mono[THEN le_pair_fst]
lemmas inv_less_mono_snd [intro] = inv_less_mono[THEN le_pair_snd]

lemmas inv_eq_mono_fst [intro] = inv_eq_mono[THEN le_pair_fst]
lemmas inv_eq_mono_snd [intro] = inv_eq_mono[THEN le_pair_snd]

lemmas inv_plus_mono_fst [intro] = inv_plus_mono[THEN le_pair_fst]
lemmas inv_plus_mono_snd [intro] = inv_plus_mono[THEN le_pair_snd]

lemmas inv_minus_mono_fst [intro] = inv_minus_mono[THEN le_pair_fst]
lemmas inv_minus_mono_snd [intro] = inv_minus_mono[THEN le_pair_snd]
```

```
lemmas inv_times_mono_fst [intro] = inv_times_mono[THEN le_pair_fst]
lemmas inv_times_mono_snd [intro] = inv_times_mono[THEN le_pair_snd]


lemma afilter_mono:
  "a1 ≤ a2 ⟹ d1 ≤ d2 ⟹ afilter e a1 d1 ≤ afilter e a2 d2"
proof (induction e arbitrary: a1 a2 d1 d2)
  case (N n)
  then show ?case by simp
next
  case (V x)
  show ?case
    unfolding afilter.simps
  proof (rule le_funI)
    fix y
    show "(d1(x := intersect a1 (d1 x))) y ≤ (d2(x := intersect a2 (d2 x))) y"
    proof (cases "y = x")
      case True
      thus ?thesis using intersect_mono[OF V.prems(1) le_funD[OF V.prems(2)]] by simp
    next
      case False thus ?thesis using le_funD[OF V.prems(2)] by simp
    qed
  qed
next
  case (Plus e1 e2)
  have v1: "⟦e1⟧♯ d1 ≤ ⟦e1⟧♯ d2" and v2: "⟦e2⟧♯ d1 ≤ ⟦e2⟧♯ d2"
    using aval_abs_mono[OF Plus.prems(2)] by simp_all
  show ?case unfolding afilter_Plus_unfold
    using v1 v2 Plus.prems
    by (blast intro: Plus.IH)
next
  case (Minus e1 e2)
  have v1: "⟦e1⟧♯ d1 ≤ ⟦e1⟧♯ d2" and v2: "⟦e2⟧♯ d1 ≤ ⟦e2⟧♯ d2"
    using aval_abs_mono[OF Minus.prems(2)] by simp_all
  show ?case unfolding afilter_Minus_unfold
    using v1 v2 Minus.prems
    by (blast intro: Minus.IH)
next
  case (Times e1 e2)
  have v1: "⟦e1⟧♯ d1 ≤ ⟦e1⟧♯ d2" and v2: "⟦e2⟧♯ d1 ≤ ⟦e2⟧♯ d2"
    using aval_abs_mono[OF Times.prems(2)] by simp_all
  show ?case unfolding afilter_Times_unfold
    using v1 v2 Times.prems
    by (blast intro: Times.IH)
next
  case (Less e1 e2) then show ?case by simp
next
  case (Div e1 e2) then show ?case by simp
next
  case (Mod e1 e2) then show ?case by simp
next
  case (GreaterEq e1 e2) then show ?case by simp
next
  case (Greater e1 e2) then show ?case by simp
next
  case (LessEq e1 e2) then show ?case by simp
next
  case (Eq e1 e2) then show ?case by simp
next
  case (NotEq e1 e2) then show ?case by simp
next
  case (Not e) then show ?case by simp
next
  case (And e1 e2) then show ?case by simp
next
  case (Or e1 e2) then show ?case by simp
qed
```

Monotonicity companion to `afilter_pair_sound`: the same two-operand filtering step, once `inv_less`/`inv_eq`/`inv_plus`/... has narrowed the operand pair monotonically, is itself monotone in both the narrowed pair and the incoming state. Stated generically here from the completed `afilter_mono`, so it cannot help `afilter_mono`'s own Plus/Minus/Times cases – `afilter_mono` is still mid-induction when they run – but lets later clients such as `bfilter_mono`'s Less/Eq cases apply it directly instead of repeating this chaining.

```
lemma afilter_pair_mono [intro]:
```

```
  assumes fst: "fst p1 ≤ fst p2" and snd: "snd p1 ≤ snd p2" and st: "d1 ≤ d2"
  shows "afilter e1 (fst p1) (afilter e2 (snd p1) d1)
       ≤ afilter e1 (fst p2) (afilter e2 (snd p2) d2)"
proof -
  have inner: "afilter e2 (snd p1) d1 ≤ afilter e2 (snd p2) d2"
    by (rule afilter_mono[OF snd st])
  show ?thesis by (rule afilter_mono[OF fst inner])
qed
```

Monotonicity companion to `bfilter_default_sound`: the same `inv_eq`- against-`0` reduction is monotone in `d`, following directly from `aval_abs_mono`, `inv_eq_mono`, and `afilter_mono`.

```
lemma bfilter_default_mono:
  assumes "d1 ≤ d2"
  shows "afilter e (fst (inv_eq (¬ res) ([e]♯ d1) ([N 0]♯ d1))) d1
       ≤ afilter e (fst (inv_eq (¬ res) ([e]♯ d2) ([N 0]♯ d2))) d2"
proof -
  have v1: "[e]♯ d1 ≤ [e]♯ d2" by (rule aval_abs_mono[OF assms])
  have v0: "[N 0]♯ d1 ≤ [N 0]♯ d2" by (rule aval_abs_mono[OF assms])
  have iv: "fst (inv_eq (¬ res) ([e]♯ d1) ([N 0]♯ d1))
            ≤ fst (inv_eq (¬ res) ([e]♯ d2) ([N 0]♯ d2))"
    using inv_eq_mono[OF v1 v0] by simp
  show ?thesis by (rule afilter_mono[OF iv assms])
qed
```

The forward gate is monotone in the direction its call sites need: a state whose forward value already rules the selected polarity out cannot become feasible by shrinking. `is_empty_antimono` settles the bottom case and `tobool_mono` the definite-answer case. `tobool_mono` explicitly excludes bottom operands – an author's `tobool` may answer arbitrarily there, since `gamma bot = {}` makes every answer vacuously sound – which is why `feasible` tests `is_empty` first rather than relying on `tobool` to agree across a bottom/non-bottom pair.

```
lemma feasible_mono:
  assumes "sigma1 ≤ sigma2" and "feasible e pol sigma1"
  shows "feasible e pol sigma2"
proof -
  have v: "[e]♯ sigma1 ≤ [e]♯ sigma2" by (rule aval_abs_mono[OF assms(1)])
  have nb1: "¬ is_empty ([e]♯ sigma1)" using assms(2) by (simp add: feasible_def)
  have nb2: "¬ is_empty ([e]♯ sigma2)" using nb1 is_empty_antimono[OF v] by blast
  have "tobool ([e]♯ sigma2) ≠ Some (¬ pol)"
  proof
    assume "tobool ([e]♯ sigma2) = Some (¬ pol)"
    then have "tobool ([e]♯ sigma1) = Some (¬ pol)"
      using tobool_mono[OF nb1 v] by simp
    then show False using assms(2) by (simp add: feasible_def)
  qed
  with nb2 show ?thesis by (simp add: feasible_def)
qed
```

`bfilter`'s two join cases test `feasible` and fall back to `bot` on failure; `bfilter_lifted`'s corresponding cases do the same against `Bot`. Both reduce to this one order fact, generic in the carrier: a monotone guard together with a monotone payload keeps the whole if-then-else monotone.

```
lemma if_bot_mono:
  fixes x y :: "'b::order_bot"
  assumes pq: "p ⟹ q"
      and xy: "x ≤ y"
  shows "(if p then x else bot) ≤ (if q then y else bot)"
  using assms by auto
```

Monotonicity of the feasibility-gated join `bfilter`'s two join cases compute. An infeasible disjunct contributes `bot`, which is below everything; a feasible one stays feasible in the larger state, and the recursive monotonicity fact carries through unchanged.

```
lemma gated_join_mono:
  assumes "sigma1 ≤ sigma2"
    and "bfilter b1 pol sigma1 ≤ bfilter b1 pol sigma2"
    and "bfilter b2 pol sigma1 ≤ bfilter b2 pol sigma2"
  shows "(if feasible b1 pol sigma1 then bfilter b1 pol sigma1 else bot)
           ⊔ (if feasible b2 pol sigma1 then bfilter b2 pol sigma1 else bot)
         ≤ (if feasible b1 pol sigma2 then bfilter b1 pol sigma2 else bot)
           ⊔ (if feasible b2 pol sigma2 then bfilter b2 pol sigma2 else bot)"
  by (intro sup_mono if_bot_mono feasible_mono[OF assms(1)] assms(2,3))

lemma bfilter_mono:
  "d1 ≤ d2 ⟹ bfilter b res d1 ≤ bfilter b res d2"
proof (induction b arbitrary: res d1 d2)
```

```
  case (Not b) show ?case unfolding bfilter.simps by (rule Not.IH[OF Not.prems])
next
  case (And b1 b2)
  show ?case
  proof (cases res)
    case True
    have c: "bfilter b2 True d1 ≤ bfilter b2 True d2" by (rule And.IH(2)[OF And.prems])
    have "bfilter b1 True (bfilter b2 True d1) ≤ bfilter b1 True (bfilter b2 True d2)"
      by (rule And.IH(1)[OF c])
    thus ?thesis using True by simp
  next
    case False
    then have res: "res = False" by simp
    show ?thesis
      unfolding res bfilter.simps
      by (rule gated_join_mono[OF And.prems And.IH[OF And.prems]])
  qed
next
  case (Or b1 b2)
  show ?case
  proof (cases res)
    case True
    then have res: "res = True" by simp
    show ?thesis
      unfolding res bfilter.simps
      by (rule gated_join_mono[OF Or.prems Or.IH[OF Or.prems]])
  next
    case False
    have c: "bfilter b2 False d1 ≤ bfilter b2 False d2" by (rule Or.IH(2)[OF Or.prems])
    have "bfilter b1 False (bfilter b2 False d1) ≤ bfilter b1 False (bfilter b2 False d2)"
      by (rule Or.IH(1)[OF c])
    thus ?thesis using False by simp
  qed
next
  case (Less e1 e2)
  have v1: "⟦e1⟧♯ d1 ≤ ⟦e1⟧♯ d2" and v2: "⟦e2⟧♯ d1 ≤ ⟦e2⟧♯ d2"
    using aval_abs_mono[OF Less.prems] by simp_all
  show ?case unfolding bfilter_Less_unfold
    by (rule afilter_pair_mono[OF inv_less_mono_fst[OF v1 v2] inv_less_mono_snd[OF v1 v2]
                                  Less.prems])
next
  case (GreaterEq e1 e2)
  have v1: "⟦e1⟧♯ d1 ≤ ⟦e1⟧♯ d2" and v2: "⟦e2⟧♯ d1 ≤ ⟦e2⟧♯ d2"
    using aval_abs_mono[OF GreaterEq.prems] by simp_all
  show ?case unfolding bfilter_GreaterEq_unfold
    by (rule afilter_pair_mono[OF inv_less_mono_fst[OF v1 v2] inv_less_mono_snd[OF v1 v2]
                                  GreaterEq.prems])
next
  case (Greater e1 e2)
  have v1: "⟦e1⟧♯ d1 ≤ ⟦e1⟧♯ d2" and v2: "⟦e2⟧♯ d1 ≤ ⟦e2⟧♯ d2"
    using aval_abs_mono[OF Greater.prems] by simp_all
  show ?case unfolding bfilter_Greater_unfold
    by (rule afilter_pair_mono[OF inv_less_mono_fst[OF v2 v1] inv_less_mono_snd[OF v2 v1]
                                  Greater.prems])
next
  case (LessEq e1 e2)
  have v1: "⟦e1⟧♯ d1 ≤ ⟦e1⟧♯ d2" and v2: "⟦e2⟧♯ d1 ≤ ⟦e2⟧♯ d2"
    using aval_abs_mono[OF LessEq.prems] by simp_all
  show ?case unfolding bfilter_LessEq_unfold
    by (rule afilter_pair_mono[OF inv_less_mono_fst[OF v2 v1] inv_less_mono_snd[OF v2 v1]
                                  LessEq.prems])
next
  case (Eq e1 e2)
  have v1: "⟦e1⟧♯ d1 ≤ ⟦e1⟧♯ d2" and v2: "⟦e2⟧♯ d1 ≤ ⟦e2⟧♯ d2"
    using aval_abs_mono[OF Eq.prems] by simp_all
  show ?case unfolding bfilter_Eq_unfold
    by (rule afilter_pair_mono[OF inv_eq_mono_fst[OF v1 v2] inv_eq_mono_snd[OF v1 v2] Eq.prems])

next
  case (NotEq e1 e2)
  have v1: "⟦e1⟧♯ d1 ≤ ⟦e1⟧♯ d2" and v2: "⟦e2⟧♯ d1 ≤ ⟦e2⟧♯ d2"
    using aval_abs_mono[OF NotEq.prems] by simp_all
  show ?case unfolding bfilter_NotEq_unfold
    by (rule afilter_pair_mono[OF inv_eq_mono_fst[OF v1 v2] inv_eq_mono_snd[OF v1 v2] NotEq.prems])
qed (simp_all only: bfilter.simps Let_def case_prod_beta bfilter_default_mono)
```

`bfilter_lifted`'s atomic shapes reduce to `bfilter_mono` lifted through `normalize_lift`; the sequential `And`/`Or`

cases lift (≫=)'s own premise (case `Bot` is trivial, case `Lifted` is the head recursion's own monotonicity fact), and the join cases are `gated_join_mono` restated against `Bot`.

```
lemma gated_join_lifted_mono:
  assumes "sigma1 ≤ sigma2"
    and "bfilter_lifted b1 pol sigma1 ≤ bfilter_lifted b1 pol sigma2"
    and "bfilter_lifted b2 pol sigma1 ≤ bfilter_lifted b2 pol sigma2"
  shows "(if feasible b1 pol sigma1 then bfilter_lifted b1 pol sigma1 else Bot)
           ⊔ (if feasible b2 pol sigma1 then bfilter_lifted b2 pol sigma1 else Bot)
         ≤ (if feasible b1 pol sigma2 then bfilter_lifted b1 pol sigma2 else Bot)
           ⊔ (if feasible b2 pol sigma2 then bfilter_lifted b2 pol sigma2 else Bot)"
  unfolding bot_lifted_eq[symmetric]
  by (intro sup_mono if_bot_mono feasible_mono[OF assms(1)] assms(2,3))

lemma bfilter_lifted_mono:
  "sigma1 ≤ sigma2 ⟹ bfilter_lifted e pol sigma1 ≤ bfilter_lifted e pol sigma2"
proof (induction e arbitrary: pol sigma1 sigma2)
  case (And b1 b2)
  show ?case
  proof (cases pol)
    case True
    have step: "bfilter_lifted b2 True sigma1 ≤ bfilter_lifted b2 True sigma2"
      by (rule And.IH(2)[OF And.prems])
    have ih1: "⋀a b. a ≤ b ⟹ bfilter_lifted b1 True a ≤ bfilter_lifted b1 True b"
      using And.IH(1) by simp
    show ?thesis using True by (simp add: bind_lift_mono2[OF step ih1])
  next
    case False
    then have pol: "pol = False" by simp
    show ?thesis
      unfolding pol bfilter_lifted.simps
      by (rule gated_join_lifted_mono[OF And.prems And.IH[OF And.prems]])
  qed
next
  case (Or b1 b2)
  show ?case
  proof (cases pol)
    case True
    then have pol: "pol = True" by simp
    show ?thesis
      unfolding pol bfilter_lifted.simps
      by (rule gated_join_lifted_mono[OF Or.prems Or.IH[OF Or.prems]])
  next
    case False
    have step: "bfilter_lifted b2 False sigma1 ≤ bfilter_lifted b2 False sigma2"
      by (rule Or.IH(2)[OF Or.prems])
    have ih1: "⋀a b. a ≤ b ⟹ bfilter_lifted b1 False a ≤ bfilter_lifted b1 False b"
      using Or.IH(1) by simp
    show ?thesis using False by (simp add: bind_lift_mono2[OF step ih1])
  qed
qed (simp_all only: bfilter_lifted.simps normalize_state_lift_mono bfilter_mono)

lemma branch_lifted_mono:
  assumes "sigma1 ≤ sigma2"
  shows "branch_lifted e pol sigma1 ≤ branch_lifted e pol sigma2"
proof (cases "feasible e pol sigma1")
  case True
  have "feasible e pol sigma2" by (rule feasible_mono[OF assms True])
  with True assms show ?thesis
    unfolding branch_lifted_def by (simp add: bfilter_lifted_mono)
next
  case False
  then show ?thesis unfolding branch_lifted_def by simp
qed

lemma branch_mono:
  assumes "sigma1 ≤ sigma2"
  shows "branch e pol sigma1 ≤ branch e pol sigma2"
  unfolding branch_def
  by (rule collapse_lift_mono[OF branch_lifted_mono[OF assms]])
```

`bfilter_lifted` only ever narrows further than raw `bfilter`: each leaf either falls through to `bfilter` unchanged or short-circuits to `Bot`, and every `And`/`Or` recursion step preserves that ordering by `bfilter_lifted_mono`. `branch_le_bfilter` transports this to the collapsed carrier: callers that only need an upper bound on `branch`'s result – e.g. post-fixpoint checks – can reuse their existing `bfilter`-level reasoning through it instead of re-deriving one against `branch`'s definition.

```
lemma gated_join_lifted_le_raw:
  fixes y1 y2 :: "'a abs_state"
  assumes "x1 ≤ Lifted y1" and "x2 ≤ Lifted y2"
  shows "(if p1 then x1 else Bot) ⊔ (if p2 then x2 else Bot)
    ≤ Lifted ((if p1 then y1 else bot) ⊔ (if p2 then y2 else bot))"
  using assms sup_mono[OF assms] by (cases p1; cases p2) simp_all

lemma bfilter_lifted_le_raw: "bfilter_lifted e pol sigma ≤ Lifted (bfilter e pol sigma)"
proof (induction e arbitrary: pol sigma)
  case (And b1 b2)
  show ?case
  proof (cases pol)
    case True
    show ?thesis
    proof (cases "bfilter_lifted b2 True sigma")
      case Bot
      then show ?thesis using True by simp
    next
      case (Lifted t)
      have le_t: "t ≤ bfilter b2 True sigma"
        using And.IH(2)[of True sigma] Lifted by simp
      have "bfilter_lifted b1 True t ≤ bfilter_lifted b1 True (bfilter b2 True sigma)"
        by (rule bfilter_lifted_mono[OF le_t])
      also have "... ≤ Lifted (bfilter b1 True (bfilter b2 True sigma))"
        using And.IH(1)[of True "bfilter b2 True sigma"] by simp
      finally show ?thesis using True Lifted by simp
    qed
  next
    case False
    then have pol: "pol = False" by simp
    show ?thesis
      unfolding pol bfilter_lifted.simps bfilter.simps
      by (rule gated_join_lifted_le_raw[OF And.IH])
  qed
next
  case (Or b1 b2)
  show ?case
  proof (cases pol)
    case True
    then have pol: "pol = True" by simp
    show ?thesis
      unfolding pol bfilter_lifted.simps bfilter.simps
      by (rule gated_join_lifted_le_raw[OF Or.IH])
  next
    case False
    show ?thesis
    proof (cases "bfilter_lifted b2 False sigma")
      case Bot
      then show ?thesis using False by simp
    next
      case (Lifted t)
      have le_t: "t ≤ bfilter b2 False sigma"
        using Or.IH(2)[of False sigma] Lifted by simp
      have "bfilter_lifted b1 False t ≤ bfilter_lifted b1 False (bfilter b2 False sigma)"
        by (rule bfilter_lifted_mono[OF le_t])
      also have "... ≤ Lifted (bfilter b1 False (bfilter b2 False sigma))"
        using Or.IH(1)[of False "bfilter b2 False sigma"] by simp
      finally show ?thesis using False Lifted by simp
    qed
  qed
qed (simp_all add: normalize_lift_def)

lemma branch_lifted_le_raw: "branch_lifted e pol sigma ≤ Lifted (bfilter e pol sigma)"
  unfolding branch_lifted_def
  by (cases "feasible e pol sigma") (simp_all add: bfilter_lifted_le_raw)

lemma branch_le_bfilter: "branch e pol sigma ≤ bfilter e pol sigma"
  unfolding branch_def
  using collapse_lift_mono[OF branch_lifted_le_raw] by simp

end

end
```

2.11 Numeric_Queries

```
theory Numeric_Queries
  imports State_Concretization
begin
```

Numeric relational query interface

Entailment/refutation of `<`/`=` over an abstract numeric value is not check-specific: it is the same question a backward/guard domain answers, phrased as a query instead of a narrowing. `less`/`eq` package that answer in Goblint's own shape – `ID.lt`/`ID.eq` – a definite `Some True`, a definite `Some False`, or `None` when neither direction is provable, the same three-way shape `tobool` already gives a single abstract value's truthiness.

`sound_numeric_queries` fixes that shape together with its soundness obligation, as `sound_truth_test` does for `tobool`. Kept in a session with no `exp`/`store` concept at all, so a domain that only has these two operations could still interpret it: the `sound_refinement` locale refines an abstract value under an assumed truth value; `sound_numeric_queries` instead classifies an already-fixed pair of values as provably related, provably unrelated, or neither.

`numeric_query_judgments`, below, is how every domain here builds such an instance: it packages four yes/no judgments into the two queries and proves the obligation once, so no domain restates that step. Where the four judgments come from is the domain's business. Every `sound_refinement` instance reads them off its own narrowing operators, in the `Backward_Numeric_Queries` theory, and gets an instance for free; a concrete domain may instead supply sharper, hand-tuned judgments when that derivation loses precision it cannot recover, or judgments unrelated to narrowing altogether.

The soundness obligation, stated so that a `Some` answer is exact rather than merely one-sided: every concrete pair drawn from the two values must relate the way the answer says. `None` carries no obligation at all, which is what lets a domain answer it whenever it cannot decide.

```
locale sound_numeric_queries =
  fixes less :: "'a::numeric_domain ⇒ 'a ⇒ bool option"
    and eq :: "'a ⇒ 'a ⇒ bool option"
  assumes less_sound[intro]:
      "less a b = Some r ⟹ i ∈ γ a ⟹ j ∈ γ b ⟹ (i < j) = r"
    and eq_sound[intro]:
      "eq a b = Some r ⟹ i ∈ γ a ⟹ j ∈ γ b ⟹ (i = j) = r"
```

Building an instance from four judgments

A domain rarely decides `less`/`eq` in one step. It decides four separate judgments – every witness pair satisfies `<`, every witness pair refutes it, and the same two for `=` – and the two queries are those four read in order: prefer the affirmative judgment, fall back to the refuting one, answer `None` when neither fires. `numeric_query_judgments` fixes the four and their soundness, and nothing else, so a domain interprets it whatever the judgments were read off: a narrowing operator, a comparison of bounds, a test that one side denotes nothing.

The two queries are named `query_less`/`query_eq` rather than `less`/`eq`, which would share a base name with the order's `(<)` and the vendored solver's `eq`. Interpreting the locale introduces them as ordinary top-level constants through a **defines** clause, and therefore with the code equation a constant living in a still-abstract locale context does not have.

`less_true` and `less_false` (and `eq_true` and `eq_false`) can both hold at once only when an operand denotes nothing: a live witness pair on both sides would have to satisfy `i < j` and ¬ `i < j` simultaneously. So on any pair of feasible operands the `if` choice between `Some True` and `Some False` below is never live.

```
locale numeric_query_judgments =
  fixes less_true :: "'a::numeric_domain ⇒ 'a ⇒ bool"
    and less_false :: "'a ⇒ 'a ⇒ bool"
    and eq_true :: "'a ⇒ 'a ⇒ bool"
    and eq_false :: "'a ⇒ 'a ⇒ bool"
  assumes less_true_sound:
      "less_true a b ⟹ i ∈ γ a ⟹ j ∈ γ b ⟹ i < j"
    and less_false_sound:
      "less_false a b ⟹ i ∈ γ a ⟹ j ∈ γ b ⟹ ¬ i < j"
    and eq_true_sound:
      "eq_true a b ⟹ i ∈ γ a ⟹ j ∈ γ b ⟹ i = j"
    and eq_false_sound:
      "eq_false a b ⟹ i ∈ γ a ⟹ j ∈ γ b ⟹ i ≠ j"
begin

definition query_less :: "'a ⇒ 'a ⇒ bool option" where
  "query_less a b = (if less_true a b then Some True else if less_false a b then Some False else None)"
```

```
definition query_eq :: "'a ⇒ 'a ⇒ bool option" where
  "query_eq a b = (if eq_true a b then Some True else if eq_false a b then Some False else None)"

lemma less_opt_sound:
  assumes "query_less a b = Some r" and "i ∈ γ a" and "j ∈ γ b"
  shows "(i < j) = r"
  using assms less_true_sound less_false_sound unfolding query_less_def by (auto split: if_splits)

lemma eq_opt_sound:
  assumes "query_eq a b = Some r" and "i ∈ γ a" and "j ∈ γ b"
  shows "(i = j) = r"
  using assms eq_true_sound eq_false_sound unfolding query_eq_def by (auto split: if_splits)

end

sublocale numeric_query_judgments ⊆ sound_numeric_queries query_less query_eq
proof
  fix a b r i j
  assume "query_less a b = Some r" and "i ∈ γ a" and "j ∈ γ b"
  then show "(i < j) = r" by (rule less_opt_sound)
next
  fix a b r i j
  assume "query_eq a b = Some r" and "i ∈ γ a" and "j ∈ γ b"
  then show "(i = j) = r" by (rule eq_opt_sound)
qed

end
```

2.12 Backward_Numeric_Queries

```
theory Backward_Numeric_Queries
  imports Backward_Domain Numeric_Queries
begin
```

Queries derived from backward inversion

Any `sound_inverse_ops` instance already proves `inv_less_sound`/ `intersect_sound`, so it answers entailment/refutation of `<`/`=` over two atomic values for free: narrowing under an assumed truth value yields an `is_empty` component only if no represented witness pair can satisfy that assumption. Thus emptiness of the inverse result gives a sound entailment/refutation test, though an imprecise inverse may still answer "unknown" even when the relation is already concrete-semantically fixed. The four judgments need the inverse operators and nothing about states, so they are defined in `sound_inverse_ops`'s own context; every `sound_refinement` interpretation extends it and gets them with no restated proof obligation. Turning the four into the two queries a caller asks for is not backward-specific and is not done here: this file ends by interpreting `numeric_query_judgments` at them, so every `sound_inverse_ops` instance is a numeric-query instance too.

Classification tests `is_empty`, not canonical-`bot` equality: a value can denote `{}` without being the representation's chosen `bot` element (`numeric_domain`'s whole reason for distinguishing the two), so a `= bot` test would underclassify any domain with non-canonical empty representations. `less_true`/`less_false`/`eq_true`/`eq_false` stay implementation judgments, not the public interface `Voblint_Domain.Numeric_Queries` gives: a caller after the generic default wants `less`/`eq`; these four remain named and proved because `Sign_Backward` and `Int_Classify` cite them directly off a concrete `sound_refinement` interpretation to state that a domain's own sharper, hand-tuned query agrees with this generic default.

```
context sound_inverse_ops
begin
```

Comparison judgments

A comparison is decided by narrowing against its own negation: if refining the two values under `¬ a < b` empties either side, no concrete pair can violate `a < b`, so the judgment holds. Nothing new is assumed of the domain – the answer comes from the narrowing operator it already supplies.

```
definition less_true :: "'a ⇒ 'a ⇒ bool" where
  "less_true a b ⟷ is_empty (fst (inv_less False a b)) ∨ is_empty (snd (inv_less False a b))"

definition less_false :: "'a ⇒ 'a ⇒ bool" where
  "less_false a b ⟷ is_empty (fst (inv_less True a b)) ∨ is_empty (snd (inv_less True a b))"

lemma less_true_sound:
  assumes "less_true a b" and "i ∈ γ a" and "j ∈ γ b"
  shows "i < j"
  using assms inv_less_sound[OF assms(2,3), of False]
  unfolding less_true_def
  by (auto simp: is_empty_correct)

lemma less_false_sound:
  assumes "less_false a b" and "i ∈ γ a" and "j ∈ γ b"
  shows "¬ i < j"
  using assms inv_less_sound[OF assms(2,3), of True]
  unfolding less_false_def
  by (auto simp: is_empty_correct)
```

Equality judgments

Two independent derivations, from different existing capabilities: `eq_true` from `less_false` in both directions (integer trichotomy – if every pair is provably `¬ i < j` and every pair is provably `¬ j < i`, every pair is `i = j`); `eq_false` from `intersect` (a common witness would belong to its concretization, so an `is_empty` intersection rules one out).

```
definition eq_true :: "'a ⇒ 'a ⇒ bool" where
  "eq_true a b ⟷ less_false a b ∧ less_false b a"

lemma eq_true_sound:
  assumes "eq_true a b" and "i ∈ γ a" and "j ∈ γ b"
  shows "i = j"
  using assms less_false_sound[of a b i j] less_false_sound[of b a j i]
  unfolding eq_true_def
  by fastforce
```

```
definition eq_false :: "'a ⇒ 'a ⇒ bool" where
  "eq_false a b ⟷ is_empty (intersect a b)"

lemma eq_false_sound:
  assumes "eq_false a b" and "i ∈ γ a" and "j ∈ γ b"
  shows "i ≠ j"
  using assms intersect_sound
  unfolding eq_false_def
  by (fastforce simp: is_empty_correct)

end
```

Packaging the derived judgments

The qualifier is mandatory rather than cosmetic: the packaging locale assumes the same four soundness statements this context proves, under the same four names, so an unqualified registration would try to declare each twice in every **global_interpretation** of `sound_inverse_ops`.

```
sublocale sound_inverse_ops ⊆ queries:
  numeric_query_judgments less_true less_false eq_true eq_false
  by unfold_locales
     (fact less_true_sound, fact less_false_sound, fact eq_true_sound, fact eq_false_sound)

end
```

2.13 Interval_Lattice

```
theory Interval_Lattice
  imports "Voblint_Domain.State_Concretization" "Voblint_VIMP.VIMP_Proc"
begin
```

Interval lattice

An interval `[l, u]` abstracts the integers `{n. l ≤ n ∧ n ≤ u}`. Bounds are extended integers (`MinInf` / `Fin n` / `PlusInf`). Empty (e.g. `[+inf, -inf]`) is `bot`; `[-inf, +inf]` is `top`.

Extended integer bounds

An interval needs endpoints that can say "no lower limit" and "no upper limit", so `eint` adds `MinInf` and `PlusInf` to the integers. Ordering puts `MinInf` below and `PlusInf` above everything, which makes `eint` a linear order; addition and subtraction extend the integer operations, with the mixed-infinity cases (`PlusInf + MinInf` and its mirror) pinned to a fixed choice rather than left undefined, since a total function is what

```
datatype eint = MinInf | Fin (eint_val: int) | PlusInf

fun eint_le :: "eint => eint => bool" where
    "eint_le MinInf  _       = True"
  | "eint_le _       PlusInf = True"
  | "eint_le (Fin n) (Fin m) = (n <= m)"
  | "eint_le _       MinInf  = False"
  | "eint_le PlusInf _       = False"

lemma eint_le_refl [simp]: "eint_le x x"
  by (cases x) simp_all

lemma eint_le_antisym: "eint_le x y ⟹ eint_le y x ⟹ x = y"
  by (cases x; cases y) simp_all

lemma eint_le_trans: "eint_le x y ⟹ eint_le y z ⟹ eint_le x z"
  by (cases x; cases y; cases z) simp_all

lemma eint_le_PlusInf [simp]: "eint_le x PlusInf"
  by (cases x) simp_all

lemma eint_le_MinInf_left [simp]: "eint_le MinInf x"
  by simp

lemma eint_le_linear: "eint_le x y ∨ eint_le y x"
  by (cases x; cases y) auto

instantiation eint :: ord begin
definition less_eq_eint :: "eint => eint => bool" where "less_eq_eint = eint_le"
definition less_eint    :: "eint => eint => bool" where
  "(a :: eint) < b = (eint_le a b ∧ ¬ eint_le b a)"
instance ..
end

declare less_eq_eint_def [simp]

instance eint :: linorder
proof intro_classes
  fix x y z :: eint
  show "(x < y) = (x ≤ y ∧ ¬ y ≤ x)" unfolding less_eint_def less_eq_eint_def by simp
  show "x ≤ x"                          unfolding less_eq_eint_def by (rule eint_le_refl)
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"    unfolding less_eq_eint_def by (rule eint_le_trans)
  show "x ≤ y ⟹ y ≤ x ⟹ x = y"    unfolding less_eq_eint_def by (rule eint_le_antisym)
  show "x ≤ y ∨ y ≤ x"                  unfolding less_eq_eint_def by (rule eint_le_linear)
qed


instantiation eint :: plus begin
fun plus_eint :: "eint => eint => eint" where
    "plus_eint (Fin n)   (Fin m)   = Fin (n + m)"
  | "plus_eint (Fin _)   MinInf    = MinInf"
  | "plus_eint (Fin _)   PlusInf   = PlusInf"
  | "plus_eint MinInf    MinInf    = MinInf"
  | "plus_eint MinInf    (Fin _)   = MinInf"
  | "plus_eint MinInf    PlusInf   = MinInf"
  | "plus_eint PlusInf   MinInf    = PlusInf"
```

```
  | "plus_eint PlusInf   (Fin _)   = PlusInf"
  | "plus_eint PlusInf   PlusInf   = PlusInf"
instance ..
end

instantiation eint :: minus begin
fun minus_eint :: "eint => eint => eint" where
    "minus_eint (Fin n)   (Fin m)   = Fin (n - m)"
  | "minus_eint (Fin _)   MinInf    = PlusInf"
  | "minus_eint (Fin _)   PlusInf   = MinInf"
  | "minus_eint MinInf    MinInf    = MinInf"
  | "minus_eint MinInf    (Fin _)   = MinInf"
  | "minus_eint MinInf    PlusInf   = MinInf"
  | "minus_eint PlusInf   MinInf    = PlusInf"
  | "minus_eint PlusInf   (Fin _)   = PlusInf"
  | "minus_eint PlusInf   PlusInf   = PlusInf"
instance ..
end

lemma eint_le_PlusInf_iff: "eint_le PlusInf z ⟷ z = PlusInf"
  by (cases z) auto
lemma eint_le_MinInf_iff: "eint_le z MinInf ⟷ z = MinInf"
  by (cases z) auto

lemma eint_plus_mono:
  "eint_le a1 a2 ⟹ eint_le b1 b2 ⟹ eint_le (a1 + b1) (a2 + b2)"
  by (cases a1; cases a2; cases b1; cases b2; auto)

lemma eint_minus_mono:
  "eint_le a1 a2 ⟹ eint_le b2 b1 ⟹ eint_le (a1 - b1) (a2 - b2)"
  by (cases a1; cases a2; cases b1; cases b2; auto)
```

`Fin` embeds `int` into `eint` as an order isomorphism onto the finite bounds, so it commutes with `min`/`max`.

```
lemma Fin_min: "Fin (min i j) = min (Fin i) (Fin j)"
  by (simp add: min_def)

lemma Fin_max: "Fin (max i j) = max (Fin i) (Fin j)"
  by (simp add: max_def)

lemma eint_min_mono:
  "eint_le a1 a2 ⟹ eint_le b1 b2 ⟹ eint_le (min a1 b1) (min a2 b2)"
  by (cases a1; cases a2; cases b1; cases b2; auto simp: min_def)

lemma eint_max_mono:
  "eint_le a1 a2 ⟹ eint_le b1 b2 ⟹ eint_le (max a1 b1) (max a2 b2)"
  by (cases a1; cases a2; cases b1; cases b2; auto simp: max_def)
```

Carrier and concretization

An interval `Ivl l u` over extended integers denotes the integers between its bounds; a reversed pair denotes the empty set, so the raw carrier has many empty values.

```
datatype ivl = Ivl (ivl_lower: eint) (ivl_upper: eint)   — Ivl l u = [l, u]
```

Named exhaustion, cited instead of `(cases x) auto` at every destructuring site. It stays untagged on purpose: its conclusion matches any term, so the classical reasoner would apply it to every goal.

```
lemma ivl_exhaustE:
  obtains l u where "x = Ivl l u"
  by (cases x) auto


fun gamma_ivl :: "ivl => int set" where
    "gamma_ivl (Ivl l u) = {n. l ≤ Fin n ∧ Fin n ≤ u}"

definition ivl_of_int :: "int ⇒ ivl" where
  [simp]: "ivl_of_int n = Ivl (Fin n) (Fin n)"
```

Order

Interval inclusion on the bounds, with `Ivl PlusInf MinInf` as bottom and `ivl_top` as the unbounded interval.

```
instantiation ivl :: ord begin
definition less_eq_ivl :: "ivl => ivl => bool" where
  "(a::ivl) <= b = (case (a, b) of (Ivl l1 u1, Ivl l2 u2) => l2 ≤ l1 ∧ u1 ≤ u2)"
definition less_ivl :: "ivl => ivl => bool" where
```

```
  "(a::ivl) < b = (a <= b ∧ ¬ b <= a)"
instance ..
end

instantiation ivl :: bot begin
definition bot_ivl :: ivl where
  "bot_ivl = Ivl PlusInf MinInf"
instance ..
end

instantiation ivl :: order begin
instance proof intro_classes
  fix x y z :: ivl
  show "(x < y) = (x ≤ y ∧ ¬ y ≤ x)"
    unfolding less_ivl_def by simp
  show "x ≤ x"
    unfolding less_eq_ivl_def by (cases x) simp
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
    unfolding less_eq_ivl_def by (cases x; cases y; cases z) (auto intro: eint_le_trans)
  show "x ≤ y ⟹ y ≤ x ⟹ x = y"
    unfolding less_eq_ivl_def by (cases x; cases y) (auto intro: eint_le_antisym)
qed
end

instantiation ivl :: order_bot begin
instance proof intro_classes
  fix x :: ivl
  show "bot ≤ x"
    unfolding less_eq_ivl_def bot_ivl_def by (cases x) simp
qed
end

definition ivl_top :: ivl where
  "ivl_top = Ivl MinInf PlusInf"

instantiation ivl :: top begin
definition "top_ivl = ivl_top"
instance ..
end

instantiation ivl :: order_top begin
instance proof intro_classes
  fix x :: ivl
  show "x ≤ top"
    unfolding top_ivl_def by (cases x) (simp add: less_eq_ivl_def ivl_top_def)
qed
end

lemma gamma_ivl_bot: "gamma_ivl bot = {}"
  unfolding bot_ivl_def by auto

lemma gamma_ivl_top: "gamma_ivl ivl_top = UNIV"
  unfolding ivl_top_def by auto

lemma gamma_ivl_mono:
  "a ≤ b ⟹ gamma_ivl a ⊆ gamma_ivl b"
proof (cases a; cases b)
  fix l1 u1 l2 u2 :: eint
  assume "a = Ivl l1 u1" "b = Ivl l2 u2" and ab: "a ≤ b"
  from ab have le: "eint_le l2 l1" "eint_le u1 u2"
    unfolding less_eq_ivl_def ‹a = Ivl l1 u1› ‹b = Ivl l2 u2› by simp_all
  show "gamma_ivl a ⊆ gamma_ivl b"
    unfolding ‹a = Ivl l1 u1› ‹b = Ivl l2 u2›
    using le by (auto intro: eint_le_trans)
qed

lemma ivl_le_top: "(x::ivl) ≤ ivl_top"
  by (cases x) (simp add: less_eq_ivl_def ivl_top_def)
```

Join

The join takes the smaller lower and the larger upper bound; the lemmas prove it is the least upper bound and instantiate `semilattice_sup`.

```
fun join_ivl :: "ivl => ivl => ivl" where
    "join_ivl (Ivl l1 u1) (Ivl l2 u2) =
```

```
      Ivl (if l1 ≤ l2 then l1 else l2)
          (if u2 ≤ u1 then u1 else u2)"

lemma join_ivl_le_ub1: "(a :: ivl) ≤ join_ivl a b"
proof (cases a; cases b)
  fix l1 u1 l2 u2 :: eint
  assume "a = Ivl l1 u1" "b = Ivl l2 u2"
  then show "a ≤ join_ivl a b"
    unfolding less_eq_ivl_def
    using eint_le_linear[of l1 l2] eint_le_linear[of u1 u2]
    by auto
qed

lemma join_ivl_le_ub2: "(b :: ivl) ≤ join_ivl a b"
proof (cases a; cases b)
  fix l1 u1 l2 u2 :: eint
  assume "a = Ivl l1 u1" "b = Ivl l2 u2"
  then show "b ≤ join_ivl a b"
    unfolding less_eq_ivl_def
    using eint_le_linear[of l1 l2] eint_le_linear[of u1 u2]
    by auto
qed

lemma join_ivl_le_least:
  "(a :: ivl) ≤ c ⟹ b ≤ c ⟹ join_ivl a b ≤ c"
proof (cases a; cases b; cases c)
  fix l1 u1 l2 u2 l3 u3 :: eint
  assume "a = Ivl l1 u1" "b = Ivl l2 u2" "c = Ivl l3 u3"
    and "a ≤ c" "b ≤ c"
  then show "join_ivl a b ≤ c"
    unfolding less_eq_ivl_def by auto
qed

instantiation ivl :: sup begin
definition sup_ivl :: "ivl => ivl => ivl" where
  "sup_ivl = join_ivl"
instance ..
end

instance ivl :: semilattice_sup
proof intro_classes
  fix x y z :: ivl
  show "x ≤ x ⊔ y" unfolding sup_ivl_def by (rule join_ivl_le_ub1)
  show "y ≤ x ⊔ y" unfolding sup_ivl_def by (rule join_ivl_le_ub2)
  show "y ≤ x ⟹ z ≤ x ⟹ y ⊔ z ≤ x"
    unfolding sup_ivl_def by (rule join_ivl_le_least)
qed

instance ivl :: bounded_semilattice_sup_bot ..
```

Meet

Interval intersection. Used by `bfilter_ivl` to refine a variable's interval on a guard (`x < n` narrows the upper bound).

```
instantiation ivl :: inf begin
fun inf_ivl :: "ivl => ivl => ivl" where
    "inf_ivl (Ivl l1 u1) (Ivl l2 u2) =
       Ivl (if l2 ≤ l1 then l1 else l2)
           (if u1 ≤ u2 then u1 else u2)"
instance ..
end

lemma meet_ivl_gamma:
  "n ∈ gamma_ivl a ⟹ n ∈ gamma_ivl b ⟹ n ∈ gamma_ivl (a ⊓ b)"
  by (cases a; cases b; auto split: if_splits intro: eint_le_trans)

lemma gamma_inf_ivl [simp]:
  "gamma_ivl (a ⊓ b) = gamma_ivl a ∩ gamma_ivl b"
  apply (cases a; cases b; simp split: if_splits )
  using eint_le_linear eint_le_trans by blast+

lemma meet_ivl_le_lb1: "(a :: ivl) ⊓ b ≤ a"
proof (cases a; cases b)
  fix l1 u1 l2 u2 :: eint
  assume h: "a = Ivl l1 u1" "b = Ivl l2 u2"
```

```
  show "a ⊓ b ≤ a"
    using h eint_le_linear[of l2 l1] eint_le_linear[of u1 u2]
    by (auto simp: less_eq_ivl_def split: if_splits)
qed

lemma meet_ivl_le_lb2: "(a :: ivl) ⊓ b ≤ b"
proof (cases a; cases b)
  fix l1 u1 l2 u2 :: eint
  assume h: "a = Ivl l1 u1" "b = Ivl l2 u2"
  show "a ⊓ b ≤ b"
    using h eint_le_linear[of l1 l2] eint_le_linear[of u1 u2]
    by (auto simp: less_eq_ivl_def split: if_splits)
qed

lemma meet_ivl_greatest: "(a :: ivl) ≤ c ⟹ a ≤ b ⟹ a ≤ c ⊓ b"
proof (cases a; cases b; cases c)
  fix l1 u1 l2 u2 l3 u3 :: eint
  assume h: "a = Ivl l1 u1" "b = Ivl l2 u2" "c = Ivl l3 u3" "a ≤ c" "a ≤ b"
  show "a ≤ c ⊓ b"
    using h by (auto simp: less_eq_ivl_def split: if_splits intro: eint_le_trans)
qed

instance ivl :: semilattice_inf
proof intro_classes
  fix x y z :: ivl
  show "x ⊓ y ≤ x" by (rule meet_ivl_le_lb1)
  show "x ⊓ y ≤ y" by (rule meet_ivl_le_lb2)
  show "x ≤ y ⟹ x ≤ z ⟹ x ≤ y ⊓ z" by (rule meet_ivl_greatest)
qed

instance ivl :: lattice ..
instance ivl :: bounded_lattice_bot ..
```

Executable interface

Unlike `bot`, `top` has no other representation: `gamma_ivl (Ivl l u) = UNIV` forces `l ≤ Fin n` and `Fin n ≤ u` for every `n`, which pins `l = MinInf` and `u = PlusInf` exactly — so a direct equality test against `ivl_top` is already exact, no bound comparison needed.

```
definition is_top_ivl :: "ivl ⇒ bool" where
  "is_top_ivl i = (i = ivl_top)"

lemma is_top_ivl_correct_gamma: "is_top_ivl i ⟷ gamma_ivl i = UNIV"
  by (cases i; case_tac x1; case_tac x2)
     (auto simp add: is_top_ivl_def ivl_top_def set_eq_iff; presburger)+
```

`bot` is one fixed empty interval (`Ivl PlusInf MinInf`), but the $\leq$ instance's bound comparison makes `x ≤ bot` true only for that exact representation: any other inverted bound pair (e.g. a guard refinement narrowing a variable against its own infeasible range, `Ivl (Fin 5) (Fin (- 1))`) is just as empty by `gamma_ivl`, yet not `≤ bot` — the instance never normalizes. `is_bottom_ivl` tests emptiness directly against the bounds instead of against one particular representative.

Bound comparison alone is not quite enough, though: `Ivl MinInf MinInf` has `l ≤ u` (reflexively), yet no integer is "at" `MinInf` itself, so its `gamma_ivl` is still empty (symmetrically for `Ivl PlusInf PlusInf`). `l = PlusInf` and `u = MinInf` are exactly the two cases where `eint_le` can hold reflexively without any integer ever satisfying it, so both are called out explicitly alongside the general inverted-bounds case.

```
definition is_bottom_ivl :: "ivl ⇒ bool" where
  "is_bottom_ivl i =
     (case i of Ivl l u ⇒ l = PlusInf ∨ u = MinInf ∨ ¬ l ≤ u)"

lemma is_bottom_ivl_correct: "is_bottom_ivl i ⟷ gamma_ivl i = {}"
proof (cases i)
  case (Ivl l u)
  show ?thesis
    unfolding Ivl is_bottom_ivl_def
    by (cases l; cases u) auto
qed
```

Goblint shows an interval as `[l,u]`, an unconstrained end as the machine integer's limit in decimal; these bounds are unbounded, so it prints as ∞. Every empty representation prints as $\bot$, not only $\bot$. A standalone interval prints its top as $\top$, as Goblint's integer-domain lifter does, while a product component keeps the bounds.

```
fun string_of_eint :: "eint ⇒ String.literal" where
    "string_of_eint MinInf  = STR ''-'' + sym_infinity"
```

```
  | "string_of_eint PlusInf = STR ''+'' + sym_infinity"
  | "string_of_eint (Fin n) = string_of_int n"

fun string_of_ivl :: "ivl ⇒ String.literal" where
  "string_of_ivl (Ivl l u) =
     (if is_bottom_ivl (Ivl l u) then sym_bottom
      else STR ''['' + string_of_eint l + STR '','' + string_of_eint u + STR '']'')"
```

Canonical representatives

Canonical empty interval. The raw `ivl` lattice has infinitely many empty representations (any `Ivl l u` with `l > u`, or an infinite bound on the wrong side). `normalize_ivl` collapses every empty interval to the single $\bot$ representative `Ivl PlusInf MinInf`, leaving non-empty intervals untouched.

Arithmetic feeds its operands through `normalize_ivl` first. Since the pointwise sum/difference of $\bot$ with anything is again $\bot$, this keeps every empty result canonical. Without it, `[1,0] + [1,1] = [2,1]` would manufacture ever-new empty representations, which as context keys defeat fixpoint convergence in the recursive interval example.

```
definition normalize_ivl :: "ivl ⇒ ivl" where
  "normalize_ivl v =
     (case v of Ivl l u ⇒
        if l ≤ u ∧ l ≠ PlusInf ∧ u ≠ MinInf then v else bot)"

lemma normalize_ivl_gamma: "gamma_ivl (normalize_ivl v) = gamma_ivl v"
  by (cases v) (auto simp: normalize_ivl_def bot_ivl_def
        split: eint.splits if_splits intro: eint_le_trans)


lemma normalize_ivl_mono: "x ≤ y ⟹ normalize_ivl x ≤ normalize_ivl y"
  by (cases x; cases y)
     (auto simp: normalize_ivl_def less_eq_ivl_def bot_ivl_def
        eint_le_PlusInf_iff eint_le_MinInf_iff
        split: eint.splits if_splits intro: eint_le_trans)
```

Emptiness test on intervals. The raw `ivl` lattice contains many unnormalised empty intervals (any `Ivl l u` with `l > u`, including $\bot$). `ivl_nonempty` captures exactly the intervals with non-empty concretization (proved below: implied by any witness, and monotone along $\leq$); multiplication uses it for proper bottom handling, which is what makes the precise corner product monotone.

```
fun ivl_nonempty :: "ivl => bool" where
  "ivl_nonempty (Ivl l u) = (l ≤ u ∧ l ≠ PlusInf ∧ u ≠ MinInf)"

lemma gamma_ivl_nonempty: "i ∈ gamma_ivl v ⟹ ivl_nonempty v"
  by (cases v; cases "case v of Ivl l _ => l"; cases "case v of Ivl _ u => u"; auto)

lemma ivl_nonempty_mono: "ivl_nonempty a ⟹ a ≤ b ⟹ ivl_nonempty b"
  by (cases a; cases b; auto simp: less_eq_ivl_def intro: eint_le_trans
        elim: eint_le.elims)
```

`normalize_ivl` picks one representative per abstract value: non-empty intervals are already determined by their bounds, and every empty one collapses to $\bot$. This mirrors the component-local normalisation each Goblint integer domain performs for itself (`Interval.norm` sending `l > u` to `None`, `Congruence.normalize` pinning the residue and modulus), which is what makes a structural stability test on a representation meaningful.

`is_bottom_ivl` stays an exact test on arbitrary bounds rather than an equality against $\bot$: `ivl` is an ordinary datatype, so `Ivl (Fin 5) (Fin 3)` remains writable by any caller and must still be recognised as empty.

```
lemma is_bottom_ivl_bot [simp]: "is_bottom_ivl bot"
  by (simp add: is_bottom_ivl_def bot_ivl_def)

lemma normalize_ivl_bot_case [simp]:
  "is_bottom_ivl v ⟹ normalize_ivl v = bot"
  by (cases v) (auto simp: normalize_ivl_def is_bottom_ivl_def)

lemma normalize_ivl_id [simp]:
  "¬ is_bottom_ivl v ⟹ normalize_ivl v = v"
  by (cases v) (simp add: normalize_ivl_def is_bottom_ivl_def)

lemma normalize_ivl_bot [simp]: "normalize_ivl bot = bot"
  by simp

lemma normalize_ivl_idem [simp]:
  "normalize_ivl (normalize_ivl v) = normalize_ivl v"
  by (cases "is_bottom_ivl v") simp_all
```

```
lemma normalize_ivl_le: "normalize_ivl v ≤ v"
  by (cases "is_bottom_ivl v") simp_all

lemma is_bottom_ivl_normalize [simp]:
  "is_bottom_ivl (normalize_ivl v) = is_bottom_ivl v"
  by (cases "is_bottom_ivl v") simp_all

definition mk_ivl :: "eint ⇒ eint ⇒ ivl" where
  [simp]: "mk_ivl l u = normalize_ivl (Ivl l u)"
```

`mk_ivl` is the canonical constructor: any operation that computes new bounds should build its result through it instead of applying `Ivl` to possibly inverted bounds and leaving the emptiness implicit. Its defining equation is a simp rule, like (⊓)'s above, so a caller that already rewrites with `normalize_ivl`'s equations sees through it rather than meeting an opaque constant.

```
lemma gamma_mk_ivl [simp]: "gamma_ivl (mk_ivl l u) = gamma_ivl (Ivl l u)"
  by (simp add: normalize_ivl_gamma)

lemma normalize_ivl_mk_ivl [simp]: "normalize_ivl (mk_ivl l u) = mk_ivl l u"
  by simp

lemma mk_ivl_mono:
  "Ivl l1 u1 ≤ Ivl l2 u2 ⟹ mk_ivl l1 u1 ≤ mk_ivl l2 u2"
  by (simp add: normalize_ivl_mono)
```

(⊓) is the one operation that is deliberately left un-normalised: it intersects the bounds and lets an infeasible intersection come out as an inverted pair.

```
lemma meet_ivl_not_canonical:
  "normalize_ivl (Ivl (Fin 0) (Fin 3) ⊓ Ivl (Fin 5) (Fin 9))
     ≠ Ivl (Fin 0) (Fin 3) ⊓ Ivl (Fin 5) (Fin 9)"
  by (simp add: normalize_ivl_def bot_ivl_def)
```

Normalising it instead would break `semilattice_inf`: the greatest-lower-bound law `meet_ivl_greatest` quantifies over *all* lower bounds, including the un-normalised empty ones a caller may write down, and those are not below ⊥. The three facts below are exactly that counterexample – `Ivl (Fin 5) (Fin 3)` is a lower bound of both operands, yet it is not below the normalised meet. Canonicity therefore belongs to the operations that build new values (arithmetic, `mk_ivl`), not to the order-theoretic (⊓).

```
lemma meet_ivl_normalized_breaks_greatest:
  "Ivl (Fin 5) (Fin 3) ≤ Ivl (Fin 0) (Fin 3)"
  "Ivl (Fin 5) (Fin 3) ≤ Ivl (Fin 5) (Fin 9)"
  "¬ Ivl (Fin 5) (Fin 3)
       ≤ normalize_ivl (Ivl (Fin 0) (Fin 3) ⊓ Ivl (Fin 5) (Fin 9))"
  by (simp_all add: less_eq_ivl_def normalize_ivl_def bot_ivl_def)
```

Semantic intersection

`intersect_ivl` is the intersection a backward filter uses, distinct from the order's `inf` on this raw carrier.

```
definition intersect_ivl :: "ivl ⇒ ivl ⇒ ivl" where
  [simp]: "intersect_ivl a b = normalize_ivl (a ⊓ b)"
```

(⊓) is the greatest lower bound of the representation order; the operation a backward filter actually wants is the intersection of the concretizations, which is a weaker requirement — it need only preserve `gamma_ivl` and be reductive and monotone, and is free to pick any representative of the result. `intersect_ivl` is that operation: it normalizes the bound intersection so an infeasible result becomes ⊥ rather than a reversed pair.

Goblint's interval `meet` is simultaneously both, because its carrier keeps only canonical inhabited values — `norm` maps `l > u` to `None`, so no element corresponding to `Ivl (Fin 5) (Fin 3)` exists there to be a common lower bound. On this larger raw carrier the two notions come apart, which is what `meet_ivl_normalized_breaks_greatest` records.

```
lemma gamma_intersect_ivl [simp]:
  "gamma_ivl (intersect_ivl a b) = gamma_ivl (a ⊓ b)"
  by (simp add: normalize_ivl_gamma)

lemma intersect_ivl_gamma:
  "n ∈ gamma_ivl a ⟹ n ∈ gamma_ivl b ⟹ n ∈ gamma_ivl (intersect_ivl a b)"
  using meet_ivl_gamma[of n a b] by (simp add: normalize_ivl_gamma)

lemma intersect_ivl_le1: "intersect_ivl a b ≤ a"
proof -
  have "normalize_ivl (a ⊓ b) ≤ a ⊓ b" by (rule normalize_ivl_le)
  also have "a ⊓ b ≤ a" using meet_ivl_le_lb1[of a b] by simp
```

```
  finally show ?thesis by simp
qed

lemma intersect_ivl_le2: "intersect_ivl a b ≤ b"
proof -
  have "normalize_ivl (a ⊓ b) ≤ a ⊓ b" by (rule normalize_ivl_le)
  also have "a ⊓ b ≤ b" using meet_ivl_le_lb2[of a b] by simp
  finally show ?thesis by simp
qed

lemma intersect_ivl_mono:
  assumes "a1 ≤ a2" and "b1 ≤ b2"
  shows "intersect_ivl a1 b1 ≤ intersect_ivl a2 b2"
proof -
  have "a1 ⊓ b1 ≤ a2 ⊓ b2" using inf_mono[OF assms] by simp
  then show ?thesis by (simp add: normalize_ivl_mono)
qed

lemma normalize_ivl_intersect_ivl [simp]:
  "normalize_ivl (intersect_ivl a b) = intersect_ivl a b"
  by simp

end
```

2.14 Sign_Lattice

```
theory Sign_Lattice
  imports "Voblint_Domain.State_Concretization"
begin
```

Sign lattice

sign abstracts integers by their sign: Bot – empty (unreachable / undefined) Neg – strictly negative n | n < 0 NonPos – non-positive {n | n ≤ 0} Zero – exactly zero 0 NonNeg – non-negative {n | n ≥ 0} Pos – strictly positive n | n > 0 Top – all integers UNIV

Seven-element lattice (Bot ≤ Neg,Zero,Pos ≤ NonPos,NonNeg ≤ Top). Finite; no widening needed.

Carrier and concretization

```
datatype sign = SBot | SNeg | SNonPos | SZero | SNonNeg | SPos | STop

fun gamma_sign :: "sign => int set" where
    "gamma_sign SBot    = {}"
  | "gamma_sign SNeg    = {n. n < 0}"
  | "gamma_sign SNonPos = {n. n ≤ 0}"
  | "gamma_sign SZero   = {0}"
  | "gamma_sign SNonNeg = {n. n ≥ 0}"
  | "gamma_sign SPos    = {n. n > 0}"
  | "gamma_sign STop    = UNIV"

fun sign_of_int :: "int => sign" where
  "sign_of_int n = (if n < 0 then SNeg else if n = 0 then SZero else SPos)"

lemma sign_of_int_gamma: "n : gamma_sign (sign_of_int n)"
  by (auto split: if_splits)
```

Order

sign_le puts SNeg and SZero below SNonPos, and SZero and SPos below SNonNeg; concretization is monotone in it.

```
fun sign_le :: "sign => sign => bool" where
    "sign_le SBot    _       = True"
  | "sign_le _       STop    = True"
  | "sign_le SNeg    SNeg    = True"
  | "sign_le SNeg    SNonPos = True"
  | "sign_le SNonPos SNonPos = True"
  | "sign_le SZero   SZero   = True"
  | "sign_le SZero   SNonPos = True"
  | "sign_le SZero   SNonNeg = True"
  | "sign_le SNonNeg SNonNeg = True"
  | "sign_le SPos    SPos    = True"
  | "sign_le SPos    SNonNeg = True"
  | "sign_le _       _       = False"

lemma sign_le_refl:    "sign_le s s"                              by (cases s) simp_all

lemma sign_le_antisym:
  assumes st: "sign_le s t" and ts: "sign_le t s"
  shows "s = t"
  using st ts by (cases s; cases t; simp)

lemma sign_le_trans:
  assumes st: "sign_le s t" and tu: "sign_le t u"
  shows "sign_le s u"
  using st tu by (cases s; cases t; cases u; simp)

lemma gamma_sign_mono:
  assumes st: "sign_le s t"
  shows "gamma_sign s <= gamma_sign t"
  using st by (cases s; cases t; auto)

instantiation sign :: ord begin
definition less_eq_sign :: "sign => sign => bool" where "(a::sign) <= b = sign_le a b"
definition less_sign    :: "sign => sign => bool" where
  "(a::sign) <  b = (sign_le a b ∧ ¬ sign_le b a)"
instance ..
end
```

```
instance sign :: preorder
proof intro_classes
  fix x y z :: sign
  show "(x < y) = (x ≤ y ∧ ¬ y ≤ x)"
    unfolding less_sign_def less_eq_sign_def by simp
  show "x ≤ x"
    by (simp add: less_eq_sign_def sign_le_refl)
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
    unfolding less_eq_sign_def by (rule sign_le_trans)
qed

instantiation sign :: order begin
instance proof intro_classes
  fix x y :: sign
  assume "x ≤ y" "y ≤ x"
  then show "x = y"
    unfolding less_eq_sign_def by (blast intro: sign_le_antisym)
qed
end
```

The bottom instance lifts pointwise to abstract states and enables monotone least-upper-bound iteration.

```
instantiation sign :: bot begin
definition "bot_sign = SBot"
instance ..
end

instantiation sign :: order_bot begin
instance proof intro_classes
  fix x :: sign
  show "bot ≤ x"
    unfolding less_eq_sign_def bot_sign_def by simp
qed
end

instantiation sign :: top begin
definition "top_sign = STop"
instance ..
end

instantiation sign :: order_top begin
instance proof intro_classes
  fix x :: sign
  show "x ≤ top"
    unfolding less_eq_sign_def top_sign_def by (cases x) simp_all
qed
end

lemma gamma_sign_top: "gamma_sign top = UNIV"
  unfolding top_sign_def by simp
```

Join

`join_sign` returns the least sign covering both arguments, for example `SNeg` and `SZero` join to `SNonPos`.

```
fun join_sign :: "sign => sign => sign" where
    "join_sign SBot    b       = b"
  | "join_sign a       SBot    = a"
  | "join_sign STop    _       = STop"
  | "join_sign _       STop    = STop"
  | "join_sign SNeg    SNeg    = SNeg"
  | "join_sign SNeg    SZero   = SNonPos"
  | "join_sign SNeg    SNonPos = SNonPos"
  | "join_sign SZero   SNeg    = SNonPos"
  | "join_sign SZero   SZero   = SZero"
  | "join_sign SZero   SPos    = SNonNeg"
  | "join_sign SZero   SNonPos = SNonPos"
  | "join_sign SZero   SNonNeg = SNonNeg"
  | "join_sign SNonPos SNeg    = SNonPos"
  | "join_sign SNonPos SZero   = SNonPos"
  | "join_sign SNonPos SNonPos = SNonPos"
  | "join_sign SNonNeg SZero   = SNonNeg"
  | "join_sign SNonNeg SPos    = SNonNeg"
  | "join_sign SNonNeg SNonNeg = SNonNeg"
  | "join_sign SPos    SZero   = SNonNeg"
  | "join_sign SPos    SNonNeg = SNonNeg"
```

```
  | "join_sign SPos    SPos    = SPos"
  | "join_sign _       _       = STop"

lemma join_sign_ub1: "sign_le a (join_sign a b)"
  by (cases a; cases b; simp)

lemma join_sign_ub2: "sign_le b (join_sign a b)"
  by (cases a; cases b; simp)

lemma join_sign_least: "sign_le a csg ⟹ sign_le b csg ⟹ sign_le (join_sign a b) csg"
  by (cases a; cases b; cases csg; simp)

instantiation sign :: sup begin
definition sup_sign :: "sign => sign => sign" where
  "sup_sign = join_sign"
instance ..
end

instance sign :: semilattice_sup
proof intro_classes
  fix x y z :: sign
  show "x ≤ x ⊔ y"
    unfolding sup_sign_def less_eq_sign_def by (rule join_sign_ub1)
  show "y ≤ x ⊔ y"
    unfolding sup_sign_def less_eq_sign_def by (rule join_sign_ub2)
  show "y ≤ x ⟹ z ≤ x ⟹ y ⊔ z ≤ x"
    unfolding sup_sign_def less_eq_sign_def by (rule join_sign_least)
qed
```

`sign` is already an `order_bot` and a `semilattice_sup`, so the bounded semilattice comes for free.

```
instance sign :: bounded_semilattice_sup_bot ..
```

Meet

The sign meet is exact: it concretizes to the intersection of its operands. As a `semilattice_inf` instance it gives `inf_mono` for free, which the monotonicity proof of `bfilter` needs.

```
instantiation sign :: inf begin

fun inf_sign :: "sign => sign => sign" where
    "inf_sign SBot    _       = SBot"
  | "inf_sign _       SBot    = SBot"
  | "inf_sign STop    b       = b"
  | "inf_sign a       STop    = a"
  | "inf_sign SNeg    SNeg    = SNeg"
  | "inf_sign SNeg    SNonPos = SNeg"
  | "inf_sign SNonPos SNeg    = SNeg"
  | "inf_sign SNonPos SNonPos = SNonPos"
  | "inf_sign SNonPos SZero   = SZero"
  | "inf_sign SZero   SNonPos = SZero"
  | "inf_sign SNonPos SNonNeg = SZero"
  | "inf_sign SNonNeg SNonPos = SZero"
  | "inf_sign SZero   SZero   = SZero"
  | "inf_sign SZero   SNonNeg = SZero"
  | "inf_sign SNonNeg SZero   = SZero"
  | "inf_sign SNonNeg SNonNeg = SNonNeg"
  | "inf_sign SNonNeg SPos    = SPos"
  | "inf_sign SPos    SNonNeg = SPos"
  | "inf_sign SPos    SPos    = SPos"
  | "inf_sign _       _       = SBot"

instance ..

end

lemma gamma_inf_sign [simp]:
  "gamma_sign (a ⊓ b) = gamma_sign a ∩ gamma_sign b"
  by (cases a; cases b) auto

lemma inf_sign_sound:
  "n ∈ gamma_sign a ⟹ n ∈ gamma_sign b ⟹ n ∈ gamma_sign (a ⊓ b)"
  by simp

instance sign :: semilattice_inf
proof intro_classes
  fix x y z :: sign
```

```
  show "x ⊓ y ≤ x"
    by (cases x; cases y; auto simp: less_eq_sign_def)
  show "x ⊓ y ≤ y"
    by (cases x; cases y; auto simp: less_eq_sign_def)
  show "x ≤ y ⟹ x ≤ z ⟹ x ≤ y ⊓ z"
    by (cases x; cases y; cases z; auto simp: less_eq_sign_def)
qed

instance sign :: lattice ..
instance sign :: bounded_lattice_bot ..
```

Executable interface

`SBot` is the only empty value a finite enumerated domain can have (every other constructor denotes a nonempty set of integers), so a direct equality test is already exact — unlike Interval's analogous fact, which cannot use equality against one representative because Interval's bound-pair representation has many empty values besides its canonical `bot`. Exposed here, at the domain's own theory, rather than inlined where a caller happens to need it, so every consumer (not just one) gets the same domain-owned fact.

```
definition is_bottom_sign :: "sign ⇒ bool" where
  "is_bottom_sign s = (s = SBot)"

lemma is_bottom_sign_correct: "is_bottom_sign s ⟷ gamma_sign s = {}"
  unfolding is_bottom_sign_def
  by (cases s) (auto intro: exI[of _ "-1"] exI[of _ "0"] exI[of _ "1"])
```

`STop` is likewise the unique top of a finite enumeration.

```
definition is_top_sign :: "sign ⇒ bool" where
  "is_top_sign s = (s = STop)"

lemma is_top_sign_correct_gamma: "is_top_sign s ⟷ gamma_sign s = UNIV"
  by(cases s) (auto simp: is_top_sign_def set_eq_iff; presburger)+
```

Goblint has no sign value domain; its tutorial sign analysis prints `-`, `0` and `+`, which the two non-strict elements extend as $\leq$`0` and $\geq$`0`.

```
fun string_of_sign :: "sign ⇒ String.literal" where
    "string_of_sign SBot    = sym_bottom"
  | "string_of_sign SNeg    = STR ''-''"
  | "string_of_sign SNonPos = sym_le + STR ''0''"
  | "string_of_sign SZero   = STR ''0''"
  | "string_of_sign SNonNeg = sym_ge + STR ''0''"
  | "string_of_sign SPos    = STR ''+''"
  | "string_of_sign STop    = sym_top"

end
```

2.15 Parity_Lattice

```
theory Parity_Lattice
  imports "Voblint_Domain.State_Concretization"
begin
```

Parity lattice

parity abstracts integers by their parity: Bot – empty (unreachable) Even – n | even n Odd – n | odd n Top – all integers

Four-element lattice (`Bot` $\leq$ `Even,Odd` $\leq$ `Top`). Finite; widen = sup. This analysis does not implement backward guard refinement. Its branch transfer therefore preserves parity information from the incoming state.

Carrier and concretization

```
datatype parity = PBot | PEven | POdd | PTop

fun gamma_parity :: "parity => int set" where
    "gamma_parity PBot  = {}"
  | "gamma_parity PEven = {n. even n}"
  | "gamma_parity POdd  = {n. odd n}"
  | "gamma_parity PTop  = UNIV"

fun parity_of_int :: "int => parity" where
  "parity_of_int n = (if even n then PEven else POdd)"

lemma parity_of_int_gamma: "n ∈ gamma_parity (parity_of_int n)" by auto
```

Order

The flat order with `PBot` below and `PTop` above the two parities; `gamma_parity_mono` shows concretization respects it.

```
fun parity_le :: "parity => parity => bool" where
    "parity_le PBot  _     = True"
  | "parity_le _     PTop  = True"
  | "parity_le PEven PEven = True"
  | "parity_le POdd  POdd  = True"
  | "parity_le _     _     = False"

lemma parity_le_refl: "parity_le s s" by (cases s) simp_all

lemma parity_le_antisym: "parity_le s t ⟹ parity_le t s ⟹ s = t"
  by (cases s; cases t; simp)

lemma parity_le_trans: "parity_le s t ⟹ parity_le t u ⟹ parity_le s u"
  by (cases s; cases t; cases u; simp)

lemma gamma_parity_mono: "parity_le s t ⟹ gamma_parity s ⊆ gamma_parity t"
  by (cases s; cases t; auto)

instantiation parity :: ord begin
definition less_eq_parity :: "parity => parity => bool" where "(a::parity) ≤ b = parity_le a b"
definition less_parity    :: "parity => parity => bool" where
  "(a::parity) <  b = (parity_le a b ∧ ¬ parity_le b a)"
instance ..
end

instance parity :: preorder
proof
  fix x y z :: parity
  show "(x < y) = (x ≤ y ∧ ¬ y ≤ x)" unfolding less_parity_def less_eq_parity_def by simp
  show "x ≤ x" by (simp add: less_eq_parity_def parity_le_refl)
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z" unfolding less_eq_parity_def by (rule parity_le_trans)
qed

instantiation parity :: order begin
instance proof
  fix x y :: parity
  assume "x ≤ y" "y ≤ x"
  then show "x = y" unfolding less_eq_parity_def by (blast intro: parity_le_antisym)
qed
end
```

```
instantiation parity :: bot begin
definition "bot_parity = PBot"
instance ..
end

instantiation parity :: order_bot begin
instance proof
  fix x :: parity
  show "bot ≤ x" unfolding less_eq_parity_def bot_parity_def by simp
qed
end

instantiation parity :: top begin
definition "top_parity = PTop"
instance ..
end

instantiation parity :: order_top begin
instance proof intro_classes
  fix x :: parity
  show "x ≤ top"
    unfolding less_eq_parity_def top_parity_def by (cases x) simp_all
qed
end

lemma gamma_parity_top: "gamma_parity top = UNIV"
  unfolding top_parity_def by simp
```

Join

Joining two different parities loses all information and yields `PTop`. The lemmas prove `join_parity` is the least upper bound.

```
fun join_parity :: "parity => parity => parity" where
    "join_parity PBot  b     = b"
  | "join_parity a     PBot  = a"
  | "join_parity PTop  _     = PTop"
  | "join_parity _     PTop  = PTop"
  | "join_parity PEven PEven = PEven"
  | "join_parity POdd  POdd  = POdd"
  | "join_parity _     _     = PTop"

lemma join_parity_ub1: "parity_le a (join_parity a b)" by (cases a; cases b; simp)
lemma join_parity_ub2: "parity_le b (join_parity a b)" by (cases a; cases b; simp)
lemma join_parity_least: "parity_le a c ⟹ parity_le b c ⟹ parity_le (join_parity a b) c"
  by (cases a; cases b; cases c; simp)

instantiation parity :: sup begin
definition sup_parity :: "parity => parity => parity" where "sup_parity = join_parity"
instance ..
end

instance parity :: semilattice_sup
proof
  fix x y z :: parity
  show "x ≤ x ⊔ y" unfolding sup_parity_def less_eq_parity_def by (rule join_parity_ub1)
  show "y ≤ x ⊔ y" unfolding sup_parity_def less_eq_parity_def by (rule join_parity_ub2)
  show "y ≤ x ⟹ z ≤ x ⟹ y ⊔ z ≤ x"
    unfolding sup_parity_def less_eq_parity_def by (rule join_parity_least)
qed

instance parity :: bounded_semilattice_sup_bot ..
```

Meet

Parity is closed under intersection: the meet of two classes is their common class, or bottom when one is even and the other odd.

```
instantiation parity :: inf begin

fun inf_parity :: "parity => parity => parity" where
    "inf_parity PTop b    = b"
  | "inf_parity a    PTop = a"
  | "inf_parity a    b    = (if a = b then a else PBot)"

instance ..
```

```
end

lemma gamma_inf_parity [simp]:
  "gamma_parity (a ⊓ b) = gamma_parity a ∩ gamma_parity b"
  by (cases a; cases b; auto)

instance parity :: semilattice_inf
proof intro_classes
  fix x y z :: parity
  show "x ⊓ y ≤ x"
    by (cases x; cases y; simp add: less_eq_parity_def)
  show "x ⊓ y ≤ y"
    by (cases x; cases y; simp add: less_eq_parity_def)
  show "x ≤ y ⟹ x ≤ z ⟹ x ≤ y ⊓ z"
    by (cases x; cases y; cases z; simp add: less_eq_parity_def)
qed

instance parity :: lattice ..
instance parity :: bounded_lattice_bot ..
```

Executable interface

`PBot` is the only value with empty concretization, so a direct equality test decides emptiness exactly. Finiteness of the carrier is not what settles this — a finite domain may perfectly well have two empty values. What settles it is the four concretizations themselves: the other three are each inhabited.

```
definition is_bottom_parity :: "parity ⇒ bool" where
  "is_bottom_parity p = (p = PBot)"

lemma is_bottom_parity_correct: "is_bottom_parity p ⟷ gamma_parity p = {}"
  unfolding is_bottom_parity_def
  by (cases p) (auto intro: exI[of _ "0"] exI[of _ "1"])
```

`PTop` is the unique top of a finite enumeration, the same reasoning as Sign's `is_top_sign`.

```
definition is_top_parity :: "parity ⇒ bool" where
  "is_top_parity p = (p = PTop)"

lemma is_top_parity_correct_gamma: "is_top_parity p ⟷ gamma_parity p = UNIV"
  by(cases p) (auto simp: is_top_parity_def set_eq_iff; presburger)+
```

Goblint has no parity domain. A parity is the congruence class modulo 2, so it prints in Goblint's congruence notation: $2\mathbb{Z}$, $1+2\mathbb{Z}$, and $\mathbb{Z}$ for a product component's top. A standalone parity prints its top as $\top$.

```
fun string_of_parity :: "parity ⇒ String.literal" where
    "string_of_parity PBot  = sym_bottom"
  | "string_of_parity PEven = STR ''2'' + sym_int"
  | "string_of_parity POdd  = STR ''1+2'' + sym_int"
  | "string_of_parity PTop  = sym_int"

end
```

2.16 Congruence_Lattice

```
theory Congruence_Lattice
  imports "Voblint_Domain.State_Concretization" "HOL-Computational_Algebra.Euclidean_Algorithm"
begin
```

Congruence lattice

Goblint's congruence domain as a normalized subtype of `(int * int) option`, with concretization `gamma_congruence`, the lattice order, join and the executable tests the `numeric_domain` instance needs.

Carrier and concretization

The representation follows Goblint's congruence domain: bottom is `None` and `Some (c, m)` denotes the class of integers congruent to `c` modulo `m`. Modulus zero denotes the singleton `{c}`.

Isabelle's order laws quantify over every carrier value, so the public type is the subtype of normalized representations. Public construction applies the same normalization discipline as Goblint: a nonzero modulus becomes positive and its residue lies in the half-open range from zero to the modulus.

```
type_synonym congruence_rep = "(int * int) option"

fun normalized_congruence_rep :: "congruence_rep => bool" where
  "normalized_congruence_rep None = True"
| "normalized_congruence_rep (Some (c, m)) =
     (m = 0 ∨ (0 <= c ∧ c < m))"

fun normalize_congruence_rep :: "congruence_rep => congruence_rep" where
  "normalize_congruence_rep None = None"
| "normalize_congruence_rep (Some (c, m)) =
     (if m = 0 then Some (c, 0)
      else
        let p = abs m;
            r = c mod p
        in Some (r, p))"

lemma normalized_normalize_congruence_rep [simp]:
  "normalized_congruence_rep (normalize_congruence_rep x)"
proof (cases x)
  case None
  then show ?thesis by simp
next
  case (Some p)
  obtain c m where p: "p = (c, m)"
    by (cases p)
  show ?thesis
  proof (cases "m = 0")
    case True
    then show ?thesis unfolding Some p by simp
  next
    case False
    then have positive: "0 < abs m" by simp
    have nonnegative: "0 <= c mod abs m"
      using positive by simp
    have less: "c mod abs m < abs m"
      by (rule pos_mod_bound[OF positive])
    show ?thesis
      unfolding Some p
      using False nonnegative less
      by (simp add: Let_def)
  qed
qed

lemma normalize_congruence_rep_fixed:
  assumes "normalized_congruence_rep x"
  shows "normalize_congruence_rep x = x"
proof (cases x)
  case None
  then show ?thesis by simp
next
  case (Some p)
  obtain c m where p: "p = (c, m)"
    by (cases p)
  show ?thesis
  proof (cases "m = 0")
```

```
    case True
    then show ?thesis unfolding Some p by simp
  next
    case False
    with assms have bounds: "0 <= c" "c < m"
      unfolding Some p by simp_all
    then have positive: "0 < m" by linarith
    have "c mod m = c"
      by (rule mod_pos_pos_trivial[OF bounds])
    with positive False show ?thesis
      unfolding Some p by (simp add: Let_def abs_of_pos)
  qed
qed

lemma normalize_congruence_rep_idem [simp]:
  "normalize_congruence_rep (normalize_congruence_rep x) =
   normalize_congruence_rep x"
  by (rule normalize_congruence_rep_fixed) simp

lemma normalized_congruence_rep_modulus_nonnegative:
  assumes "normalized_congruence_rep (Some (c, m))"
  shows "0 <= m"
  using assms by auto

lemma normalized_congruence_rep_nonzero:
  assumes "normalized_congruence_rep (Some (c, m))"
      and "m ≠ 0"
  shows "0 <= c" and "c < m" and "0 < m"
  using assms by auto
typedef congruence =
  "{x :: congruence_rep. normalized_congruence_rep x}"
  morphisms Rep_congruence Abs_congruence
  by (rule exI[of _ None]) simp

setup_lifting type_definition_congruence

lemma normalized_Rep_congruence [simp]:
  "normalized_congruence_rep (Rep_congruence a)"
  using Rep_congruence[of a] by simp

lemma congruence_rep_cases [case_names Bot Class]:
  obtains (Bot) "Rep_congruence a = None"
    | (Class) c m where "Rep_congruence a = Some (c, m)"
proof (cases "Rep_congruence a")
  case (Some p)
  then show thesis by (cases p) (auto intro: Class)
qed (rule Bot)
instantiation congruence :: equal
begin

definition equal_congruence :: "congruence => congruence => bool" where
  "equal_congruence a b = (Rep_congruence a = Rep_congruence b)"

instance
proof
  fix a b :: congruence
  show "HOL.equal a b ⟷ a = b"
    unfolding equal_congruence_def
    by (simp only: Rep_congruence_inject)
qed

end

lift_definition mk_congruence :: "int => int => congruence"
  is "λc m. normalize_congruence_rep (Some (c, m))"
  by (rule normalized_normalize_congruence_rep)

lift_definition bottom_congruence :: congruence
  is None
  by simp

declare mk_congruence.rep_eq [simp] bottom_congruence.rep_eq [simp]

lemma congruence_cases [cases type: congruence]:
  obtains "a = bottom_congruence"
    | c m where "a = mk_congruence c m"
```

```
proof (cases "Rep_congruence a")
  case None
  have "Rep_congruence a = Rep_congruence bottom_congruence"
    using None by simp
  then have "a = bottom_congruence"
    by (simp only: Rep_congruence_inject)
  then show thesis by (rule that)
next
  case (Some p)
  obtain c m where p: "p = (c, m)"
    by (cases p)
  have normalized: "normalized_congruence_rep (Some (c, m))"
    using normalized_Rep_congruence[of a] unfolding Some p by simp
  then have fixed:
    "normalize_congruence_rep (Some (c, m)) = Some (c, m)"
    by (rule normalize_congruence_rep_fixed)
  have reps: "Rep_congruence a = Rep_congruence (mk_congruence c m)"
  proof -
    have "Rep_congruence a = Some (c, m)"
      using Some p by simp
    also have "... = Rep_congruence (mk_congruence c m)"
      using fixed by simp
    finally show ?thesis .
  qed
  then have "a = mk_congruence c m"
    by (simp only: Rep_congruence_inject)
  then show thesis by (rule that)
qed

fun gamma_congruence_rep :: "congruence_rep => int set" where
  "gamma_congruence_rep None = {}"
| "gamma_congruence_rep (Some (c, m)) = {n. m dvd n - c}"

definition gamma_congruence :: "congruence => int set" where
  "gamma_congruence a = gamma_congruence_rep (Rep_congruence a)"

lemma congruence_class_subset_iff:
  fixes m1 m2 c1 c2 :: int
  shows
    "{n. m1 dvd n - c1} ⊆ {n. m2 dvd n - c2} ⟷
     m2 dvd m1 ∧ m2 dvd c1 - c2"
proof
  assume subset:
    "{n. m1 dvd n - c1} ⊆ {n. m2 dvd n - c2}"
  have offset: "m2 dvd c1 - c2"
    using subsetD[OF subset, of c1] by simp
  have shifted: "m2 dvd (c1 + m1) - c2"
    using subsetD[OF subset, of "c1 + m1"] by simp
  have "m2 dvd ((c1 + m1) - c2) - (c1 - c2)"
    by (rule dvd_diff[OF shifted offset])
  then have "m2 dvd m1" by simp
  with offset show "m2 dvd m1 ∧ m2 dvd c1 - c2" by blast
next
  assume divisibility: "m2 dvd m1 ∧ m2 dvd c1 - c2"
  show "{n. m1 dvd n - c1} ⊆ {n. m2 dvd n - c2}"
  proof
    fix n
    assume "n ∈ {n. m1 dvd n - c1}"
    then have n: "m1 dvd n - c1" by simp
    have first: "m2 dvd n - c1"
      by (rule dvd_trans[OF divisibility[THEN conjunct1] n])
    have "m2 dvd (n - c1) + (c1 - c2)"
      by (rule dvd_add[OF first divisibility[THEN conjunct2]])
    then show "n ∈ {n. m2 dvd n - c2}" by simp
  qed
qed

lemma gamma_normalize_congruence_rep [simp]:
  "gamma_congruence_rep (normalize_congruence_rep x) =
   gamma_congruence_rep x"
proof (cases x)
  case None
  then show ?thesis by simp
next
  case (Some p)
  obtain c m where p: "p = (c, m)"
```

```
    by (cases p)
  show ?thesis
  proof (cases "m = 0")
    case True
    then show ?thesis unfolding Some p by simp
  next
    case False
    let ?modulus = "abs m"
    let ?residue = "c mod ?modulus"
    have modulus1: "m dvd ?modulus" by simp
    have modulus2: "?modulus dvd m" by simp
    have offset2: "?modulus dvd c - ?residue"
      by (rule dvd_minus_mod)
    have offset1: "m dvd c - ?residue"
      using offset2 by (simp only: abs_dvd_iff)
    have negated_offset: "m dvd -(c - ?residue)"
      using offset1 by (simp only: dvd_minus_iff)
    have offset1_reverse: "m dvd ?residue - c"
      using negated_offset by simp
    have classes:
      "{n. ?modulus dvd n - ?residue} = {n. m dvd n - c}"
    proof (rule Set.subset_antisym)
      show "{n. ?modulus dvd n - ?residue} ⊆
        {n. m dvd n - c}"
        unfolding congruence_class_subset_iff
        using modulus1 offset1_reverse by blast
      show "{n. m dvd n - c} ⊆
        {n. ?modulus dvd n - ?residue}"
        unfolding congruence_class_subset_iff
        using modulus2 offset2 by blast
    qed
    show ?thesis
      unfolding Some p
      using False classes by (simp add: Let_def)
  qed
qed

lemma gamma_congruence_rep_inject:
  assumes normalized_x: "normalized_congruence_rep x"
      and normalized_y: "normalized_congruence_rep y"
      and gamma_eq: "gamma_congruence_rep x = gamma_congruence_rep y"
  shows "x = y"
proof (cases x)
  case None
  show ?thesis
  proof (cases y)
    case None
    with ‘x = None‘ show ?thesis by simp
  next
    case (Some p)
    obtain c m where p: "p = (c, m)"
      by (cases p)
    have "c ∈ gamma_congruence_rep y"
      unfolding Some p by simp
    moreover have "gamma_congruence_rep y = {}"
      using gamma_eq ‘x = None‘ by simp
    ultimately show ?thesis by simp
  qed
next
  case (Some p1)
  obtain c1 m1 where p1: "p1 = (c1, m1)"
    by (cases p1)
  show ?thesis
  proof (cases y)
    case None
    have "c1 ∈ gamma_congruence_rep x"
      unfolding Some p1 by simp
    moreover have "gamma_congruence_rep x = {}"
      using gamma_eq None by simp
    ultimately show ?thesis by simp
  next
    case (Some p2)
    obtain c2 m2 where p2: "p2 = (c2, m2)"
      by (cases p2)
    have subset12:
      "{n. m1 dvd n - c1} ⊆ {n. m2 dvd n - c2}"
```

```
    using gamma_eq unfolding ‘x = Some p1‘ Some p1 p2 by auto
  have subset21:
    "{n. m2 dvd n - c2} ⊆ {n. m1 dvd n - c1}"
    using gamma_eq unfolding ‘x = Some p1‘ Some p1 p2 by auto
  have divisibility12: "m2 dvd m1 ∧ m2 dvd c1 - c2"
    using subset12 unfolding congruence_class_subset_iff .
  have divisibility21: "m1 dvd m2 ∧ m1 dvd c2 - c1"
    using subset21 unfolding congruence_class_subset_iff .
  have nonnegative1: "0 <= m1"
    using normalized_x unfolding ‘x = Some p1‘ p1
    by (rule normalized_congruence_rep_modulus_nonnegative)
  have nonnegative2: "0 <= m2"
    using normalized_y unfolding Some p2
    by (rule normalized_congruence_rep_modulus_nonnegative)
  have moduli: "m1 = m2"
    by (rule Int.zdvd_antisym_nonneg[OF nonnegative1 nonnegative2
          divisibility21[THEN conjunct1] divisibility12[THEN conjunct1]])
  have residues: "c1 = c2"
  proof (cases "m1 = 0")
    case True
    with moduli divisibility12 show ?thesis by simp
  next
    case False
    have bounds1: "0 <= c1" "c1 < m1" "0 < m1"
      using normalized_x False unfolding ‘x = Some p1‘ p1
      by (rule normalized_congruence_rep_nonzero)+
    have bounds2: "0 <= c2" "c2 < m1"
      using normalized_y False moduli unfolding Some p2
      by auto
    have mod_eq: "c1 mod m1 = c2 mod m1"
      using divisibility12 moduli
      by (simp only: Euclidean_Rings.euclidean_ring_cancel_class.mod_eq_dvd_iff)
    have left: "c1 mod m1 = c1"
      by (rule mod_pos_pos_trivial[OF bounds1(1) bounds1(2)])
    have right: "c2 mod m1 = c2"
      by (rule mod_pos_pos_trivial[OF bounds2])
    show ?thesis using mod_eq left right by simp
  qed
  show ?thesis
    unfolding ‘x = Some p1‘ Some p1 p2 moduli residues by simp
 qed
qed

lemma gamma_mk_congruence [simp]:
 "gamma_congruence (mk_congruence c m) = {n. m dvd n - c}"
 unfolding gamma_congruence_def
 by (simp only: mk_congruence.rep_eq gamma_normalize_congruence_rep
     gamma_congruence_rep.simps)

lemma mk_congruence_member [simp]:
 "c ∈ gamma_congruence (mk_congruence c m)"
 by simp

lemma gamma_bottom_congruence [simp]:
 "gamma_congruence bottom_congruence = {}"
 unfolding gamma_congruence_def by simp

lemma gamma_congruence_empty_iff [simp]:
 "gamma_congruence a = {} ⟷ a = bottom_congruence"
proof (cases "Rep_congruence a")
 case None
 have "Rep_congruence a = Rep_congruence bottom_congruence"
   using None by simp
 then have "a = bottom_congruence"
   by (simp only: Rep_congruence_inject)
 with None show ?thesis
   unfolding gamma_congruence_def by simp
next
 case (Some p)
 obtain c m where p: "p = (c, m)"
   by (cases p)
 have member: "c ∈ gamma_congruence a"
   unfolding gamma_congruence_def Some p by simp
 have "a ≠ bottom_congruence"
 proof
   assume "a = bottom_congruence"
```

```
    with Some show False by simp
  qed
  with member show ?thesis by blast
qed

lemma gamma_congruence_inject:
  assumes "gamma_congruence a = gamma_congruence b"
  shows "a = b"
proof -
  have reps: "Rep_congruence a = Rep_congruence b"
  proof (rule gamma_congruence_rep_inject)
    show "normalized_congruence_rep (Rep_congruence a)"
      by simp
    show "normalized_congruence_rep (Rep_congruence b)"
      by simp
    show "gamma_congruence_rep (Rep_congruence a) =
      gamma_congruence_rep (Rep_congruence b)"
      using assms unfolding gamma_congruence_def .
  qed
  from reps show ?thesis
    by (simp only: Rep_congruence_inject)
qed

lemma mk_congruence_negative_modulus [simp]:
  "mk_congruence 5 (-4) = mk_congruence 1 4"
  by (rule Rep_congruence_inject[THEN iffD1])
     (simp add: Let_def)


definition congruence_of_int :: "int => congruence" where
  "congruence_of_int n = mk_congruence n 0"

lemma congruence_of_int_gamma [simp]:
  "n ∈ gamma_congruence (congruence_of_int n)"
  unfolding congruence_of_int_def by simp
```

Order

A congruence value denotes a set of integers: `None` denotes none at all and `Some (c, m)` the numbers congruent to `c` modulo `m`, with `m = 0` meaning the single number `c`. Containment between two such sets is decidable arithmetic: `Some (c1, m1)` sits inside `Some (c2, m2)` exactly when the coarser modulus `m2` divides both `m1` and the offset `c1 - c2`. That test is `congruence_le_rep` just below, and it is what everything else here is built on.

From it come the order instance on the normalized type `congruence`, its bottom (the empty class) and top (every integer, `Some (0, 1)`), the join — the coarsest class containing both arguments, computed through the gcd of the moduli and the difference of the residues — and the executable emptiness and printing operations the abstract-domain interface asks for.

```
fun congruence_le_rep :: "congruence_rep => congruence_rep => bool" where
  "congruence_le_rep None _ = True"
| "congruence_le_rep (Some _) None = False"
| "congruence_le_rep (Some (c1, m1)) (Some (c2, m2)) =
     (m2 dvd m1 ∧ m2 dvd c1 - c2)"

lemma congruence_le_rep_iff:
  "congruence_le_rep x y ⟷
   gamma_congruence_rep x ⊆ gamma_congruence_rep y"
proof (cases x)
  case None
  then show ?thesis by simp
next
  case (Some p1)
  obtain c1 m1 where p1: "p1 = (c1, m1)"
    by (cases p1)
  have x_rep: "x = Some (c1, m1)"
    using Some p1 by simp
  show ?thesis
  proof (cases y)
    case None
    have member: "c1 ∈ gamma_congruence_rep x"
      unfolding x_rep by simp
    have not_subset:
      "¬ gamma_congruence_rep x ⊆ gamma_congruence_rep y"
    proof
      assume subset: "gamma_congruence_rep x ⊆ gamma_congruence_rep y"
      have "c1 ∈ gamma_congruence_rep y"
```

```
        by (rule subsetD[OF subset member])
      with None show False by simp
    qed
    with None show ?thesis unfolding x_rep by simp
  next
    case (Some p2)
    obtain c2 m2 where p2: "p2 = (c2, m2)"
      by (cases p2)
    have y_rep: "y = Some (c2, m2)"
      using Some p2 by simp
    show ?thesis
      unfolding x_rep y_rep
      by (simp only: congruence_le_rep.simps gamma_congruence_rep.simps
          congruence_class_subset_iff)
  qed
qed

definition congruence_le :: "congruence => congruence => bool" where
  "congruence_le a b =
     congruence_le_rep (Rep_congruence a) (Rep_congruence b)"

lemma congruence_le_iff_gamma:
  "congruence_le a b ⟷
   gamma_congruence a ⊆ gamma_congruence b"
  unfolding congruence_le_def gamma_congruence_def
  by (rule congruence_le_rep_iff)

instantiation congruence :: ord
begin

definition less_eq_congruence :: "congruence => congruence => bool" where
  "(a :: congruence) <= b = congruence_le a b"

definition less_congruence :: "congruence => congruence => bool" where
  "(a :: congruence) < b = (a <= b ∧ ¬ b <= a)"

instance ..

end

lemma less_eq_congruence_iff_gamma:
  "a ≤ b ⟷ gamma_congruence a ⊆ gamma_congruence b"
  for a b :: congruence
  unfolding less_eq_congruence_def by (rule congruence_le_iff_gamma)

instance congruence :: order
proof intro_classes
  fix a b c :: congruence
  show "(a < b) = (a <= b ∧ ¬ b <= a)"
    unfolding less_congruence_def ..
  show "a <= a"
    unfolding less_eq_congruence_iff_gamma by simp
  assume ab: "a <= b" and bc: "b <= c"
  show "a <= c"
    using ab bc
    unfolding less_eq_congruence_iff_gamma
    by blast
next
  fix a b :: congruence
  assume ab: "a <= b" and ba: "b <= a"
  have gamma_eq: "gamma_congruence a = gamma_congruence b"
    using ab ba
    unfolding less_eq_congruence_iff_gamma
    by blast
  show "a = b"
    by (rule gamma_congruence_inject[OF gamma_eq])
qed

instantiation congruence :: bot
begin

definition bot_congruence :: congruence where
  "bot_congruence = bottom_congruence"

instance ..
```

```
end

instance congruence :: order_bot
proof intro_classes
  fix a :: congruence
  show "bot <= a"
    unfolding less_eq_congruence_iff_gamma bot_congruence_def
    by simp
qed

instantiation congruence :: top
begin

definition top_congruence :: congruence where
  "top_congruence = mk_congruence 0 1"

instance ..

end

lemma gamma_top_congruence [simp]:
  "gamma_congruence (top :: congruence) = UNIV"
  unfolding top_congruence_def by auto

lemma mk_congruence_mod_one [simp]:
  "mk_congruence c 1 = (top :: congruence)"
proof (rule gamma_congruence_inject)
  show "gamma_congruence (mk_congruence c 1) =
    gamma_congruence (top :: congruence)"
    by auto
qed

instance congruence :: order_top
proof intro_classes
  fix a :: congruence
  show "a <= top"
    unfolding less_eq_congruence_iff_gamma
    by simp
qed
```

Join

Two classes join to the class modulo the gcd of both moduli and the residue difference, renormalized. The lemmas show the join is an upper bound in `gamma_congruence` and the least one in the order.

```
fun join_congruence_rep :: "congruence_rep => congruence_rep => congruence_rep" where
  "join_congruence_rep None y = y"
| "join_congruence_rep x None = x"
| "join_congruence_rep (Some (c1, m1)) (Some (c2, m2)) =
     Some (c1, gcd m1 (gcd m2 (c1 - c2)))"

lift_definition join_congruence :: "congruence => congruence => congruence"
  is "λx y. normalize_congruence_rep (join_congruence_rep x y)"
  by (rule normalized_normalize_congruence_rep)

declare join_congruence.rep_eq [simp]

lemma gamma_join_congruence [simp]:
  "gamma_congruence (join_congruence a b) =
   gamma_congruence_rep
     (join_congruence_rep (Rep_congruence a) (Rep_congruence b))"
  unfolding gamma_congruence_def by simp

lemma gamma_join_congruence_rep_ub1:
  "gamma_congruence_rep x ⊆
   gamma_congruence_rep (join_congruence_rep x y)"
proof (cases x)
  case None
  then show ?thesis by simp
next
  case (Some p1)
  obtain c1 m1 where p1: "p1 = (c1, m1)"
    by (cases p1)
  have x_rep: "x = Some (c1, m1)"
    using Some p1 by simp
  show ?thesis
```

```
  proof (cases y)
    case None
    with x_rep show ?thesis by simp
  next
    case (Some p2)
    obtain c2 m2 where p2: "p2 = (c2, m2)"
      by (cases p2)
    have y_rep: "y = Some (c2, m2)"
      using Some p2 by simp
    have modulus:
      "gcd m1 (gcd m2 (c1 - c2)) dvd m1"
      by (rule gcd_dvd1)
    show ?thesis
      unfolding x_rep y_rep
      apply (simp only: join_congruence_rep.simps gamma_congruence_rep.simps)
      apply (subst congruence_class_subset_iff)
      using modulus by simp
  qed
qed

lemma gamma_join_congruence_rep_ub2:
  "gamma_congruence_rep y ⊆
   gamma_congruence_rep (join_congruence_rep x y)"
proof (cases x)
  case None
  then show ?thesis by simp
next
  case (Some p1)
  obtain c1 m1 where p1: "p1 = (c1, m1)"
    by (cases p1)
  have x_rep: "x = Some (c1, m1)"
    using Some p1 by simp
  show ?thesis
  proof (cases y)
    case None
    with x_rep show ?thesis by simp
  next
    case (Some p2)
    obtain c2 m2 where p2: "p2 = (c2, m2)"
      by (cases p2)
    have y_rep: "y = Some (c2, m2)"
      using Some p2 by simp
    let ?g = "gcd m1 (gcd m2 (c1 - c2))"
    have modulus: "?g dvd m2"
      by (rule dvd_trans[OF gcd_dvd2 gcd_dvd1])
    have offset_forward: "?g dvd c1 - c2"
      by (rule dvd_trans[OF gcd_dvd2 gcd_dvd2])
    have offset: "?g dvd c2 - c1"
      by (subst dvd_diff_commute) (rule offset_forward)
    show ?thesis
      unfolding x_rep y_rep
      apply (simp only: join_congruence_rep.simps gamma_congruence_rep.simps)
      apply (subst congruence_class_subset_iff)
      using modulus offset by simp
  qed
qed

lemma gamma_join_congruence_rep_least:
  assumes xz:
    "gamma_congruence_rep x ⊆ gamma_congruence_rep z"
      and yz:
    "gamma_congruence_rep y ⊆ gamma_congruence_rep z"
  shows
    "gamma_congruence_rep (join_congruence_rep x y) ⊆
     gamma_congruence_rep z"
proof (cases x)
  case None
  with yz show ?thesis by simp
next
  case (Some p1)
  obtain c1 m1 where p1: "p1 = (c1, m1)"
    by (cases p1)
  have x_rep: "x = Some (c1, m1)"
    using Some p1 by simp
  show ?thesis
  proof (cases y)
```

```
    case None
    with x_rep xz show ?thesis by simp
  next
    case (Some p2)
    obtain c2 m2 where p2: "p2 = (c2, m2)"
      by (cases p2)
    have y_rep: "y = Some (c2, m2)"
      using Some p2 by simp
    show ?thesis
    proof (cases z)
      case None
      have member: "c1 ∈ gamma_congruence_rep x"
        unfolding x_rep by simp
      have "c1 ∈ gamma_congruence_rep z"
        by (rule subsetD[OF xz member])
      with None show ?thesis by simp
    next
      case (Some p3)
      obtain c3 m3 where p3: "p3 = (c3, m3)"
        by (cases p3)
      have z_rep: "z = Some (c3, m3)"
        using Some p3 by simp
      have x_divisibility:
        "m3 dvd m1 ∧ m3 dvd c1 - c3"
        using xz
        unfolding x_rep z_rep
        by (simp only: gamma_congruence_rep.simps
            congruence_class_subset_iff)
      have y_divisibility:
        "m3 dvd m2 ∧ m3 dvd c2 - c3"
        using yz
        unfolding y_rep z_rep
        by (simp only: gamma_congruence_rep.simps
            congruence_class_subset_iff)
      have offset: "m3 dvd c1 - c2"
      proof -
        have "m3 dvd (c1 - c3) - (c2 - c3)"
          by (rule dvd_diff[OF x_divisibility[THEN conjunct2]
                y_divisibility[THEN conjunct2]])
        then show ?thesis by simp
      qed
      have inner: "m3 dvd gcd m2 (c1 - c2)"
        by (rule gcd_greatest[OF y_divisibility[THEN conjunct1] offset])
      have modulus:
        "m3 dvd gcd m1 (gcd m2 (c1 - c2))"
        by (rule gcd_greatest[OF x_divisibility[THEN conjunct1] inner])
      show ?thesis
        unfolding x_rep y_rep z_rep
        apply (simp only: join_congruence_rep.simps gamma_congruence_rep.simps)
        apply (subst congruence_class_subset_iff)
        using modulus x_divisibility by simp
    qed
  qed
qed

lemma join_congruence_ub1:
  "a <= join_congruence a b"
  unfolding less_eq_congruence_iff_gamma
    gamma_congruence_def
  apply (simp only: join_congruence.rep_eq gamma_normalize_congruence_rep)
  by (rule gamma_join_congruence_rep_ub1)

lemma join_congruence_ub2:
  "b <= join_congruence a b"
  unfolding less_eq_congruence_iff_gamma
    gamma_congruence_def
  apply (simp only: join_congruence.rep_eq gamma_normalize_congruence_rep)
  by (rule gamma_join_congruence_rep_ub2)

lemma join_congruence_least:
  assumes "a <= c" and "b <= c"
  shows "join_congruence a b <= c"
  using assms
  unfolding less_eq_congruence_iff_gamma
    gamma_congruence_def
  apply (simp only: join_congruence.rep_eq gamma_normalize_congruence_rep)
```

```
  by (rule gamma_join_congruence_rep_least)

instantiation congruence :: sup
begin

definition sup_congruence :: "congruence => congruence => congruence" where
  "sup_congruence = join_congruence"

instance ..

end

instance congruence :: semilattice_sup
proof intro_classes
  fix a b c :: congruence
  show "a <= a ⊔ b"
    unfolding sup_congruence_def by (rule join_congruence_ub1)
  show "b <= a ⊔ b"
    unfolding sup_congruence_def by (rule join_congruence_ub2)
  show "b <= a ⟹ c <= a ⟹ b ⊔ c <= a"
    unfolding sup_congruence_def by (rule join_congruence_least)
qed

instance congruence :: bounded_semilattice_sup_bot ..
```

Meet

Intersecting two non-singleton congruence classes is the executable Chinese remainder construction. The Bezout coefficient computes one shared residue; the least common multiple is the period of every shared solution.

```
fun intersect_congruence_rep ::
    "congruence_rep => congruence_rep => congruence_rep"
where
  "intersect_congruence_rep None y = None"
| "intersect_congruence_rep x None = None"
| "intersect_congruence_rep (Some (c1, m1)) (Some (c2, m2)) =
     normalize_congruence_rep
       (if m1 = 0 then
          if m2 dvd c1 - c2 then Some (c1, 0) else None
        else if m2 = 0 then
          if m1 dvd c2 - c1 then Some (c2, 0) else None
        else
          let g = gcd m1 m2
          in if g dvd c2 - c1 then
               let s = fst (bezout_coefficients m1 m2);
                   q = (c2 - c1) div g
               in Some (c1 + m1 * (q * s), lcm m1 m2)
             else None)"

lemma normalized_intersect_congruence_rep [simp]:
  "normalized_congruence_rep (intersect_congruence_rep x y)"
  by (cases x; cases y)
     (auto simp: split_def split: if_splits prod.splits)

instantiation congruence :: inf
begin

lift_definition inf_congruence ::
    "congruence => congruence => congruence"
  is intersect_congruence_rep
  by simp

instance ..

end

lemma bezout_shared_residue:
  assumes compatible: "gcd m1 m2 dvd c2 - c1"
  defines
    "q == (c2 - c1) div gcd m1 m2"
  shows
    "m1 dvd
       (c1 + m1 * (q * fst (bezout_coefficients m1 m2))) - c1 ∧
     m2 dvd
       (c1 + m1 * (q * fst (bezout_coefficients m1 m2))) - c2"
proof (intro conjI)
```

```
  show
    "m1 dvd
      c1 + m1 * (q * fst (bezout_coefficients m1 m2)) - c1"
    by simp
  have bezout:
    "fst (bezout_coefficients m1 m2) * m1 +
     snd (bezout_coefficients m1 m2) * m2 =
     gcd m1 m2"
    by (rule bezout_coefficients_fst_snd)
  have quotient:
    "q * gcd m1 m2 = c2 - c1"
    unfolding q_def
    using dvd_mult_div_cancel[OF compatible]
    by (simp add: mult.commute)
  have scaled0:
    "q * (fst (bezout_coefficients m1 m2) * m1 +
       snd (bezout_coefficients m1 m2) * m2) =
     q * gcd m1 m2"
    by (rule arg_cong[OF bezout])
  have scaled:
    "m1 * (q * fst (bezout_coefficients m1 m2)) +
     m2 * (q * snd (bezout_coefficients m1 m2)) =
     q * gcd m1 m2"
    using scaled0 by (simp add: algebra_simps)
  show
    "m2 dvd
      c1 + m1 * (q * fst (bezout_coefficients m1 m2)) - c2"
    unfolding dvd_def
  proof (rule exI[of _ "- q * snd (bezout_coefficients m1 m2)"])
    show
      "c1 + m1 * (q * fst (bezout_coefficients m1 m2)) - c2 =
       m2 * (- q * snd (bezout_coefficients m1 m2))"
      using scaled quotient by (simp add: algebra_simps)
  qed
qed


lemma gamma_intersect_congruence_rep:
  assumes normalized_x: "normalized_congruence_rep x"
      and normalized_y: "normalized_congruence_rep y"
  shows
    "gamma_congruence_rep (intersect_congruence_rep x y) =
     gamma_congruence_rep x ∩ gamma_congruence_rep y"
proof (cases x)
  case None
  then show ?thesis by simp
next
  case (Some cm1)
  note x_some = Some
  obtain c1 m1 where cm1: "cm1 = (c1, m1)"
    by (cases cm1)
  show ?thesis
  proof (cases y)
    case None
    with x_some show ?thesis by simp
  next
    case (Some cm2)
    note y_some = Some
    obtain c2 m2 where cm2: "cm2 = (c2, m2)"
      by (cases cm2)
    show ?thesis
    proof (cases "m1 = 0")
      case True
      with x_some y_some cm1 cm2 show ?thesis
        by auto
    next
      case m1_nonzero: False
      show ?thesis
      proof (cases "m2 = 0")
        case True
        with x_some y_some cm1 cm2 show ?thesis
          by auto
      next
        case m2_nonzero: False
        let ?g = "gcd m1 m2"
        let ?q = "(c2 - c1) div ?g"
```

```
        let ?s = "fst (bezout_coefficients m1 m2)"
        let ?z = "c1 + m1 * (?q * ?s)"
        show ?thesis
        proof (cases "?g dvd c2 - c1")
          case incompatible: False
          have empty:
            "{n. m1 dvd n - c1} ∩
             {n. m2 dvd n - c2} = {}"
          proof (rule ccontr)
            assume
              "{n. m1 dvd n - c1} ∩
               {n. m2 dvd n - c2} ~= {}"
            then obtain n where
              n1: "m1 dvd n - c1" and
              n2: "m2 dvd n - c2"
              by blast
            have g1: "?g dvd n - c1"
              using gcd_dvd1 n1 by (rule dvd_trans)
            have g2: "?g dvd n - c2"
              using gcd_dvd2 n2 by (rule dvd_trans)
            have "?g dvd (n - c1) - (n - c2)"
              by (rule dvd_diff[OF g1 g2])
            then have "?g dvd c2 - c1"
              by simp
            with incompatible show False by contradiction
          qed
          with x_some y_some cm1 cm2 m1_nonzero m2_nonzero
            incompatible
          show ?thesis by simp
        next
          case compatible: True
          have shared:
            "m1 dvd ?z - c1 ∧ m2 dvd ?z - c2"
            by (rule bezout_shared_residue[OF compatible])
          have classes:
            "{n. lcm m1 m2 dvd n - ?z} =
             {n. m1 dvd n - c1} ∩
             {n. m2 dvd n - c2}"
          proof -
            have "m1 dvd n - ?z ⟷ m1 dvd n - c1" "m2 dvd n - ?z ⟷ m2 dvd n - c2" for n
              using shared dvd_add_left_iff[of m1 "?z - c1" "n - ?z"]
                dvd_add_left_iff[of m2 "?z - c2" "n - ?z"]
              by simp_all
            then show ?thesis by auto
          qed
          have normalized_gamma:
            "gamma_congruence_rep
               (normalize_congruence_rep
                 (Some (?z, lcm m1 m2))) =
             {n. lcm m1 m2 dvd n - ?z}"
            by (simp only: gamma_normalize_congruence_rep
                gamma_congruence_rep.simps)
          from x_some y_some cm1 cm2 m1_nonzero m2_nonzero
            compatible normalized_gamma classes
          show ?thesis
            by (simp only: intersect_congruence_rep.simps if_False
                if_True Let_def gamma_congruence_rep.simps)
        qed
      qed
    qed
  qed
qed

lemma gamma_inf_congruence [simp]:
  "gamma_congruence (a ⊓ b) =
   gamma_congruence a ∩ gamma_congruence b"
proof -
  have norm_a: "normalized_congruence_rep (Rep_congruence a)"
    by simp
  have norm_b: "normalized_congruence_rep (Rep_congruence b)"
    by simp
  show ?thesis
    using gamma_intersect_congruence_rep[OF norm_a norm_b]
    by (simp add: gamma_congruence_def inf_congruence.rep_eq)
qed
```

```
instance congruence :: semilattice_inf
  by standard (auto simp: less_eq_congruence_iff_gamma)

instance congruence :: lattice ..
instance congruence :: bounded_lattice_bot ..
```

Executable interface

Executable bottom and top tests, each proved against `gamma_congruence`.

```
definition is_bottom_congruence :: "congruence => bool" where
  "is_bottom_congruence a = (a = bot)"

lemma is_bottom_congruence_correct:
  "is_bottom_congruence a ⟷ gamma_congruence a = {}"
  unfolding is_bottom_congruence_def bot_congruence_def
  by simp

definition is_top_congruence :: "congruence => bool" where
  "is_top_congruence a = (a = top)"

lemma is_top_congruence_correct_gamma:
  "is_top_congruence a ⟷ gamma_congruence a = UNIV"
  unfolding is_top_congruence_def
  by (metis gamma_congruence_inject gamma_top_congruence)
```

Goblint's congruence notation: a constant prints as itself, the class of `r` modulo `m` as `r+mℤ`, with a zero residue and a unit modulus left out, so `3ℤ`, `1+3ℤ` and `ℤ`. A standalone congruence prints its top as ⊤.

```
definition string_of_congruence :: "congruence ⇒ String.literal" where
  "string_of_congruence c =
     (case Rep_congruence c of
        None ⇒ sym_bottom
      | Some (r, m) ⇒
          if m = 0 then string_of_int r
          else (if r = 0 then STR '''' else string_of_int r + STR ''+'')
             + (if m = 1 then STR '''' else string_of_int m) + sym_int)"

end
```

2.17 Int_Lattice

```
theory Int_Lattice
  imports Sign_Lattice Interval_Lattice Parity_Lattice Congruence_Lattice
begin
```

Integer product lattice

The carrier keeps the four scalar abstractions independently. Its concretization is their intersection; componentwise order and join preserve the existing domain structures. Cross-domain reduction is separate from this carrier definition.

Carrier

```
record int_dom =
  int_sign :: sign
  int_ivl :: ivl
  int_parity :: parity
  int_congruence :: congruence
```

Order, join and meet

The record package represents a closed record through an extensible scheme. Giving the scheme a componentwise instance makes the closed `int_dom` abbreviation usable at every existing lattice-polymorphic boundary. The scheme's `more` field is Isabelle machinery, not a semantic extension point: new integer components belong in `int_dom` and must participate explicitly in its lattice, concretization, and exact bottom decision.

```
instantiation int_dom_ext :: (bounded_lattice) bounded_lattice
begin

definition less_eq_int_dom_ext ::
  "'a int_dom_scheme => 'a int_dom_scheme => bool" where
  "less_eq_int_dom_ext a b ⟷
     int_sign a <= int_sign b ∧
     int_ivl a <= int_ivl b ∧
     int_parity a <= int_parity b ∧
     int_congruence a <= int_congruence b ∧
     more a <= more b"

definition less_int_dom_ext ::
  "'a int_dom_scheme => 'a int_dom_scheme => bool" where
  "less_int_dom_ext a b ⟷ a ≤ b ∧ ¬ b ≤ a"

definition sup_int_dom_ext ::
  "'a int_dom_scheme => 'a int_dom_scheme => 'a int_dom_scheme" where
  "sup_int_dom_ext a b =
     int_dom_ext
       (sup (int_sign a) (int_sign b))
       (sup (int_ivl a) (int_ivl b))
       (sup (int_parity a) (int_parity b))
       (sup (int_congruence a) (int_congruence b))
       (sup (more a) (more b))"

definition inf_int_dom_ext ::
  "'a int_dom_scheme => 'a int_dom_scheme => 'a int_dom_scheme" where
  "inf_int_dom_ext a b =
     int_dom_ext
       (inf (int_sign a) (int_sign b))
       (inf (int_ivl a) (int_ivl b))
       (inf (int_parity a) (int_parity b))
       (inf (int_congruence a) (int_congruence b))
       (inf (more a) (more b))"

definition bot_int_dom_ext :: "'a int_dom_scheme" where
  "bot_int_dom_ext = int_dom_ext bot bot bot bot bot"

definition top_int_dom_ext :: "'a int_dom_scheme" where
  "top_int_dom_ext = int_dom_ext top top top top top"

instance
proof intro_classes
  fix x y z :: "'a int_dom_scheme"
  show "x < y ⟷ x ≤ y ∧ ¬ y ≤ x"
    by (simp add: less_int_dom_ext_def)
```

```
  show "x ≤ x"
    by (simp add: less_eq_int_dom_ext_def)
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
    by (auto simp: less_eq_int_dom_ext_def intro: order_trans)
  show "x ≤ y ⟹ y ≤ x ⟹ x = y"
    by (cases x; cases y)
       (auto simp: less_eq_int_dom_ext_def intro: antisym)
  show "x ≤ sup x y"
    by (simp add: less_eq_int_dom_ext_def sup_int_dom_ext_def)
  show "y ≤ sup x y"
    by (simp add: less_eq_int_dom_ext_def sup_int_dom_ext_def)
  show "y ≤ x ⟹ z ≤ x ⟹ sup y z ≤ x"
    by (auto simp: less_eq_int_dom_ext_def sup_int_dom_ext_def)
  show "inf x y ≤ x"
    by (simp add: less_eq_int_dom_ext_def inf_int_dom_ext_def)
  show "inf x y ≤ y"
    by (simp add: less_eq_int_dom_ext_def inf_int_dom_ext_def)
  show "x ≤ y ⟹ x ≤ z ⟹ x ≤ inf y z"
    by (auto simp: less_eq_int_dom_ext_def inf_int_dom_ext_def)
  show "bot ≤ x"
    by (simp add: less_eq_int_dom_ext_def bot_int_dom_ext_def)
  show "x ≤ top"
    by (simp add: less_eq_int_dom_ext_def top_int_dom_ext_def)
qed

end

definition int_dom_sip :: "sign => ivl => parity => int_dom" where
  "int_dom_sip s i p =
     (top :: int_dom)(|int_sign := s, int_ivl := i, int_parity := p|)"

definition int_dom_sipc ::
  "sign => ivl => parity => congruence => int_dom" where
  "int_dom_sipc s i p c =
     (top :: int_dom)(|
       int_sign := s,
       int_ivl := i,
       int_parity := p,
       int_congruence := c
     |)"

lemma int_congruence_int_dom_sip [simp]:
  "int_congruence (int_dom_sip s i p) = top"
  by (simp add: int_dom_sip_def top_int_dom_ext_def)
```

Concretization

A product value denotes the integers every component admits: `gamma_int_dom` intersects the sign, interval, parity and congruence concretizations.

```
definition gamma_int_dom :: "'a int_dom_scheme => int set" where
  "gamma_int_dom d =
     gamma_sign (int_sign d) ∩
     gamma_ivl (int_ivl d) ∩
     gamma_parity (int_parity d) ∩
     gamma_congruence (int_congruence d)"
```

Every component meet is exact, so the product meet is too: it concretizes to the intersection of its operands. Query answers from several analyses are combined by ⊓, which therefore neither loses nor invents values.

```
lemma gamma_inf_int_dom [simp]:
  "gamma_int_dom (a ⊓ b) = gamma_int_dom a ∩ gamma_int_dom b"
  by (auto simp: gamma_int_dom_def inf_int_dom_ext_def)

lemma gamma_int_dom_update_sign [simp]:
  "gamma_int_dom (d(|int_sign := s|)) =
   gamma_sign s ∩
   gamma_ivl (int_ivl d) ∩
   gamma_parity (int_parity d) ∩
   gamma_congruence (int_congruence d)"
  by (simp add: gamma_int_dom_def)

lemma gamma_int_dom_update_ivl [simp]:
  "gamma_int_dom (d(|int_ivl := i|)) =
   gamma_sign (int_sign d) ∩
   gamma_ivl i ∩
```

```
   gamma_parity (int_parity d) ∩
   gamma_congruence (int_congruence d)"
  by (simp add: gamma_int_dom_def)

lemma gamma_int_dom_update_parity [simp]:
  "gamma_int_dom (d(|int_parity := p|)) =
   gamma_sign (int_sign d) ∩
   gamma_ivl (int_ivl d) ∩
   gamma_parity p ∩
   gamma_congruence (int_congruence d)"
  by (simp add: gamma_int_dom_def)

lemma gamma_int_dom_update_congruence [simp]:
  "gamma_int_dom (d(|int_congruence := c|)) =
   gamma_sign (int_sign d) ∩
   gamma_ivl (int_ivl d) ∩
   gamma_parity (int_parity d) ∩
   gamma_congruence c"
  by (simp add: gamma_int_dom_def)

fun interval_fact_of_sign :: "sign ⇒ ivl" where
  "interval_fact_of_sign SBot = ⊥"
| "interval_fact_of_sign SNeg = Ivl MinInf (Fin (-1))"
| "interval_fact_of_sign SNonPos = Ivl MinInf (Fin 0)"
| "interval_fact_of_sign SZero = Ivl (Fin 0) (Fin 0)"
| "interval_fact_of_sign SNonNeg = Ivl (Fin 0) PlusInf"
| "interval_fact_of_sign SPos = Ivl (Fin 1) PlusInf"
| "interval_fact_of_sign STop = ⊤"

lemma gamma_interval_fact_of_sign [simp]:
  "gamma_ivl (interval_fact_of_sign s) = gamma_sign s"
  by (cases s) (auto simp: bot_ivl_def top_ivl_def ivl_top_def)

fun congruence_fact_of_parity :: "parity => congruence" where
  "congruence_fact_of_parity PBot = bot"
| "congruence_fact_of_parity PEven = mk_congruence 0 2"
| "congruence_fact_of_parity POdd = mk_congruence 1 2"
| "congruence_fact_of_parity PTop = top"

lemma gamma_congruence_fact_of_parity [simp]:
  "gamma_congruence (congruence_fact_of_parity p) = gamma_parity p"
  by (cases p)
     (auto simp: bot_congruence_def)

lemma gamma_intersect_ivl_exact:
  "gamma_ivl (intersect_ivl a b) = gamma_ivl a ∩ gamma_ivl b"
  using gamma_ivl_mono[OF intersect_ivl_le1]
        gamma_ivl_mono[OF intersect_ivl_le2]
        intersect_ivl_gamma
  by blast
```

Executable interface

Deciding emptiness of the product needs more than the components' own tests, since each can be inhabited while their intersection is empty. Interval and congruence are checked jointly, after restricting the congruence by parity; `is_bottom_int_dom_correct` and `is_top_int_dom_correct_gamma` tie the tests to `gamma_int_dom`.

```
lemma inter_nonempty_iff:
  "A ∩ B ≠ {} ⟷ (∃x. x ∈ A ∧ x ∈ B)"
  by blast

lemma congruence_between_iff:
  assumes normalized: "normalized_congruence_rep (Some (c, m))"
  shows
    "(∃n::int. l ≤ n ∧ n ≤ u ∧ m dvd n - c) ⟷
     (if m = 0 then l ≤ c ∧ c ≤ u
      else l ≤ u ∧ l + ((c - l) mod m) ≤ u)"
proof (cases "m = 0")
  case True
  then show ?thesis by simp
next
  case False
  from normalized_congruence_rep_nonzero[OF normalized False]
  have modulus: "0 < m" by simp
  show ?thesis
  proof
```

```
    assume "∃n::int. l ≤ n ∧ n ≤ u ∧ m dvd n - c"
    then obtain n where n: "l ≤ n" "n ≤ u" "m dvd n - c"
      by blast
    have offset: "m dvd c - n"
      using n(3) by (metis dvd_minus_iff minus_diff_eq)
    have mod_eq: "(c - l) mod m = (n - l) mod m"
      using offset
      by (simp add:
          Euclidean_Rings.euclidean_ring_cancel_class.mod_eq_dvd_iff)
    have remainder_le: "(n - l) mod m ≤ n - l"
      by (rule Euclidean_Rings.zmod_le_nonneg_dividend) (use n(1) in linarith)
    show
      "if m = 0 then l ≤ c ∧ c ≤ u
       else l ≤ u ∧ l + ((c - l) mod m) ≤ u"
      using False mod_eq n(1,2) remainder_le
      by simp
  next
    assume rhs:
      "if m = 0 then l ≤ c ∧ c ≤ u
       else l ≤ u ∧ l + ((c - l) mod m) ≤ u"
    let ?n = "l + ((c - l) mod m)"
    have lower: "l ≤ ?n"
      using modulus by simp
    have congruent: "m dvd ?n - c"
    proof -
      have base: "m dvd (c - l) - (c - l) mod m"
        by (rule dvd_minus_mod)
      have "m dvd - ((c - l) - (c - l) mod m)"
        using base by (simp only: dvd_minus_iff)
      also have
        "- ((c - l) - (c - l) mod m) = ?n - c"
        by simp
      finally show ?thesis .
    qed
    have upper: "?n ≤ u"
      using rhs False by simp
    show "∃n::int. l ≤ n ∧ n ≤ u ∧ m dvd n - c"
      by (rule exI[of _ ?n]) (use lower upper congruent in blast)
  qed
qed

lemma congruence_class_nonempty:
  "∃n::int. m dvd n - c"
  by (rule exI[of _ c]) simp

lemma exists_congruence_le:
  assumes normalized: "normalized_congruence_rep (Some (c, m))"
      and nonzero: "m ≠ 0"
  shows "∃n::int. n ≤ u ∧ m dvd n - c"
proof -
  from normalized_congruence_rep_nonzero[OF normalized nonzero]
  have one_le: "1 ≤ m" by simp
  let ?q = "abs (c - u) + 1"
  have q_nonnegative: "0 ≤ ?q" by simp
  have difference_le_q: "c - u ≤ ?q"
    using abs_ge_self[of "c - u"] by linarith
  have scale: "?q ≤ m * ?q"
    using mult_right_mono[OF one_le q_nonnegative] by simp
  let ?n = "c - m * ?q"
  have upper: "?n ≤ u"
    using difference_le_q scale by linarith
  have congruent: "m dvd ?n - c"
    by simp
  show ?thesis
    by (rule exI[of _ ?n]) (use upper congruent in blast)
qed

lemma exists_congruence_ge:
  assumes normalized: "normalized_congruence_rep (Some (c, m))"
      and nonzero: "m ≠ 0"
  shows "∃n::int. l ≤ n ∧ m dvd n - c"
proof -
  from normalized_congruence_rep_nonzero[OF normalized nonzero]
  have one_le: "1 ≤ m" by simp
  let ?q = "abs (l - c) + 1"
  have q_nonnegative: "0 ≤ ?q" by simp
```

```
  have difference_le_q: "l - c ≤ ?q"
    using abs_ge_self[of "l - c"] by linarith
  have scale: "?q ≤ m * ?q"
    using mult_right_mono[OF one_le q_nonnegative] by simp
  let ?n = "c + m * ?q"
  have lower: "l ≤ ?n"
    using difference_le_q scale by linarith
  have congruent: "m dvd ?n - c"
    by simp
  show ?thesis
    by (rule exI[of _ ?n]) (use lower congruent in blast)
qed

fun ivl_congruence_rep_nonempty ::
  "ivl => congruence_rep => bool" where
  "ivl_congruence_rep_nonempty i None = False"
| "ivl_congruence_rep_nonempty (Ivl MinInf MinInf) (Some cm) = False"
| "ivl_congruence_rep_nonempty (Ivl MinInf (Fin u)) (Some (c, m)) =
     (if m = 0 then c ≤ u else True)"
| "ivl_congruence_rep_nonempty (Ivl MinInf PlusInf) (Some cm) = True"
| "ivl_congruence_rep_nonempty (Ivl (Fin l) MinInf) (Some cm) = False"
| "ivl_congruence_rep_nonempty (Ivl (Fin l) (Fin u)) (Some (c, m)) =
     (if m = 0 then l ≤ c ∧ c ≤ u
      else l ≤ u ∧ l + ((c - l) mod m) ≤ u)"
| "ivl_congruence_rep_nonempty (Ivl (Fin l) PlusInf) (Some (c, m)) =
     (if m = 0 then l ≤ c else True)"
| "ivl_congruence_rep_nonempty (Ivl PlusInf u) (Some cm) = False"

lemma ivl_congruence_rep_nonempty_correct:
  assumes normalized: "normalized_congruence_rep r"
  shows
    "ivl_congruence_rep_nonempty i r ⟷
     gamma_ivl i ∩ gamma_congruence_rep r ≠ {}"
proof (cases r)
  case None
  then show ?thesis by simp
next
  case (Some cm)
  obtain c m where cm: "cm = (c, m)"
    by (cases cm)
  have rep: "r = Some (c, m)"
    using Some cm by simp
  have normalized': "normalized_congruence_rep (Some (c, m))"
    using normalized unfolding rep .
  show ?thesis
  proof (cases i)
    case (Ivl l u)
    show ?thesis
      unfolding rep Ivl
      by (cases l; cases u; cases "m = 0")
         (auto simp: inter_nonempty_iff
            congruence_between_iff[OF normalized']
            congruence_class_nonempty
            exists_congruence_le[OF normalized']
            exists_congruence_ge[OF normalized'])
  qed
qed

fun parity_accepts :: "parity => int => bool" where
  "parity_accepts PBot n = False"
| "parity_accepts PEven n = even n"
| "parity_accepts POdd n = odd n"
| "parity_accepts PTop n = True"

lemma parity_accepts_gamma [simp]:
  "parity_accepts p n ⟷ n ∈ gamma_parity p"
  by (cases p) simp_all

lemma parity_accepts_difference:
  assumes concrete: "p = PEven ∨ p = POdd"
      and accepts: "parity_accepts p r"
  shows "2 dvd n - r ⟷ parity_accepts p n"
  using concrete accepts
  by (cases p) auto

lemma dvd_shift_modulus:
```

```
  fixes m n c :: int
  shows "m dvd n - (c + m) ⟷ m dvd n - c"
proof
  assume shifted: "m dvd n - (c + m)"
  have "m dvd (n - (c + m)) + m"
    by (rule dvd_add[OF shifted dvd_refl[of m]])
  then show "m dvd n - c" by simp
next
  assume original: "m dvd n - c"
  have "m dvd (n - c) - m"
    by (rule dvd_diff[OF original dvd_refl[of m]])
  also have "(n - c) - m = n - (c + m)"
    by simp
  finally show "m dvd n - (c + m)" .
qed

lemma gamma_normalized_lcm_two:
  "gamma_congruence_rep
     (normalize_congruence_rep (Some (r, lcm m 2))) =
   {n. m dvd n - r ∧ 2 dvd n - r}"
  by (simp only: gamma_normalize_congruence_rep
      gamma_congruence_rep.simps
      GCD.semiring_gcd_class.lcm_least_iff)

lemma congruent_even_modulus:
  fixes m n c :: int
  assumes modulus: "even m"
      and congruent: "m dvd n - c"
  shows "even n ⟷ even c"
proof -
  obtain k where difference: "n - c = m * k"
    using congruent unfolding dvd_def by blast
  have difference_even: "even (n - c)"
    using modulus difference by simp
  have parity_eq: "even (n - c) = even (n + c)"
    by (rule even_diff_iff)
  have sum_even: "even (n + c)"
    using parity_eq difference_even by blast
  show ?thesis
    using sum_even
    by (simp only: Parity.semiring_parity_class.even_add)
qed

lemma normalized_lcm_two_inter_parity:
  assumes concrete: "p = PEven ∨ p = POdd"
      and accepts: "parity_accepts p r"
  shows
    "gamma_congruence_rep
       (normalize_congruence_rep (Some (r, lcm m 2))) =
     gamma_parity p ∩ {n. m dvd n - r}"
proof -
  have difference:
    "⋀n. 2 dvd n - r ⟷ parity_accepts p n"
    by (rule parity_accepts_difference[OF concrete accepts])
  show ?thesis
    unfolding gamma_normalized_lcm_two
    using difference by auto
qed

lemma even_modulus_disjoint_parity:
  assumes concrete: "p = PEven ∨ p = POdd"
      and modulus: "even m"
      and rejects: "¬ parity_accepts p c"
  shows "gamma_parity p ∩ {n. m dvd n - c} = {}"
proof (rule equals0I)
  fix n
  assume member:
    "n ∈ gamma_parity p ∩ {n. m dvd n - c}"
  from member have accepts: "parity_accepts p n"
    by simp
  from member have congruent: "m dvd n - c"
    by simp
  have same: "even n ⟷ even c"
    by (rule congruent_even_modulus[OF modulus congruent])
  from concrete rejects accepts same show False
    by (cases p) auto
```

```
qed

lemma parity_accepts_add_odd:
  assumes concrete: "p = PEven ∨ p = POdd"
      and rejects: "¬ parity_accepts p c"
      and modulus: "odd m"
  shows "parity_accepts p (c + m)"
  using concrete rejects modulus by (cases p) auto

fun restrict_congruence_rep_by_parity ::
  "parity => congruence_rep => congruence_rep" where
  "restrict_congruence_rep_by_parity PBot r = None"
| "restrict_congruence_rep_by_parity PTop r = r"
| "restrict_congruence_rep_by_parity p None = None"
| "restrict_congruence_rep_by_parity p (Some (c, m)) =
    (if m = 0 then
       if parity_accepts p c then Some (c, 0) else None
     else if even m ∧ ¬ parity_accepts p c then None
     else
       normalize_congruence_rep
         (Some
           (c + (if parity_accepts p c then 0 else m),
            lcm m 2)))"

lemma normalized_restrict_congruence_rep_by_parity:
  assumes "normalized_congruence_rep r"
  shows
    "normalized_congruence_rep
       (restrict_congruence_rep_by_parity p r)"
  using assms
  by (cases p; cases r)
     (auto simp only: restrict_congruence_rep_by_parity.simps
       normalized_congruence_rep.simps
       normalized_normalize_congruence_rep
       option.simps
       split: prod.splits if_splits)

lemma restrict_concrete_parity_correct:
  assumes concrete: "p = PEven ∨ p = POdd"
  shows
    "gamma_congruence_rep
       (restrict_congruence_rep_by_parity p (Some (c, m))) =
     gamma_parity p ∩ gamma_congruence_rep (Some (c, m))"
proof (cases "m = 0")
  case True
  with concrete show ?thesis
    by (cases p) auto
next
  case nonzero: False
  show ?thesis
  proof (cases "even m")
    case modulus: True
    show ?thesis
    proof (cases "parity_accepts p c")
      case accepts: True
      have reduced:
        "restrict_congruence_rep_by_parity p (Some (c, m)) =
         normalize_congruence_rep (Some (c, lcm m 2))"
        using concrete nonzero modulus accepts
        by (cases p) simp_all
      have exact:
        "gamma_congruence_rep
           (normalize_congruence_rep (Some (c, lcm m 2))) =
         gamma_parity p ∩ {n. m dvd n - c}"
        by (rule normalized_lcm_two_inter_parity[OF concrete accepts])
      show ?thesis
        unfolding reduced
        using exact by simp
    next
      case rejects: False
      have reduced:
        "restrict_congruence_rep_by_parity p (Some (c, m)) = None"
        using concrete nonzero modulus rejects
        by (cases p) simp_all
      have disjoint:
        "gamma_parity p ∩ {n. m dvd n - c} = {}"
```

```
      by (rule even_modulus_disjoint_parity[OF concrete modulus rejects])
    show ?thesis
      unfolding reduced
      using disjoint by simp
  qed
 next
  case modulus: False
  show ?thesis
  proof (cases "parity_accepts p c")
    case accepts: True
    have reduced:
      "restrict_congruence_rep_by_parity p (Some (c, m)) =
       normalize_congruence_rep (Some (c, lcm m 2))"
      using concrete nonzero modulus accepts
      by (cases p) simp_all
    have exact:
      "gamma_congruence_rep
         (normalize_congruence_rep (Some (c, lcm m 2))) =
       gamma_parity p ∩ {n. m dvd n - c}"
      by (rule normalized_lcm_two_inter_parity[OF concrete accepts])
    show ?thesis
      unfolding reduced
      using exact by simp
  next
    case rejects: False
    have odd_modulus: "odd m"
      using modulus by simp
    have shifted_accepts: "parity_accepts p (c + m)"
      by (rule parity_accepts_add_odd[OF concrete rejects odd_modulus])
    have reduced:
      "restrict_congruence_rep_by_parity p (Some (c, m)) =
       normalize_congruence_rep (Some (c + m, lcm m 2))"
      using concrete nonzero modulus rejects
      by (cases p) simp_all
    have exact:
      "gamma_congruence_rep
         (normalize_congruence_rep (Some (c + m, lcm m 2))) =
       gamma_parity p ∩ {n. m dvd n - (c + m)}"
      by (rule normalized_lcm_two_inter_parity[
            OF concrete shifted_accepts])
    show ?thesis
      unfolding reduced
      using exact by (simp only: gamma_congruence_rep.simps
          dvd_shift_modulus)
  qed
 qed
qed

lemma restrict_congruence_rep_by_parity_correct:
 assumes normalized: "normalized_congruence_rep r"
 shows
  "gamma_congruence_rep
     (restrict_congruence_rep_by_parity p r) =
   gamma_parity p ∩ gamma_congruence_rep r"
proof (cases p)
 case PBot
 then show ?thesis by simp
next
 case PTop
 then show ?thesis by simp
next
 case destination: PEven
 show ?thesis
 proof (cases r)
  case None
  with destination show ?thesis by simp
 next
  case (Some cm)
  obtain c m where cm: "cm = (c, m)"
    by (cases cm)
  have rep: "r = Some (c, m)"
    using Some cm by simp
  show ?thesis
    unfolding destination rep
    by (rule restrict_concrete_parity_correct) simp
 qed
```

```
next
  case destination: POdd
  show ?thesis
  proof (cases r)
    case None
    with destination show ?thesis by simp
  next
    case (Some cm)
    obtain c m where cm: "cm = (c, m)"
      by (cases cm)
    have rep: "r = Some (c, m)"
      using Some cm by simp
    show ?thesis
      unfolding destination rep
      by (rule restrict_concrete_parity_correct) simp
  qed
qed


lift_definition restrict_congruence_by_parity ::
  "parity => congruence => congruence"
  is restrict_congruence_rep_by_parity
  by (rule normalized_restrict_congruence_rep_by_parity)

declare restrict_congruence_by_parity.rep_eq [simp]

lemma gamma_restrict_congruence_by_parity [simp]:
  "gamma_congruence (restrict_congruence_by_parity p c) =
   gamma_parity p ∩ gamma_congruence c"
proof -
  have normalized:
    "normalized_congruence_rep (Rep_congruence c)"
    by simp
  have raw:
    "gamma_congruence_rep
       (restrict_congruence_rep_by_parity p (Rep_congruence c)) =
     gamma_parity p ∩
     gamma_congruence_rep (Rep_congruence c)"
    by (rule restrict_congruence_rep_by_parity_correct[OF normalized])
  show ?thesis
    unfolding gamma_congruence_def
    using raw by simp
qed

definition ivl_congruence_nonempty ::
  "ivl => congruence => bool" where
  "ivl_congruence_nonempty i c =
   ivl_congruence_rep_nonempty i (Rep_congruence c)"

lemma ivl_congruence_nonempty_correct:
  "ivl_congruence_nonempty i c ⟷
   gamma_ivl i ∩ gamma_congruence c ≠ {}"
proof -
  have normalized:
    "normalized_congruence_rep (Rep_congruence c)"
    by simp
  show ?thesis
    unfolding ivl_congruence_nonempty_def gamma_congruence_def
    by (rule ivl_congruence_rep_nonempty_correct[OF normalized])
qed

definition is_bottom_int_dom :: "'a int_dom_scheme => bool" where
  "is_bottom_int_dom d =
     (¬ ivl_congruence_nonempty
       (intersect_ivl
         (interval_fact_of_sign (int_sign d))
         (int_ivl d))
       (restrict_congruence_by_parity
         (int_parity d)
         (int_congruence d)))"

lemma is_bottom_int_dom_correct:
  "is_bottom_int_dom d ⟷ gamma_int_dom d = {}"
proof -
  let ?i =
    "intersect_ivl
```

```
      (interval_fact_of_sign (int_sign d))
      (int_ivl d)"
  let ?c =
    "restrict_congruence_by_parity
      (int_parity d)
      (int_congruence d)"
  have nonempty:
    "ivl_congruence_nonempty ?i ?c ⟷
     gamma_ivl ?i ∩ gamma_congruence ?c ≠ {}"
    by (rule ivl_congruence_nonempty_correct)
  have restricted:
    "gamma_congruence ?c =
     gamma_parity (int_parity d) ∩
     gamma_congruence (int_congruence d)"
    by simp
  have interval:
    "gamma_ivl ?i =
     gamma_sign (int_sign d) ∩ gamma_ivl (int_ivl d)"
    by (simp only: gamma_intersect_ivl_exact
          gamma_interval_fact_of_sign)
  show ?thesis
    unfolding is_bottom_int_dom_def gamma_int_dom_def
    using nonempty restricted interval by blast
qed
```

Semantic fullness, symmetric to `is_bottom_int_dom` above: `d = top` additionally pins the record's own `more` extension field to its top, which `gamma_int_dom` never inspects, so it is strictly stronger than `gamma_int_dom d = UNIV` at this instance's actual, extension-polymorphic `'a int_dom_scheme` type. Testing each of the four concrete components' own exact fullness instead reads off `gamma_int_dom`'s own definition directly, the same move `is_bottom_int_dom` already makes for emptiness.

```
definition is_top_int_dom :: "'a int_dom_scheme => bool" where
  "is_top_int_dom d ⟷
     is_top_sign (int_sign d) ∧ is_top_ivl (int_ivl d) ∧
     is_top_parity (int_parity d) ∧ is_top_congruence (int_congruence d)"

lemma is_top_int_dom_correct_gamma: "is_top_int_dom d ⟷ gamma_int_dom d = UNIV"
  unfolding is_top_int_dom_def gamma_int_dom_def
  using is_top_sign_correct_gamma is_top_ivl_correct_gamma
        is_top_parity_correct_gamma is_top_congruence_correct_gamma
  by auto

lemma gamma_int_dom_mono:
  assumes ab: "a ≤ b"
  shows "gamma_int_dom a ⊆ gamma_int_dom b"
proof -
  from ab have le:
    "int_sign a ≤ int_sign b"
    "int_ivl a ≤ int_ivl b"
    "int_parity a ≤ int_parity b"
    "int_congruence a ≤ int_congruence b"
    by (simp_all add: less_eq_int_dom_ext_def)
  have S: "gamma_sign (int_sign a) ⊆ gamma_sign (int_sign b)"
    using le(1) unfolding less_eq_sign_def by (rule gamma_sign_mono)
  have I: "gamma_ivl (int_ivl a) ⊆ gamma_ivl (int_ivl b)"
    using le(2) by (rule gamma_ivl_mono)
  have P: "gamma_parity (int_parity a) ⊆ gamma_parity (int_parity b)"
    using le(3) unfolding less_eq_parity_def by (rule gamma_parity_mono)
  have C:
    "gamma_congruence (int_congruence a) ⊆
     gamma_congruence (int_congruence b)"
    using le(4)
    unfolding less_eq_congruence_iff_gamma .
  show ?thesis
    unfolding gamma_int_dom_def using S I P C by auto
qed
```

Constants and printing

`int_dom_of_int` pins every component to one integer, so it concretizes to exactly that singleton; the printer follows Goblint's tuple format.

```
definition int_dom_of_int :: "int => int_dom" where
  "int_dom_of_int n =
     (top :: int_dom)⦇
       int_sign := sign_of_int n,
```

```
      int_ivl := Ivl (Fin n) (Fin n),
      int_parity := parity_of_int n,
      int_congruence := congruence_of_int n
    ⦈"

lemma gamma_int_dom_of_int [simp]:
  "gamma_int_dom (int_dom_of_int n) = {n}"
  by (auto simp: int_dom_of_int_def gamma_int_dom_def
       sign_of_int_gamma parity_of_int_gamma)
```

Goblint's integer-domain tuple prints a value one of its components pins to a single integer as that integer, a value no component constrains as $\top$, and anything else as its components, each named and separated by `;`. The bottom test runs first, so a component that concretizes to one integer pins a live product to exactly that integer.

```
fun ivl_constant :: "ivl ⇒ int option" where
    "ivl_constant (Ivl (Fin l) (Fin u)) = (if l = u then Some l else None)"
  | "ivl_constant _ = None"

definition congruence_constant :: "congruence ⇒ int option" where
  "congruence_constant c =
     (case Rep_congruence c of
        Some (r, m) ⇒ if m = 0 then Some r else None
      | None ⇒ None)"

definition int_dom_constant :: "'a int_dom_scheme ⇒ int option" where
  "int_dom_constant d =
     (if int_sign d = SZero then Some 0
      else case ivl_constant (int_ivl d) of
        Some n ⇒ Some n
      | None ⇒ congruence_constant (int_congruence d))"

lemma gamma_ivl_constant:
  "ivl_constant i = Some n ⟹ gamma_ivl i = {n}"
  by (induction i rule: ivl_constant.induct) (auto split: if_splits)

lemma gamma_congruence_constant:
  "congruence_constant c = Some n ⟹ gamma_congruence c = {n}"
  unfolding congruence_constant_def gamma_congruence_def
  by (auto split: option.splits if_splits)
```

Printing a live product as the integer `int_dom_constant` names loses nothing: that integer is all the product denotes.

```
lemma gamma_int_dom_constant_subset:
  "int_dom_constant d = Some n ⟹ gamma_int_dom d ⊆ {n}"
  using gamma_ivl_constant gamma_congruence_constant
  unfolding int_dom_constant_def gamma_int_dom_def
  by (auto split: if_splits option.splits)

definition string_of_int_dom :: "'a int_dom_scheme ⇒ String.literal" where
  "string_of_int_dom d =
     (if is_bottom_int_dom d then sym_bottom
      else if is_top_int_dom d then sym_top
      else case int_dom_constant d of
        Some n ⇒ string_of_int n
      | None ⇒
          STR ''signs:'' + string_of_sign (int_sign d)
        + STR ''; intervals:'' + string_of_ivl (int_ivl d)
        + STR ''; parities:'' + string_of_parity (int_parity d)
        + STR ''; congruences:'' + string_of_congruence (int_congruence d))"

end
```

2.18 Order_Lattice

```
theory Order_Lattice
  imports "Voblint_Domain.State_Concretization"
    "HOL-Library.Product_Lexorder"
begin
```

Order lattice

`relc` tracks a finite set of known pairwise-ordered variables, `(x, y)` meaning `x` $\leq$ `y` at every store the value describes. No closure: two known facts `x` $\leq$ `y` and `y` $\leq$ `z` do not automatically yield `x` $\leq$ `z` in this carrier. This is deliberately the least amount of relational structure that is still relational (a pair of variables, not one) and not `abs_state` (no `vname` $\Rightarrow$ `'a` function type anywhere in the carrier). Its concretization is a set of stores, not of integers, so it is an `executable_domain` but not a `numeric_domain`.

Carrier and order

```
datatype relc = RelBot | RelC (relc_pairs: "(vname × vname) set")
```

Order is reverse inclusion on the constraint set: more known pairs is more information, hence lower (more precise) in the abstract-interpretation order. `sup` keeps only the pairs both sides agree on.

`RelBot` is a separate explicit constructor rather than `RelC UNIV` (the most-constrained set, "every pair known ordered"): representing `UNIV` forces the code generator to use the `Coset` branch of HOL's executable-set representation, and the stock library does not give every set operation (subset test among them) a code equation for every `Set`/`Coset` combination over an infinite element type such as `vname` – confirmed directly: `RelC UNIV` batch-checked and even unit-tested via `value` cleanly, but running it through the solver raised `exception Match` in the generated code the first time a genuine `Coset`/`Coset` combination arose. Keeping `RelC`'s field always finite avoids the gap entirely: no value this theory ever constructs is a `Coset`.

```
instantiation relc :: bounded_semilattice_sup_bot
begin

fun less_eq_relc :: "relc ⇒ relc ⇒ bool" where
  "less_eq_relc RelBot _ = True"
| "less_eq_relc (RelC _) RelBot = False"
| "less_eq_relc (RelC a) (RelC b) = (b ⊆ a)"

definition less_relc :: "relc ⇒ relc ⇒ bool" where
  "less_relc a b ⟷ a ≤ b ∧ ¬ b ≤ a"

fun sup_relc :: "relc ⇒ relc ⇒ relc" where
  "sup_relc RelBot b = b"
| "sup_relc a RelBot = a"
| "sup_relc (RelC a) (RelC b) = RelC (a ∩ b)"

definition bot_relc :: relc where
  "bot_relc = RelBot"

instance
proof intro_classes
  fix x y z :: relc
  show "x < y ⟷ x ≤ y ∧ ¬ y ≤ x" by (simp add: less_relc_def)
  show "x ≤ x" by (cases x) simp_all
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z" by (cases x; cases y; cases z) auto
  show "x ≤ y ⟹ y ≤ x ⟹ x = y" by (cases x; cases y) auto
  show "x ≤ x ⊔ y" by (cases x; cases y) auto
  show "y ≤ x ⊔ y" by (cases x; cases y) auto
  show "y ≤ x ⟹ z ≤ x ⟹ y ⊔ z ≤ x" by (cases x; cases y; cases z) auto
  show "bot ≤ x" by (cases x) (simp_all add: bot_relc_def)
qed

end

lemma bot_relc_eq [simp]: "(⊥ :: relc) = RelBot"
  by (simp add: bot_relc_def)
```

$\top$ is the empty constraint set: vacuously true of every pair, so its concretization is `UNIV` (`gamma_relc_top`).

```
instantiation relc :: order_top
begin

definition top_relc :: relc where
  "top_relc = RelC {}"
```

```
instance by intro_classes (case_tac a; simp add: top_relc_def)

end
```

The vendored TD solver's `TD_side_upd_rule` locale fixes its equation value type at sort `{bounded_semilattice_sup_bot, warrowing}` uniformly – every update rule in the solver menu needs it, not only the `warrow` entry, even on a loop-free equation system where widening is never actually invoked. `widen = sup` reuses the join laws already proved above; `narrow a b = b` is the simplest sound choice ("accept the incoming value, refine nothing") – consistent with this carrier's own no-closure, no-normalization scope.

```
instantiation relc :: warrowing
begin

definition widen_relc :: "relc ⇒ relc ⇒ relc" where
  "widen_relc a b = a ⊔ b"

definition narrow_relc :: "relc ⇒ relc ⇒ relc" where
  "narrow_relc a b = b"

instance
proof intro_classes
  fix a b :: relc
  show "a ≤ a ∇ b" by (simp add: widen_relc_def)
  show "b ≤ a ∇ b" by (simp add: widen_relc_def)
  show "b ≤ a ⟹ b ≤ a Δ b" by (simp add: narrow_relc_def)
  show "b ≤ a ⟹ a Δ b ≤ a" by (simp add: narrow_relc_def)
qed

end
```

Concretization

A set of pairs denotes the stores satisfying `s x <= s y` for each pair; `RelBot` denotes no store. Concretization is monotone in the order.

```
fun gamma_relc :: "relc ⇒ store set" where
  "gamma_relc RelBot = {}"
| "gamma_relc (RelC ps) = {s. ∀(x, y) ∈ ps. s x ≤ s y}"

adhoc_overloading gamma_S == gamma_relc

lemma gamma_relc_top [simp]: "γ (⊤ :: relc) = UNIV"
  unfolding top_relc_def by simp

lemma gamma_relc_mono:
  fixes d d' :: relc
  assumes "d ≤ d'"
  shows "γ d ⊆ γ d'"
  using assms by (cases d; cases d') auto
```

Executable membership reader. `RelBot` answers `True` for every pair: its concretization is empty, so every fact holds of it vacuously, and it never needs to materialize `UNIV`.

```
fun relc_has :: "vname ⇒ vname ⇒ relc ⇒ bool" where
  "relc_has x y RelBot = True"
| "relc_has x y (RelC ps) = ((x, y) ∈ ps)"
```

Forgetting a variable

The one operation every imprecise fallback reuses.

```
fun forget_relc :: "vname ⇒ relc ⇒ relc" where
  "forget_relc x RelBot = RelBot"
| "forget_relc x (RelC ps) = RelC {(a, b) ∈ ps. a ≠ x ∧ b ≠ x}"

lemma forget_relc_sound[intro]:
  assumes "s ∈ γ d"
  shows "s(x := v) ∈ γ (forget_relc x d)"
  using assms by (cases d) auto
```

Executable operations

The notation follows Goblint's two-variable equality domain, which prints a conjunction `{x=y ∧ z=y}` and its extremes as ⊥ and ⊤; here each conjunct is an ordering `x≤y`. `vname × vname` is `linorder` (`Product_Lexorder`), so `sorted_list_of_set` gives a deterministic enumeration.

```
fun string_of_pairs :: "(vname × vname) list ⇒ String.literal" where
  "string_of_pairs [] = STR ''''"
| "string_of_pairs [(x, y)] = x + sym_le + y"
| "string_of_pairs ((x, y) # p # ps) =
      x + sym_le + y + STR '' '' + sym_and + STR '' '' + string_of_pairs (p # ps)"

instantiation relc :: executable_domain
begin

definition is_empty_relc :: "relc ⇒ bool" where
  "is_empty_relc d ⟷ d = RelBot"

definition to_string_relc :: "relc ⇒ String.literal" where
  "to_string_relc d =
     (case d of
        RelBot ⇒ sym_bottom
      | RelC ps ⇒
          (if ps = {} then sym_top
           else STR ''{'' + string_of_pairs (sorted_list_of_set ps) + STR ''}''))"

instance ..

end

lemma is_empty_relc_gamma: "is_empty d ⟷ γ d = {}" for d :: relc
proof (cases d)
  case (RelC ps)
  then have "(λ_. 0) ∈ γ d" by auto
  with RelC show ?thesis by (auto simp: is_empty_relc_def)
qed (simp add: is_empty_relc_def)

lemma exact_emptiness_relc: "exact_emptiness (is_empty :: relc ⇒ bool) gamma_relc"
  by (rule exact_emptinessI) (rule is_empty_relc_gamma)

end
```

Chapter 3

Voblint_Solver

3.1 Strategy_Program_Fold

```
theory Strategy_Program_Fold
  imports Strategy_Tree_Program
begin
```

Folding a right-hand side from contribution programs

An equation's right-hand side is assembled from a list of contributions — one per incoming contribution, such as an intra predecessor, a call return, or a routed activation hook — joined into one answer. Each contribution is a *program*, not a finished tree: the fold sequences them with $\gg=$, joining each answer into a running accumulator, and the result is again a program. Nothing is compiled here. A generator compiles once, at the end, where the solver takes over.

That is why the characterizations below can speak one vocabulary on both sides. `traverse_program`, `sides_of_program` and `dep_program` ask a contribution the same questions they ask the fold, and each hypothesis $\forall$`p` $\in$ `set ps. sp_wf p` is what makes asking them of a program well posed at all: an arbitrary program may ignore or duplicate its continuation, and then what it publishes would depend on what follows it.

The fold never emits a `Side` itself: a fold of Side-free contributions stays Side-free, which is what lets a generator publish once, after the whole fold, instead of once per contribution — several updates to one key during a single right-hand-side evaluation otherwise change what the vendored solver's per-origin update rule widens against.

What a caller varies is *how much of each answer accumulates*. The fold therefore takes a projection `prj` into the accumulator and an embedding `emb` back out. At the identity pair the whole answer accumulates. A D/G instance instead projects the local half into the accumulator and embeds the result as `DG d bot`; the global half of a contribution simply does not reach the node's answer, and a contribution that has one to share must publish it itself through `Side`. Nothing here makes that happen.

The two parameters are used as a projection and a final embedding, but the fold assumes no inverse or homomorphism law relating them. Order preservation is required only where it is actually used, by the environment-monotonicity results; the publication and dependency characterizations hold for arbitrary `prj` and `emb`.

Join-folds over a list

$\bigsqcup$`x`$\leftarrow$`xs. f x` joins `f x` over the list `xs`; the empty join is $\bot$. It is built like HOL's `sum_list`: `join_list` is the list fold of the commutative monoid `(`$\sqcup$`, `$\bot$`)`, and the binder maps before it folds.

```
context bounded_semilattice_sup_bot
begin

sublocale join_list: comm_monoid_list sup bot
  defines join_list = join_list.F ..

end

open_bundle join_list_syntax
begin

syntax
  "_join_list" :: "pttrn ⇒ 'a list ⇒ 'b ⇒ 'b"
    (‹(‹indent=3 notation=‹binder ⨆››⨆_←_. _)› [0, 51, 10] 10)
syntax_consts
  "_join_list" == join_list
translations
  "⨆x←xs. b" == "CONST join_list (CONST map (λx. b) xs)"

end
```

Every characterization below states the fold as a list join over already-computed values, so its consumers keep needing the same handful of facts about that: the seed of a right fold can be moved out, a bound against the join is a bound against every element, one element is below the join, only the element set matters, a pointwise bound lifts, two lists that agree elementwise agree, and an all-$\bot$ join is $\bot$.

They are collected here, ahead of the fold they serve, rather than restated per generator. `foldr_sup_seed_swap` is the one with content — it needs only `semilattice_sup` — and `foldr_join_list` and `foldr_join_seed_out` read a right fold with a $\bot$ or an arbitrary seed as a list join.

```
lemma foldr_sup_seed_swap:
  fixes h :: "'a ⇒ 'b::semilattice_sup"
  shows "foldr (λt acc'. h t ⊔ acc') ts (acc ⊔ x) = x ⊔ foldr (λt acc'. h t ⊔ acc') ts acc"
  by (induction ts arbitrary: acc) (simp_all add: sup_assoc sup_commute sup_left_commute)
```

```
lemma foldr_join_list:
  fixes h :: "'a ⇒ 'b::bounded_semilattice_sup_bot"
  shows "foldr (λt acc'. h t ⊔ acc') ts bot = (⨆t←ts. h t)"
  by (induction ts) simp_all

lemma foldr_join_seed_out:
  fixes h :: "'a ⇒ 'b::bounded_semilattice_sup_bot"
  shows "foldr (λt acc'. h t ⊔ acc') ts a = a ⊔ (⨆t←ts. h t)"
  using foldr_sup_seed_swap[of h ts bot a] by (simp add: foldr_join_list)

lemma foldr_sup_le_iff [simp]:
  fixes h :: "'a ⇒ 'b::bounded_semilattice_sup_bot"
  shows "(⨆t←xs. h t) ≤ y ⟷ (∀x ∈ set xs. h x ≤ y)"
  by (induction xs) auto

lemma foldr_sup_member_le [intro]:
  fixes h :: "'a ⇒ 'b::bounded_semilattice_sup_bot"
  assumes "x ∈ set xs"
  shows "h x ≤ (⨆t←xs. h t)"
  using assms by (induction xs) (auto intro: le_supI2)

lemma foldr_sup_mono:
  fixes f g :: "'a ⇒ 'b::bounded_semilattice_sup_bot"
  assumes "⋀x. x ∈ set xs ⟹ f x ≤ g x"
  shows "(⨆t←xs. f t) ≤ (⨆t←xs. g t)"
  using assms by (auto intro: order_trans[OF _ foldr_sup_member_le])

lemma foldr_sup_set_cong:
  fixes h :: "'a ⇒ 'b::bounded_semilattice_sup_bot"
  assumes eq: "set xs = set ys"
  shows "(⨆t←xs. h t) = (⨆t←ys. h t)"
  by (rule order_antisym) (auto simp: eq)
```

Two lists whose elements agree under their respective observations have the same join. This is what lets a correspondence proof relate two generators' contribution lists elementwise instead of reasoning about the folds themselves.

```
lemma foldr_sup_list_all2_cong:
  fixes f :: "'a ⇒ 'c::bounded_semilattice_sup_bot"
  assumes "list_all2 (λx y. f x = g y) xs ys"
  shows "(⨆x←xs. f x) = (⨆y←ys. g y)"
  using assms by (induction rule: list_all2_induct) simp_all
```

The two ways such a correspondence is built and consumed: establishing it between two maps over one list, and reading it as an equality of unions. Neither mentions a fold, a tree, or a graph.

```
lemma list_all2_map_diag:
  "(⋀x. x ∈ set xs ⟹ P (f x) (g x)) ⟹ list_all2 P (map f xs) (map g xs)"
  by (induction xs) simp_all

lemma list_all2_Union_eq:
  assumes "list_all2 (λa b. f a = g b) xs ys"
  shows "(⋃x∈set xs. f x) = (⋃y∈set ys. g y)"
  using assms by (induction rule: list_all2_induct) auto

lemma foldr_sup_bot_of_all_bot:
  fixes L :: "'a list" and h :: "'a ⇒ 'b::bounded_semilattice_sup_bot"
  assumes "⋀x. x ∈ set L ⟹ h x = bot"
  shows "(⨆x←L. h x) = bot"
  using assms by (induction L) simp_all
```

Folding contributions into one right-hand side

Several contribution programs, one right-hand side: run each in list order, join its answer into the accumulator, and answer with the accumulator. `prj` and `emb` keep the fold generic in which part of an answer accumulates – that is what later lets the D/G layer join the local half while the global half travels as a side effect.

```
fun fold_rhs_program_projected ::
  "('a ⇒ 'b) ⇒ ('b ⇒ 'a) ⇒ 'b::semilattice_sup
   ⇒ ('k, 'g, 'a, 'a) strategy_program list
   ⇒ ('k, 'g, 'a, 'a) strategy_program"
where
  "fold_rhs_program_projected prj emb acc [] = sp_return (emb acc)"
| "fold_rhs_program_projected prj emb acc (p # ps) =
     do {
       res ← p;
```

```
      fold_rhs_program_projected prj emb (acc ⊔ prj res) ps
    }"
```

The accumulator read declaratively off a valuation, rather than run. Every answer characterization below is stated through it.

```
fun fold_acc_projected ::
  "('a::bounded_semilattice_sup_bot ⇒ 'b::semilattice_sup) ⇒ 'b
   ⇒ ('k + 'g ⇒ 'a) ⇒ ('k, 'g, 'a, 'a) strategy_program list ⇒ 'b"
where
  "fold_acc_projected prj acc σ [] = acc"
| "fold_acc_projected prj acc σ (p # ps) =
    fold_acc_projected prj (acc ⊔ prj (traverse_program p σ)) σ ps"
```

Declarative characterizations

Contributions are traversed in list order: each answer joins the running accumulator before the next program runs. The publication and dependency characterizations forget that order, the underlying operation being commutative — a set union for `dep_aux`, a join seeded at `bot` for `sides_of_rhs` — and neither sees `prj` or `emb` at all: what a fold publishes and reads is its contributions' business, not its accumulator's. That is also why both are independent of the accumulator it starts from.

```
theorem traverse_fold_rhs_program_projected_char:
  assumes "∀p ∈ set ps. sp_wf p"
  shows "traverse_program (fold_rhs_program_projected prj emb acc ps) σ
     = emb (fold_acc_projected prj acc σ ps)"
  using assms
  by (induction ps arbitrary: acc)
     (simp_all add: sp_compile_def sp_compile_with_bind traverse_rhs_sp_wf)

theorem sides_of_program_fold_rhs_program_projected_char:
  fixes ps :: "('x, 'g, 'd::bounded_semilattice_sup_bot, 'd) strategy_program list"
  assumes "∀p ∈ set ps. sp_wf p"
  shows "sides_of_program (fold_rhs_program_projected prj emb acc ps) σ z
     = (⨆p←ps. sides_of_program p σ z)"
  using assms
  by (induction ps arbitrary: acc)
     (auto simp add: sp_compile_def sp_compile_with_bind sides_of_rhs_sp_wf traverse_rhs_sp_wf)

theorem dep_program_fold_rhs_program_projected_char:
  assumes "∀p ∈ set ps. sp_wf p"
  shows "dep_program σ (fold_rhs_program_projected prj emb acc ps)
     = (⋃p∈set ps. dep_program σ p)"
  using assms
  by (induction ps arbitrary: acc)
     (auto simp add: sp_compile_def sp_compile_with_bind dep_aux_sp_wf traverse_rhs_sp_wf)

lemma fold_acc_projected_as_foldr:
  fixes prj :: "'a::bounded_semilattice_sup_bot ⇒ 'b::bounded_semilattice_sup_bot"
  shows "fold_acc_projected prj acc σ ps
           = acc ⊔ (⨆p←ps. prj (traverse_program p σ))"
  by (induction ps arbitrary: acc) (simp_all add: sup_assoc)

lemma fold_acc_projected_acc_mono:
  fixes prj :: "'a::bounded_semilattice_sup_bot ⇒ 'b::bounded_semilattice_sup_bot"
  assumes "acc1 ≤ acc2"
  shows "fold_acc_projected prj acc1 σ ts ≤ fold_acc_projected prj acc2 σ ts"
  using assms by (simp add: fold_acc_projected_as_foldr le_supI1)
```

Monotonicity in the *environment* rather than the accumulator. This is where `prj` and `emb` stop being arbitrary: a fold inherits its contributions' monotonicity only if the two also preserve order. Publications and dependencies need no such hypothesis, since neither passes through them.

```
lemma fold_acc_projected_env_mono:
  fixes prj :: "'a::bounded_semilattice_sup_bot ⇒ 'b::bounded_semilattice_sup_bot"
  assumes prj_mono: "mono prj"
    and progs_mono: "⋀p. p ∈ set ps ⟹ traverse_program p σ1 ≤ traverse_program p σ2"
  shows "fold_acc_projected prj acc σ1 ps ≤ fold_acc_projected prj acc σ2 ps"
  unfolding fold_acc_projected_as_foldr
  by (rule sup_mono[OF order_refl])
     (rule foldr_sup_mono, rule monoD[OF prj_mono, OF progs_mono])

lemma traverse_fold_rhs_program_projected_mono:
  fixes prj :: "'a::bounded_semilattice_sup_bot ⇒ 'b::bounded_semilattice_sup_bot"
  assumes wf: "∀p ∈ set ps. sp_wf p"
```

```
    and prj_mono: "mono prj"
    and emb_mono: "mono emb"
    and progs_mono: "⋀p. p ∈ set ps ⟹ traverse_program p σ1 ≤ traverse_program p σ2"
  shows "traverse_program (fold_rhs_program_projected prj emb acc ps) σ1
       ≤ traverse_program (fold_rhs_program_projected prj emb acc ps) σ2"
  unfolding traverse_fold_rhs_program_projected_char[OF wf]
  by (rule monoD[OF emb_mono])
     (rule fold_acc_projected_env_mono[OF prj_mono progs_mono])

lemma sides_of_program_fold_rhs_program_projected_mono:
  fixes ps :: "('x, 'g, 'd::bounded_semilattice_sup_bot, 'd) strategy_program list"
  assumes wf: "∀p ∈ set ps. sp_wf p"
    and progs_mono: "⋀p. p ∈ set ps ⟹ sides_of_program p σ1 ≤ sides_of_program p σ2"
  shows "sides_of_program (fold_rhs_program_projected prj emb acc ps) σ1
       ≤ sides_of_program (fold_rhs_program_projected prj emb acc ps) σ2"
  unfolding le_fun_def sides_of_program_fold_rhs_program_projected_char[OF wf]
  using progs_mono
  by (fastforce simp: le_fun_def
        intro: order_trans[OF _ foldr_sup_member_le])

lemma sides_of_program_fold_rhs_program_projected_acc_indep:
  fixes ps :: "('x, 'g, 'd::bounded_semilattice_sup_bot, 'd) strategy_program list"
  assumes wf: "∀p ∈ set ps. sp_wf p"
  shows "sides_of_program (fold_rhs_program_projected prj emb acc1 ps) σ
       = sides_of_program (fold_rhs_program_projected prj emb acc2 ps) σ"
  by (simp add: fun_eq_iff sides_of_program_fold_rhs_program_projected_char[OF wf])

lemma dep_program_fold_rhs_program_projected_acc_indep:
  assumes wf: "∀p ∈ set ps. sp_wf p"
  shows "dep_program σ (fold_rhs_program_projected prj emb acc1 ps)
     = dep_program σ (fold_rhs_program_projected prj emb acc2 ps)"
  by (simp add: dep_program_fold_rhs_program_projected_char[OF wf])
```

Accumulating the whole answer

The identity instance: nothing is projected away, so the accumulator has the answer type and every contribution joins into the node's answer whole.

```
lemma sp_wf_fold_rhs_program_projected [intro]:
  "∀p ∈ set ps. sp_wf p ⟹ sp_wf (fold_rhs_program_projected prj emb acc ps)"
  by (induction ps arbitrary: acc) auto

definition fold_rhs_program ::
  "'a::semilattice_sup
   ⇒ ('k, 'g, 'a, 'a) strategy_program list
   ⇒ ('k, 'g, 'a, 'a) strategy_program"
where
  "fold_rhs_program acc ps = fold_rhs_program_projected id id acc ps"

lemma sp_wf_fold_rhs_program [intro]:
  "∀p ∈ set ps. sp_wf p ⟹ sp_wf (fold_rhs_program acc ps)"
  unfolding fold_rhs_program_def by (rule sp_wf_fold_rhs_program_projected)

lemma fold_rhs_program_simps [simp, code]:
  "fold_rhs_program acc [] = sp_return acc"
  "fold_rhs_program acc (p # ps) =
     do { res ← p; fold_rhs_program (acc ⊔ res) ps }"
  by (simp_all add: fold_rhs_program_def)
```

The answer as a right fold. `fold_rhs_program` threads the accumulator in evaluation order, which is a left fold; proofs about a node's contributions are usually already phrased as a right fold, and `traverse_rhs` only ever joins, so the two agree.

```
theorem traverse_fold_rhs_program_char_foldr:
  assumes "∀p ∈ set ps. sp_wf p"
  shows "traverse_program (fold_rhs_program acc ps) σ
     = foldr (λp acc'. traverse_program p σ ⊔ acc') ps acc"
  using assms
  by (induction ps arbitrary: acc)
     (simp_all add: sp_compile_def sp_compile_with_bind foldr_sup_seed_swap
        traverse_rhs_sp_wf)

theorem sides_of_program_fold_rhs_program_char:
  fixes ps :: "('x,'g,'d::bounded_semilattice_sup_bot,'d) strategy_program list"
  assumes wf: "∀p ∈ set ps. sp_wf p"
  shows "sides_of_program (fold_rhs_program acc ps) σ z
```

```
    = (⨆p←ps. sides_of_program p σ z)"
  unfolding fold_rhs_program_def
  by (rule sides_of_program_fold_rhs_program_projected_char[OF wf])

theorem dep_program_fold_rhs_program_char:
  assumes wf: "∀p ∈ set ps. sp_wf p"
  shows "dep_program σ (fold_rhs_program acc ps) = (⋃p∈set ps. dep_program σ p)"
  unfolding fold_rhs_program_def
  by (rule dep_program_fold_rhs_program_projected_char[OF wf])
```

Dependency-property preservation

A fold of programs whose query set is (respectively environment-independent, monotone in the environment) is itself environment-independent (monotone) – the list-level analogue of `env_indep_deps`/`mono_tree_deps`'s own single-tree closure facts, and what lets a generator's whole-node dependency obligation reduce to a per-hook one.

Both read straight off `dep_program_fold_rhs_program_projected_char`: a fold's dependencies are the union of its contributions', so whatever closure property each contribution has, the union inherits.

```
lemma env_indep_deps_fold_rhs_program_projected:
  assumes wf: "∀p ∈ set ps. sp_wf p"
    and "⋀p. p ∈ set ps ⟹ env_indep_deps (sp_compile p)"
  shows "env_indep_deps (sp_compile (fold_rhs_program_projected prj emb acc ps))"
  using assms
  unfolding env_indep_deps_def
  by (auto simp add: dep_program_fold_rhs_program_projected_char[OF wf]; blast)

lemma mono_tree_deps_fold_rhs_program_projected:
  assumes wf: "∀p ∈ set ps. sp_wf p"
    and "⋀p. p ∈ set ps ⟹ mono_tree_deps (sp_compile p)"
  shows "mono_tree_deps (sp_compile (fold_rhs_program_projected prj emb acc ps))"
  using assms
  unfolding mono_tree_deps_def
  by (fastforce simp: dep_program_fold_rhs_program_projected_char[OF wf])

end
```

3.2 Strategy_Tree_Program

```
theory Strategy_Tree_Program
  imports "TD.Basics_side" "HOL-Library.Monad_Syntax"
begin
```

A typed frontend over the homogeneous vendor tree

`strategy_tree` fixes query results and terminal answers to the same carrier `'d` and therefore does not support a polymorphic bind. `strategy_program` supplies a continuation-based typed interface: a program may produce any result type `'a` while its generated vendor tree continues to use `'d` for solver values and contributions. `sp_compile_with` encodes the final result into `'d` and produces the vendor tree consumed by the solver.

```
type_synonym ('x,'g,'d,'a) strategy_program =
  "('a ⇒ ('x,'g,'d) strategy_tree) ⇒ ('x,'g,'d) strategy_tree"

definition sp_return :: "'a ⇒ ('x,'g,'d,'a) strategy_program" where
  "sp_return a k = k a"

definition sp_bind ::
  "('x,'g,'d,'a) strategy_program
   ⇒ ('a ⇒ ('x,'g,'d,'b) strategy_program)
   ⇒ ('x,'g,'d,'b) strategy_program" where
  "sp_bind m f k = m (λv. f v k)"

definition sp_read_local :: "'x ⇒ ('x,'g,'d,'d) strategy_program" where
  "sp_read_local x k = QueryL x k"

definition sp_read_global :: "'g ⇒ ('x,'g,'d,'d) strategy_program" where
  "sp_read_global g k = QueryG g k"

definition sp_publish :: "'g ⇒ 'd ⇒ ('x,'g,'d,unit) strategy_program" where
  "sp_publish g d k = Side g d (k ())"
```

`sp_read_at` reads an unknown in the solver's combined address space `'x + 'g`, dispatching `Inl` to `sp_read_local` and `Inr` to `sp_read_global` instead of fixing the constructor at the call site.

```
fun sp_read_at :: "'x + 'g ⇒ ('x,'g,'d,'d) strategy_program" where
  "sp_read_at (Inl x) = sp_read_local x"
| "sp_read_at (Inr g) = sp_read_global g"

adhoc_overloading Monad_Syntax.bind == sp_bind
```

`sp_read_local`/`sp_read_global`/`sp_publish` build a program one effect at a time. `sp_lift_tree` instead embeds an already-built vendor tree – the shape a backend combinator produces by folding several contribution trees into one – as the program that runs it and continues, by recursing over the tree's own constructors directly: an `Answer` becomes a pure result, a `QueryL`/`QueryG` becomes a read continuing into the lifted subtree, a `Side` becomes a published contribution continuing into the lifted subtree. Defined directly against the constructors, its primitive-recursion equations are already the normal form needed by proofs about vendor trees.

```
primrec sp_lift_tree :: "('x,'g,'d) strategy_tree ⇒ ('x,'g,'d,'d) strategy_program" where
  "sp_lift_tree (Answer d)   k = k d"
| "sp_lift_tree (QueryL x f) k = QueryL x (λd. sp_lift_tree (f d) k)"
| "sp_lift_tree (QueryG g f) k = QueryG g (λd. sp_lift_tree (f d) k)"
| "sp_lift_tree (Side g d t) k = Side g d (sp_lift_tree t k)"

lemma traverse_rhs_sp_lift_tree [simp]:
  "traverse_rhs (sp_lift_tree t k) σ = traverse_rhs (k (traverse_rhs t σ)) σ"
  by (induction t arbitrary: k) (auto intro: rangeI)

lemma dep_aux_sp_lift_tree [simp]:
  "dep_aux σ (sp_lift_tree t k) = dep_aux σ t ∪ dep_aux σ (k (traverse_rhs t σ))"
  by (induction t arbitrary: k) (auto intro: rangeI)

lemma sides_of_rhs_sp_lift_tree [simp]:
  fixes t :: "('x, 'g, 'd::bounded_semilattice_sup_bot) strategy_tree"
  shows "sides_of_rhs (sp_lift_tree t k) σ
         = sides_of_rhs t σ ⊔ sides_of_rhs (k (traverse_rhs t σ)) σ"
  by (induction t arbitrary: k) (auto simp: Let_def sup_fun_def fun_upd_def ac_simps intro: rangeI)
```

`traverse_rhs_sp_lift_tree_mono`: environment-monotonicity of the answer `sp_lift_tree` produces, not of its dependency set – if `t` is monotone in the environment, every continuation `k v` is monotone in the environment, and `k` is monotone in the value it receives, then `sp_lift_tree t k` is monotone in the environment.

```
lemma traverse_rhs_sp_lift_tree_mono:
```

```
  fixes t :: "('x, 'g, 'd::{order,bot}) strategy_tree"
  assumes t_mono:
    "⋀σ1 σ2. σ1 ≤ σ2 ⟹ traverse_rhs t σ1 ≤ traverse_rhs t σ2"
  assumes k_mono_env:
    "⋀v σ1 σ2. σ1 ≤ σ2 ⟹ traverse_rhs (k v) σ1 ≤ traverse_rhs (k v) σ2"
  assumes k_mono_val:
    "⋀σ v1 v2. v1 ≤ v2 ⟹ traverse_rhs (k v1) σ ≤ traverse_rhs (k v2) σ"
  assumes le: "σ1 ≤ σ2"
  shows "traverse_rhs (sp_lift_tree t k) σ1 ≤ traverse_rhs (sp_lift_tree t k) σ2"
proof -
  have "traverse_rhs (k (traverse_rhs t σ1)) σ1
      ≤ traverse_rhs (k (traverse_rhs t σ1)) σ2"
    by (rule k_mono_env[OF le])
  also have "...
      ≤ traverse_rhs (k (traverse_rhs t σ2)) σ2"
    by (rule k_mono_val[OF t_mono[OF le]])
  finally show ?thesis
    by simp
qed
```

`mono_tree_deps` gives the generally applicable composition rule: the continuation may query more unknowns as its received value grows. `env_indep_deps` is also preserved, under the stronger condition that the continuation's dependency set is independent of both its argument and the environment.

`sp_compile` only accepts a program whose answer is already `'d`. `sp_compile_with encode` generalizes that to any `'a`, packing the final answer through `encode`; `sp_compile` is its `id` specialization.

```
definition sp_compile_with ::
  "('a ⇒ 'd) ⇒ ('x,'g,'d,'a) strategy_program ⇒ ('x,'g,'d) strategy_tree" where
  "sp_compile_with encode p = p (Answer o encode)"

definition sp_compile :: "('x,'g,'d,'d) strategy_program ⇒ ('x,'g,'d) strategy_tree" where
  "sp_compile p = sp_compile_with id p"
```

Compiling programs to vendor trees

One rewrite per program constructor, each `simp`, so compiling a concrete program is normalization rather than proof: the simplifier builds the tree from the program's own syntax, and a transfer written against the program layer never has to name a tree constructor.

```
lemma sp_compile_with_bind_read_local [simp]:
  "sp_compile_with encode ((sp_read_local x) ≫= f) = QueryL x (λd. sp_compile_with encode (f d))"
  by (simp add: sp_compile_with_def sp_bind_def sp_read_local_def)

lemma sp_compile_with_bind_read_global [simp]:
  "sp_compile_with encode ((sp_read_global g) ≫= f) = QueryG g (λd. sp_compile_with encode (f d))"
  by (simp add: sp_compile_with_def sp_bind_def sp_read_global_def)

lemma sp_compile_with_bind_publish [simp]:
  "sp_compile_with encode ((sp_publish g d) ≫= f) = Side g d (sp_compile_with encode (f ()))"
  by (simp add: sp_compile_with_def sp_bind_def sp_publish_def)

lemma sp_compile_with_return [simp]:
  "sp_compile_with encode (sp_return a) = Answer (encode a)"
  by (simp add: sp_compile_with_def sp_return_def)

lemma sp_compile_with_read_local [simp]:
  "sp_compile_with encode (sp_read_local x) = QueryL x (Answer o encode)"
  by (simp add: sp_compile_with_def sp_read_local_def)

lemma sp_compile_with_read_global [simp]:
  "sp_compile_with encode (sp_read_global g) = QueryG g (Answer o encode)"
  by (simp add: sp_compile_with_def sp_read_global_def)

lemma sp_compile_with_publish [simp]:
  "sp_compile_with encode (sp_publish g d) = Side g d (Answer (encode ()))"
  by (simp add: sp_compile_with_def sp_publish_def)

lemma sp_compile_with_bind_read_at [simp]:
  "sp_compile_with encode ((sp_read_at src) ≫= f) =
     (case src of Inl x ⇒ QueryL x (λd. sp_compile_with encode (f d))
                | Inr g ⇒ QueryG g (λd. sp_compile_with encode (f d)))"
  by (cases src) simp_all

lemma sp_compile_with_read_at [simp]:
  "sp_compile_with encode (sp_read_at src) =
```

```
    (case src of Inl x ⇒ QueryL x (Answer o encode) | Inr g ⇒ QueryG g (Answer o encode))"
  by (cases src) simp_all

lemma sp_compile_sp_lift_tree [simp]: "sp_compile (sp_lift_tree t) = t"
  by (induction t) (simp_all add: sp_compile_def sp_compile_with_def)
```

Programs that run their continuation once

A `('x, 'g, 'd, 'a) strategy_program` is any function from a continuation to a tree, so nothing stops one from ignoring its continuation, or from using it twice. Such a program has no semantics the tree vocabulary can describe: what it publishes and what it reads would depend on the continuation it is handed.

`sp_wf p` names the programs that do not do this. It says `p` commutes with grafting: continuing into `k` and then grafting `h` onto every leaf of the result is the same as continuing into `k` already grafted. A program that dropped its continuation would lose `h` on the left and keep it on the right; one that used it twice would duplicate the grafting. Everything the generator builds runs its continuation once, and this is the condition under which asking a program what it publishes is a well-posed question at all.

Stating it this way rather than as `p = sp_lift_tree (sp_compile p)` is what lets it apply to a transfer, whose answer type is the analysis' own `'dl` rather than the solver's `'d`: there is no tree whose leaves carry a `'dl`, so a transfer has no compilation to be recoverable from, but it commutes with grafting all the same. For a program that does answer `'d` the two agree, which is `sp_wfD`.

```
lemma sp_lift_tree_sp_lift_tree:
  "sp_lift_tree (sp_lift_tree t g) k = sp_lift_tree t (λv. sp_lift_tree (g v) k)"
  by (induction t) simp_all

definition sp_wf :: "('x,'g,'d,'a) strategy_program ⇒ bool" where
  "sp_wf p ⟷ (∀k h. p (λv. sp_lift_tree (k v) h) = sp_lift_tree (p k) h)"

lemma sp_wfI:
  "(⋀k h. p (λv. sp_lift_tree (k v) h) = sp_lift_tree (p k) h) ⟹ sp_wf p"
  by (simp add: sp_wf_def)

lemma sp_wf_graft:
  "sp_wf p ⟹ p (λv. sp_lift_tree (k v) h) = sp_lift_tree (p k) h"
  by (simp add: sp_wf_def)
```

The homogeneous reading: a program answering the solver's own value type is recoverable from the tree it compiles to. This is the form every consumer below uses, and `sp_compile` only accepts such a program in the first place.

```
lemma sp_wfD:
  assumes "sp_wf p"
  shows "p k = sp_lift_tree (sp_compile p) k"
  using sp_wf_graft[OF assms, where k = Answer and h = k]
  by (simp add: sp_compile_def sp_compile_with_def)
```

Closure under the constructors a generator actually builds with, so a well-formedness side condition discharges by `simp` at the use site instead of being threaded through as a hypothesis.

```
lemma sp_wf_lift_tree [intro, simp]: "sp_wf (sp_lift_tree t)"
  by (simp add: sp_wf_def sp_lift_tree_sp_lift_tree)

lemma sp_wf_return [intro, simp]: "sp_wf (sp_return a)"
  by (simp add: sp_wf_def sp_return_def)

lemma sp_wf_read_local [intro, simp]: "sp_wf (sp_read_local x)"
  by (simp add: sp_wf_def sp_read_local_def)

lemma sp_wf_read_global [intro, simp]: "sp_wf (sp_read_global g)"
  by (simp add: sp_wf_def sp_read_global_def)

lemma sp_wf_publish [intro, simp]: "sp_wf (sp_publish g d)"
  by (simp add: sp_wf_def sp_publish_def)

lemma sp_wf_read_at [intro, simp]: "sp_wf (sp_read_at src)"
  by (cases src) simp_all

lemma sp_wf_bind [intro]:
  assumes "sp_wf p" and "⋀v. sp_wf (f v)"
  shows "sp_wf (p ⤜ f)"
proof (rule sp_wfI)
  fix k h
  have "(p ⤜ f) (λv. sp_lift_tree (k v) h) = p (λv. sp_lift_tree (f v k) h)"
```

```
    by (simp add: sp_bind_def sp_wf_graft[OF assms(2)])
  also have "... = sp_lift_tree (p (λv. f v k)) h"
    by (rule sp_wf_graft[OF assms(1)])
  finally show "(p ⤜ f) (λv. sp_lift_tree (k v) h) = sp_lift_tree ((p ⤜ f) k) h"
    by (simp add: sp_bind_def)
qed
```

Post-composing a program's answer with an encoding, so a transfer answering `'dl` becomes a contribution answering the packed `('dl,'dg) dg_state` the generator folds. It is `sp_compile_with`'s program-level counterpart: compiling it is compiling through the same encoding.

```
definition sp_map ::
  "('a ⇒ 'b) ⇒ ('x,'g,'d,'a) strategy_program ⇒ ('x,'g,'d,'b) strategy_program"
where
  "sp_map e p = (λk. p (λv. k (e v)))"

lemma sp_map_apply [simp]: "sp_map e p k = p (λv. k (e v))"
  by (simp add: sp_map_def)

lemma sp_compile_with_sp_map [simp]:
  "sp_compile_with e' (sp_map e p) = sp_compile_with (e' ∘ e) p"
  by (simp add: sp_map_def sp_compile_with_def comp_def)

lemma sp_compile_sp_map [simp]: "sp_compile (sp_map e p) = sp_compile_with e p"
  by (simp add: sp_compile_def sp_compile_with_def comp_def)

lemma sp_wf_map [intro, simp]: "sp_wf p ⟹ sp_wf (sp_map e p)"
  by (simp add: sp_wf_def)
```

What a well-formed program answers, publishes and reads, asked of the program rather than of a tree. Each is the tree vocabulary applied to its compilation, so the three carry over unchanged — and the point of stating them is that a characterization of a combinator can now speak one vocabulary on both sides, instead of a program on the left and its contributions' trees on the right.

```
abbreviation traverse_program ::
  "('x,'g,'d::bot,'d) strategy_program ⇒ ('x + 'g ⇒ 'd) ⇒ 'd"
  where "traverse_program p σ ≡ traverse_rhs (sp_compile p) σ"

abbreviation sides_of_program ::
  "('x,'g,'d::bounded_semilattice_sup_bot,'d) strategy_program
   ⇒ ('x + 'g ⇒ 'd) ⇒ 'x + 'g ⇒ 'd"
  where "sides_of_program p σ ≡ sides_of_rhs (sp_compile p) σ"

abbreviation dep_program ::
  "('x + 'g ⇒ 'd::bot) ⇒ ('x,'g,'d,'d) strategy_program ⇒ ('x + 'g) set"
  where "dep_program σ p ≡ dep_aux σ (sp_compile p)"
```

Running a well-formed program under any continuation: its own effects happen, then the continuation runs on the value it answered.

Left bare. Their left-hand sides are `f (p k) σ` with both `p` and `k` schematic, which is not a higher-order pattern: as `simp` rules they match any application under one of the three observers and spawn an `sp_wf` subgoal for it — including their own right-hand sides, where `sides_of_program p σ` unfolds to `sides_of_rhs (sp_compile p) σ` and matches at `?p := sp_compile`. Nothing loops while `sp_wf sp_compile` is unprovable by the simpset, but that is an accident to rely on, so each use cites the rule.

```
lemma traverse_rhs_sp_wf:
  assumes "sp_wf p"
  shows "traverse_rhs (p k) σ = traverse_rhs (k (traverse_program p σ)) σ"
  by (subst sp_wfD[OF assms]) simp

lemma dep_aux_sp_wf:
  assumes "sp_wf p"
  shows "dep_aux σ (p k) = dep_program σ p ∪ dep_aux σ (k (traverse_program p σ))"
  by (subst sp_wfD[OF assms]) simp

lemma sides_of_rhs_sp_wf:
  fixes p :: "('x,'g,'d::bounded_semilattice_sup_bot,'d) strategy_program"
  assumes "sp_wf p"
  shows "sides_of_rhs (p k) σ
         = sides_of_program p σ ⊔ sides_of_rhs (k (traverse_program p σ)) σ"
  by (subst sp_wfD[OF assms]) simp
```

The three together: what a proof that has just exposed `p k` under an observer needs, cited as one name.

```
lemmas sp_wf_observes = traverse_rhs_sp_wf sides_of_rhs_sp_wf dep_aux_sp_wf
```

The `id`-specialized counterparts of the `sp_compile_with_*` equations above, stated directly against `sp_compile` so a proof reasoning about compiled programs does not have to unfold `sp_compile_def` at every step.

```
lemma sp_compile_bind_read_local [simp]:
  "sp_compile ((sp_read_local x) ≫= f) = QueryL x (λd. sp_compile (f d))"
  by (simp add: sp_compile_def)

lemma sp_compile_bind_read_global [simp]:
  "sp_compile ((sp_read_global g) ≫= f) = QueryG g (λd. sp_compile (f d))"
  by (simp add: sp_compile_def)

lemma sp_compile_bind_publish [simp]:
  "sp_compile ((sp_publish g d) ≫= f) = Side g d (sp_compile (f ()))"
  by (simp add: sp_compile_def)

lemma sp_compile_return [simp]: "sp_compile (sp_return a) = Answer a"
  by (simp add: sp_compile_def)

lemma sp_compile_read_local [simp]: "sp_compile (sp_read_local x) = QueryL x Answer"
  by (simp add: sp_compile_def)

lemma sp_compile_read_global [simp]: "sp_compile (sp_read_global g) = QueryG g Answer"
  by (simp add: sp_compile_def)

lemma sp_compile_bind_read_at [simp]:
  "sp_compile ((sp_read_at src) ≫= f) =
     (case src of Inl x ⇒ QueryL x (λd. sp_compile (f d))
                | Inr g ⇒ QueryG g (λd. sp_compile (f d)))"
  by (cases src) simp_all

lemma sp_compile_read_at [simp]:
  "sp_compile (sp_read_at src) =
     (case src of Inl x ⇒ QueryL x Answer | Inr g ⇒ QueryG g Answer)"
  by (cases src) simp_all
```

Left bare rather than `[simp]`: using this equation on arbitrary binds exposes `m` as a raw CPS function application. The specialized equations above preserve the program abstraction and give the simplifier the vendor-tree constructors it needs.

```
lemma sp_compile_bind: "sp_compile (m ≫= f) = m (λx. sp_compile (f x))"
  by (simp add: sp_compile_def sp_compile_with_def sp_bind_def)

lemma sp_compile_with_bind: "sp_compile_with encode (m ≫= f) = m (λx. sp_compile_with encode (f x))"
  by (simp add: sp_compile_with_def sp_bind_def)
```

Monad laws

The three laws, so a caller may rewrite a program before compiling it and know the tree is unchanged. All three hold by function extensionality alone – a program is a continuation transformer, and ≫= is composition – so no tree-level reasoning is involved.

```
lemma sp_bind_sp_return_left [simp]: "(sp_return a) ≫= f = f a"
  by (rule ext) (simp add: sp_bind_def sp_return_def)

lemma sp_bind_sp_return_right [simp]: "m ≫= sp_return = m"
  by (rule ext) (simp add: sp_bind_def sp_return_def)

lemma sp_bind_assoc:
  fixes m :: "('x, 'g, 'd, 'a) strategy_program"
    and f :: "'a ⇒ ('x, 'g, 'd, 'b) strategy_program"
    and g :: "'b ⇒ ('x, 'g, 'd, 'c) strategy_program"
  shows "(m ≫= f) ≫= g = m ≫= (λx. f x ≫= g)"
  by (rule ext) (simp add: sp_bind_def)

end
```

3.3 Strategy_Tree_Side_Buffering

```
theory Strategy_Tree_Side_Buffering
  imports "TD.Basics_side"
begin
```

Per-key buffering of one right-hand side's side effects

One right-hand side may emit several `Side` writes to the same key. Declaratively that is already a join — `sides_of_rhs`'s own `Side` equation joins into the key it writes — but a solver applies its update rule once per `Side`, on the accumulate reached so far. An update rule that keeps one slot per write origin therefore records a partial accumulate, and the global it feeds can fall below a value that same origin has already established; the drop destabilizes that global's readers, whose re-evaluation restores it, and the two alternate without converging.

`buffer_sides` removes the cause rather than the symptom: it accumulates every write per key and flushes each key exactly once, so an update rule only ever sees a completed contribution. Nothing here is specific to procedure calls, contexts, or the D/G layer — the property being repaired is that a right-hand side may name one key twice, which any equation generator can do.

The per-key accumulator

An association list rather than a map: the flush below emits one `Side` per entry in list order, and `acc_add` appends an unseen key at the end, so that order is first occurrence in the traversal. A map would leave the flush order to the key type's own arrangement, which is exactly the scheduling detail a narrowing update rule is sensitive to.

```
fun acc_add ::
  "'g ⇒ 'd::bounded_semilattice_sup_bot ⇒ ('g × 'd) list ⇒ ('g × 'd) list"
where
  "acc_add k d [] = [(k, d)]"
| "acc_add k d ((k', d') # kvs) =
     (if k' = k then (k', d' ⊔ d) # kvs else (k', d') # acc_add k d kvs)"

fun acc_at :: "'g ⇒ ('g × 'd::bounded_semilattice_sup_bot) list ⇒ 'd" where
  "acc_at k [] = bot"
| "acc_at k ((k', d') # kvs) = (if k' = k then d' else bot) ⊔ acc_at k kvs"

definition acc_val ::
  "('g × 'd::bounded_semilattice_sup_bot) list ⇒ 'x + 'g ⇒ 'd" where
  "acc_val acc z = (case z of Inl _ ⇒ bot | Inr k ⇒ acc_at k acc)"

lemma acc_val_Inl [simp]: "acc_val acc (Inl u) = bot"
  by (simp add: acc_val_def)

lemma acc_val_Inr [simp]: "acc_val acc (Inr k) = acc_at k acc"
  by (simp add: acc_val_def)

lemma acc_val_Nil [simp]: "acc_val [] z = bot"
  by (cases z) simp_all

lemma acc_at_add:
  "acc_at k' (acc_add k d acc) = acc_at k' acc ⊔ (if k' = k then d else bot)"
  by (induction acc) (auto simp: ac_simps)

lemma acc_val_add:
  "acc_val (acc_add k d acc) z = acc_val acc z ⊔ (if z = Inr k then d else bot)"
  by (cases z) (simp_all add: acc_at_add)

lemma acc_val_Cons:
  "acc_val ((k, d) # kvs) z = acc_val kvs z ⊔ (if z = Inr k then d else bot)"
  by (cases z) (auto simp: ac_simps)
```

Key order and distinctness What the flush order rests on: an unseen key is appended, a seen one is joined in place, so the key list only ever grows at the end and stays distinct.

```
lemma acc_keys_add:
  "map fst (acc_add k d acc)
     = (if k ∈ set (map fst acc) then map fst acc else map fst acc @ [k])"
  by (induction acc) (auto simp add: rev_image_eqI)

lemma acc_add_append:
  "k ∉ set (map fst acc) ⟹ acc_add k d acc = acc @ [(k, d)]"
  by (induction acc) (auto simp add: rev_image_eqI)
```

```
lemma distinct_acc_add:
  "distinct (map fst acc) ⟹ distinct (map fst (acc_add k d acc))"
  by (cases "k ∈ set (map fst acc)") (simp_all add: acc_keys_add)
```

Buffering a right-hand side

The traversal that does the buffering: writes are collected into the accumulator on the way down and emitted at the next flush point, so a query branch cannot lose the writes made before it.

A flush point is every answer, and every local read `QueryL y` with `publish_before y`. Flushing only at answers gives the strongest guarantee: no key is written twice on any path, whatever the tree. But it also holds back a write past the local reads that follow it, and a local read is what makes the solver evaluate the unknown read. A call site publishes the callee's entry state and then reads the callee's exit; held back to the answer, the publication reaches the solver only after the callee has been solved from an entry it has not yet received. Goblint's normal-call transfer issues `sidel` before `getl`, and its solver applies a side effect when it is issued, so there the callee starts from the published entry. A flush before such a read costs the at-most-once guarantee for keys written again after it, so `distinct_side_path_buffer_sides` below states when it still holds, and the caller chooses `publish_before` where it can tell that it does.

A key whose completed contribution is $\bot$ is not flushed. Publishing it would be a no-op for every consumer — `sides_of_rhs` joins it in, and a join with $\bot$ changes nothing — while still costing the solver an update-rule application and a destabilization of that global's readers. The generator appends the publication to every answer path without asking whether the transfer wrote anything, so on a node that touches no global this is the only place the emptiness is known.

```
primrec flush_sides ::
  "('g × 'd::bounded_semilattice_sup_bot) list ⇒ ('x, 'g, 'd) strategy_tree
   ⇒ ('x, 'g, 'd) strategy_tree"
where
  "flush_sides [] t = t"
| "flush_sides (kv # kvs) t =
     (if snd kv ≤ bot then flush_sides kvs t
      else Side (fst kv) (snd kv) (flush_sides kvs t))"

abbreviation flushed :: "('g × 'd::bounded_semilattice_sup_bot) list ⇒ ('g × 'd) list"
  where "flushed acc ≡ filter (λkv. ¬ snd kv ≤ bot) acc"

primrec buffer_aux ::
  "('x ⇒ bool) ⇒ ('g × 'd) list
   ⇒ ('x, 'g, 'd::bounded_semilattice_sup_bot) strategy_tree
   ⇒ ('x, 'g, 'd) strategy_tree"
where
  "buffer_aux publish_before acc (Answer d) = flush_sides acc (Answer d)"
| "buffer_aux publish_before acc (QueryL y g) =
     (if publish_before y
      then flush_sides acc (QueryL y (λv. buffer_aux publish_before [] (g v)))
      else QueryL y (λv. buffer_aux publish_before acc (g v)))"
| "buffer_aux publish_before acc (QueryG y g) =
     QueryG y (λv. buffer_aux publish_before acc (g v))"
| "buffer_aux publish_before acc (Side y d t) = buffer_aux publish_before (acc_add y d acc) t"

definition buffer_sides ::
  "('x ⇒ bool) ⇒ ('x, 'g, 'd::bounded_semilattice_sup_bot) strategy_tree
   ⇒ ('x, 'g, 'd) strategy_tree"
where
  "buffer_sides publish_before t = buffer_aux publish_before [] t"

definition buffer_eqs ::
  "('x ⇒ 'x ⇒ bool) ⇒ ('x, 'g, 'd::bounded_semilattice_sup_bot) eqsT
   ⇒ ('x, 'g, 'd) eqsT" where
  "buffer_eqs publish_before T = (λx. buffer_sides (publish_before x) (T x))"

lemma buffer_eqs_apply [simp]:
  "buffer_eqs publish_before T x = buffer_sides (publish_before x) (T x)"
  by (simp add: buffer_eqs_def)
```

The declarative reading is unchanged

Buffering is invisible to the declarative reading: the local value is untouched, and the contribution at each key is the same join as before, only emitted at a flush point. That is what makes this a scheduling change and not a semantic one, wherever the flush points are.

```
lemma traverse_rhs_flush_sides [simp]:
  "traverse_rhs (flush_sides acc t) σ = traverse_rhs t σ"
  by (induction acc) simp_all

lemma traverse_rhs_buffer_aux [simp]:
  "traverse_rhs (buffer_aux publish_before acc t) σ = traverse_rhs t σ"
  by (induction t arbitrary: acc) simp_all

lemma traverse_rhs_buffer_sides [simp]:
  "traverse_rhs (buffer_sides publish_before t) σ = traverse_rhs t σ"
  by (simp add: buffer_sides_def)

lemma sides_of_rhs_flush_sides [simp]:
  "sides_of_rhs (flush_sides acc t) σ z = sides_of_rhs t σ z ⊔ acc_val acc z"
  by (induction acc arbitrary: z)
     (auto simp add: Let_def acc_val_Cons ac_simps dest!: le_bot)

lemma sides_of_rhs_buffer_aux [simp]:
  "sides_of_rhs (buffer_aux publish_before acc t) σ z = sides_of_rhs t σ z ⊔ acc_val acc z"
  by (induction t arbitrary: acc z) (auto simp add: acc_val_add Let_def ac_simps)

lemma sides_of_rhs_buffer_sides [simp]:
  "sides_of_rhs (buffer_sides publish_before t) σ = sides_of_rhs t σ"
  by (rule ext) (simp add: buffer_sides_def)

lemma dep_aux_flush_sides [simp]:
  "dep_aux σ (flush_sides acc t) = dep_aux σ t"
  by (induction acc) simp_all

lemma dep_aux_buffer_aux [simp]:
  "dep_aux σ (buffer_aux publish_before acc t) = dep_aux σ t"
  by (induction t arbitrary: acc) simp_all

lemma dep_aux_buffer_sides [simp]:
  "dep_aux σ (buffer_sides publish_before t) = dep_aux σ t"
  by (simp add: buffer_sides_def)
```

The operational property the update rules rely on

Semantic preservation alone does not say a solver sees fewer writes; it says the writes sum to the same thing. `side_path` names the keys one evaluation actually writes, in the order it writes them — the continuations are functions, so which `Side` nodes an evaluation meets depends on σ, and the statement has to be relative to it.

```
primrec side_path ::
  "('x + 'g ⇒ 'd) ⇒ ('x, 'g, 'd) strategy_tree ⇒ 'g list" where
  "side_path σ (Answer d) = []"
| "side_path σ (QueryL y g) = side_path σ (g (σ (Inl y)))"
| "side_path σ (QueryG y g) = side_path σ (g (σ (Inr y)))"
| "side_path σ (Side y d t) = y # side_path σ t"

lemma side_path_flush_sides [simp]:
  "side_path σ (flush_sides acc t) = map fst (flushed acc) @ side_path σ t"
  by (induction acc) simp_all
```

The flush points split one evaluation of the original tree into segments, and the buffer writes a key once for each segment in which the key receives a value other than $\bot$. The buffered evaluation therefore writes each key at most once exactly when no key receives such a value in two segments. `settled_writes C P` walks the evaluation and checks this: `C` holds the keys an earlier flush point has already written, and `P` the keys the current segment will write at the next one.

```
primrec settled_writes ::
  "('x ⇒ bool) ⇒ ('x + 'g ⇒ 'd) ⇒ 'g set ⇒ 'g set
   ⇒ ('x, 'g, 'd::bounded_semilattice_sup_bot) strategy_tree ⇒ bool"
where
  "settled_writes publish_before σ C P (Answer d) = True"
| "settled_writes publish_before σ C P (QueryL y g) =
     (if publish_before y
      then settled_writes publish_before σ (C ∪ P) {} (g (σ (Inl y)))
      else settled_writes publish_before σ C P (g (σ (Inl y))))"
| "settled_writes publish_before σ C P (QueryG y g) =
     settled_writes publish_before σ C P (g (σ (Inr y)))"
| "settled_writes publish_before σ C P (Side y d t) =
     ((d ≤ bot ∨ y ∉ C)
      ∧ settled_writes publish_before σ C (if d ≤ bot then P else insert y P) t)"
```

```
lemma flushed_keys_acc_add:
  "fst ‘ set (flushed (acc_add k d acc))
     = fst ‘ set (flushed acc) ∪ (if d ≤ bot then {} else {k})"
proof (induction acc)
  case Nil
  then show ?case by simp
next
  case (Cons kv acc)
  obtain k' d' where kv: "kv = (k', d')" by (cases kv)
  show ?case
  proof (cases "k' = k")
    case True
    then show ?thesis
      by (cases "d' ≤ bot"; cases "d ≤ bot") (auto simp add: kv)
  next
    case False
    then show ?thesis
      using Cons.IH by (cases "d' ≤ bot") (auto simp add: kv)
  qed
qed

lemma distinct_side_path_buffer_aux:
  assumes "distinct (map fst acc)"
    and "fst ‘ set (flushed acc) ∩ C = {}"
    and "settled_writes publish_before σ C (fst ‘ set (flushed acc)) t"
  shows "distinct (side_path σ (buffer_aux publish_before acc t))
         ∧ set (side_path σ (buffer_aux publish_before acc t)) ∩ C = {}"
  using assms
proof (induction t arbitrary: acc C)
  case (Answer d)
  then show ?case by (auto simp add: distinct_map_filter)
next
  case (QueryL y g)
  show ?case
  proof (cases "publish_before y")
    case True
    have "distinct (side_path σ (buffer_aux publish_before [] (g (σ (Inl y)))))
        ∧ set (side_path σ (buffer_aux publish_before [] (g (σ (Inl y)))))
            ∩ (C ∪ fst ‘ set (flushed acc)) = {}"
      by (rule QueryL.IH) (use QueryL.prems(3) True in simp_all)
    then show ?thesis
      using QueryL.prems(1,2) True by (auto simp add: distinct_map_filter)
  next
    case False
    then show ?thesis using QueryL.prems by (simp add: QueryL.IH)
  qed
next
  case (QueryG y g)
  then show ?case by simp
next
  case (Side y d t)
  have "distinct (map fst (acc_add y d acc))"
    using Side.prems(1) by (rule distinct_acc_add)
  moreover have "fst ‘ set (flushed (acc_add y d acc)) ∩ C = {}"
    unfolding flushed_keys_acc_add using Side.prems(2,3) by auto
  moreover have "settled_writes publish_before σ C (fst ‘ set (flushed (acc_add y d acc))) t"
    unfolding flushed_keys_acc_add using Side.prems(3) by (cases "d ≤ bot") simp_all
  ultimately have "distinct (side_path σ (buffer_aux publish_before (acc_add y d acc) t))
      ∧ set (side_path σ (buffer_aux publish_before (acc_add y d acc) t)) ∩ C = {}"
    by (rule Side.IH)
  then show ?case by simp
qed

theorem distinct_side_path_buffer_sides:
  assumes "settled_writes publish_before σ {} {} t"
  shows "distinct (side_path σ (buffer_sides publish_before t))"
  using distinct_side_path_buffer_aux[where acc = "[]" and C = "{}"] assms
  by (simp add: buffer_sides_def)
```

With no flush point before a local read nothing is ever closed, so the condition holds for every tree and every evaluation: flushing at answers only writes each key at most once unconditionally.

```
lemma settled_writes_answers_only:
  "settled_writes (λ_. False) σ {} P t"
  by (induction t arbitrary: P) simp_all

corollary distinct_side_path_buffer_sides_answers_only:
```

```
  "distinct (side_path σ (buffer_sides (λ_. False) t))"
  by (rule distinct_side_path_buffer_sides) (rule settled_writes_answers_only)
```

The elision, pinned at both polarities: a completed contribution of $\bot$ costs the evaluation no write, and one carrying a value still costs exactly one. Stated over `side_path` because the writes an evaluation performs are what the elision is for.

```
lemma side_path_flush_sides_bot [simp]:
  "side_path σ (flush_sides [(k, bot)] t) = side_path σ t"
  by simp

lemma side_path_flush_sides_value:
  assumes "¬ d ≤ bot"
  shows "side_path σ (flush_sides [(k, d)] t) = k # side_path σ t"
  using assms by simp
```

So a buffered system writes each key at most once per evaluation whenever its writes settle: every key an evaluation contributes to is flushed exactly once, with its completed contribution, and a key the evaluation never touches is not written at all. Flushing at answers only replaces the generator-level assumption that no two contributions of one right-hand side can name the same key — an assumption about how equations happen to be built, which a shared resume node refutes — with a property of the equations handed to the solver. A caller that also flushes before local reads takes that assumption back for the keys written on both sides of such a read, and must know there are none.

Buffering is idempotent

The entries a flush drops are exactly the ones it would have skipped, so flushing the filtered accumulator is the same tree.

```
lemma flush_sides_flushed [simp]: "flush_sides (flushed acc) t = flush_sides acc t"
  by (induction acc) simp_all

lemma acc_add_fold_append:
  "distinct (map fst (acc0 @ acc))
     ⟹ foldl (λa kv. acc_add (fst kv) (snd kv) a) acc0 acc = acc0 @ acc"
  by (induction acc arbitrary: acc0) (auto simp add: acc_add_append)

lemma buffer_aux_flush_sides:
  "buffer_aux publish_before acc0 (flush_sides acc t)
     = buffer_aux publish_before
         (foldl (λa kv. acc_add (fst kv) (snd kv) a) acc0 (flushed acc)) t"
  by (induction acc arbitrary: acc0) simp_all

lemma buffer_sides_idem [simp]:
  "buffer_sides publish_before (buffer_sides publish_before t) = buffer_sides publish_before t"
proof -
  have "⋀acc. distinct (map fst acc)
      ⟹ buffer_aux publish_before [] (buffer_aux publish_before acc t)
          = buffer_aux publish_before acc t"
    by (induction t)
       (auto simp add: buffer_aux_flush_sides acc_add_fold_append distinct_acc_add
             distinct_map_filter)
  then show ?thesis by (simp add: buffer_sides_def)
qed
```

The one transfer every consumer needs

`part_post_solution` — the solver interface the analyzer soundness spine consumes — is stated over $\texttt{dep}_L$, `traverse_rhs` and `sides_of_rhs` alone. All three are invariant under buffering, wherever the flush points are, so a solution of the buffered system is a solution of the original one. Buffering preserves this declarative right-hand-side semantics and the post-solution obligation the soundness proof consumes; it does not claim the buffered and unbuffered systems drive the solver through the same operational iteration, or, under widening, to the same computed post-solution.

```
lemma dep_buffer_eqs [simp]: "dep (buffer_eqs publish_before T) σ x = dep T σ x"
  by (simp add: dep_def)

lemma dep_L_buffer_eqs [simp]: "dep_L (buffer_eqs publish_before T) σ x = dep_L T σ x"
  by (simp add: dep_L_def)

theorem part_post_solution_buffer_eqs [simp]:
  "part_post_solution (buffer_eqs publish_before T) x σ vars
```

```
    = part_post_solution T x σ vars"
  by simp

end
```

3.4 Strategy_Tree_Pair

```
theory Strategy_Tree_Pair
  imports Strategy_Tree_Side_Buffering
begin
```

A right-hand side as a pair

A side-effecting right-hand side is often presented as a function from a valuation to a pair: the contributions it publishes, a partial map from unknowns to values, and its own value. A strategy tree evaluates to the same pair. The second component is `traverse_rhs`; the first is `sides_of_rhs` restricted to the keys `side_path` visits, so a key that receives only $\bot$ stays in the domain and a key that receives nothing does not. Both components come from the existing evaluators.

```
definition rhs_sides ::
  "('x,'g,'d::bounded_semilattice_sup_bot) strategy_tree ⇒ ('x + 'g ⇒ 'd) ⇒ 'x + 'g ⇀ 'd"
where
  "rhs_sides t σ = (Some ∘ sides_of_rhs t σ) |` (Inr ` set (side_path σ t))"

definition rhs_pair ::
  "('x,'g,'d::bounded_semilattice_sup_bot) strategy_tree ⇒ ('x + 'g ⇒ 'd)
     ⇒ ('x + 'g ⇀ 'd) × 'd"
where
  "rhs_pair t σ = (rhs_sides t σ, traverse_rhs t σ)"
```

The partial map loses nothing

Off the visited keys `sides_of_rhs` is $\bot$, so reading the map back with `mlup` recovers it.

```
lemma sides_of_rhs_Inl [simp]: "sides_of_rhs t σ (Inl u) = ⊥"
  by (induction t) (simp_all add: Let_def)

lemma sides_of_rhs_off_path:
  "g ∉ set (side_path σ t) ⟹ sides_of_rhs t σ (Inr g) = ⊥"
  by (induction t) (auto simp: Let_def)

lemma rhs_sides_apply:
  "rhs_sides t σ z =
     (if z ∈ Inr ` set (side_path σ t) then Some (sides_of_rhs t σ z) else None)"
  by (simp add: rhs_sides_def restrict_map_def)

lemma dom_rhs_sides: "dom (rhs_sides t σ) = Inr ` set (side_path σ t)"
  by (auto simp: dom_def rhs_sides_apply)
```

Side effects only reach global unknowns; a local unknown is never in the domain.

```
lemma rhs_sides_Inl [simp]: "rhs_sides t σ (Inl u) = None"
  by (auto simp: rhs_sides_apply)

lemma mlup_rhs_sides: "mlup (rhs_sides t σ) = sides_of_rhs t σ"
proof
  fix z
  show "mlup (rhs_sides t σ) z = sides_of_rhs t σ z"
    by (cases z) (auto simp: mlup_def rhs_sides_apply sides_of_rhs_off_path image_iff)
qed
```

Several side effects to one unknown appear once in the domain, with the join of their values, in the order of the traversal.

```
lemma rhs_sides_Side:
  "rhs_sides (Side g d t) σ = (rhs_sides t σ)(Inr g ↦ mlup (rhs_sides t σ) (Inr g) ⊔ d)"
  by (rule ext) (auto simp: mlup_rhs_sides rhs_sides_apply Let_def)

lemma rhs_pair_two_sides:
  "rhs_pair (Side g a (Side g b (Answer d))) σ = ([Inr g ↦ b ⊔ a], d)"
  unfolding rhs_pair_def by (auto simp: rhs_sides_apply Let_def intro!: ext)
```

The certificate in pair form

A side map is below a valuation exactly when every contribution it publishes is below its target.

```
lemma sides_of_rhs_le_iff_rhs_sides:
  "sides_of_rhs t σ ≤ σ ⟷ (∀z d. rhs_sides t σ z = Some d ⟶ d ≤ σ z)"
proof
  assume "sides_of_rhs t σ ≤ σ"
```

```
  then show "∀z d. rhs_sides t σ z = Some d ⟶ d ≤ σ z"
    by (auto simp: rhs_sides_apply le_fun_def)
next
  assume bound: "∀z d. rhs_sides t σ z = Some d ⟶ d ≤ σ z"
  show "sides_of_rhs t σ ≤ σ"
  proof (rule le_funI)
    fix z
    show "sides_of_rhs t σ z ≤ σ z"
    proof (cases "z ∈ Inr ‘ set (side_path σ t)")
      case True
      then show ?thesis
        using bound by (simp add: rhs_sides_apply)
    next
      case False
      then have "sides_of_rhs t σ z = ⊥"
        by (cases z) (auto simp: sides_of_rhs_off_path image_iff)
      then show ?thesis
        by simp
    qed
  qed
qed
```

`part_post_solution` with its local and side bounds stated on the pair: the value is below the unknown, and every published contribution is below its target. The bounds constrain the side-effect targets although they are global unknowns outside `vars`, and say nothing about unknowns outside `vars` that the solve never reached.

```
theorem part_post_solution_iff_rhs_pair:
  "part_post_solution T x σ vars ⟷
     x ∈ vars ∧ (∀u ∈ vars. dep_L T σ u ⊆ vars ∧
       (case rhs_pair (T u) σ of (ϱ, d) ⇒
          d ≤ σ (Inl u) ∧ (∀z d'. ϱ z = Some d' ⟶ d' ≤ σ z)))"
  unfolding rhs_pair_def by (simp add: sides_of_rhs_le_iff_rhs_sides)

end
```

3.5 TD_Solver_Bridge

```
theory TD_Solver_Bridge
  imports "TD.TD_side_Interface"
begin
```

The semantic boundary between Voblint and the vendored TD solver

Voblint relies on the vendored TD solver only through the locale below, which packages the solver facts the analysis pipeline uses.

What a pipeline asks of its solver

Four facts are everything the analysis pipeline uses about a solver: a solve in the solver's domain answers a partial post-solution over finitely many keys, and an answer of the executable solver `solve_c` places the query in that domain and is the solve itself. TD proves all four inside `TD_side_upd_rule` (`partial_post_solution`, `finite_stabl_solve`, `solve_dom_of_solve_c`, `solve_code_equation`). `certified_solver` names that contract once, over any equation system, so a pipeline assumes one locale instead of four facts and a solver discharges it once for all of its instances.

```
locale certified_solver =
  fixes solve :: "('x, 'g, 'd::bounded_semilattice_sup_bot) eqsT
                  ⇒ 'x ⇒ 'x set × ('x + 'g ⇒ 'd)"
    and solve_dom :: "('x, 'g, 'd) eqsT ⇒ 'x ⇒ bool"
    and solve_c :: "('x, 'g, 'd) eqsT ⇒ 'x ⇒ ('x set × ('x + 'g ⇒ 'd)) option"
  assumes solve_pp:
      "⋀eqs x. solve_dom eqs x
         ⟹ part_post_solution eqs x (snd (solve eqs x)) (fst (solve eqs x))"
    and solve_fin: "⋀eqs x. solve_dom eqs x ⟹ finite (fst (solve eqs x))"
    and dom_of_solve_c: "⋀eqs x. solve_c eqs x ≠ None ⟹ solve_dom eqs x"
    and solve_of_solve_c: "⋀eqs x sol. solve_c eqs x = Some sol ⟹ solve eqs x = sol"

end
```

3.6 Globals_Rule

```
theory Globals_Rule
  imports TD_Solver_Bridge
begin
```

Which rule updates a side-effected global

The vendored solver warrows every local unknown at a widening point, whichever instance runs. What its instances differ in is how a contribution side-effected into a global unknown is merged: joined into the value, joined per origin, warrowed into the value, warrowed per origin, or widened per origin with narrowing cut short once an origin has switched from widening to narrowing a given number of times. That merge is the one choice a caller makes, so it is a value here rather than a choice between five solvers: a single interpretation takes the rule as a parameter, and every solver fact holds for all five at once.

```
datatype globals_rule =
    Globals_Join
  | Globals_Per_Origin
  | Globals_Warrow
  | Globals_Warrow_Per_Origin
  | Globals_Bounded_Narrowing nat
```

One update state for every rule

The bounded-narrowing rule keeps a counter per global and origin beside the recorded contributions; the other four rules keep only the contributions. So the single interpretation runs on the state with the counter, and each of the four runs on its contribution part and leaves the counter unchanged. Every `update_rule` obligation speaks about the contributions alone, so the lifted rule inherits them.

The lifted rule also reports no change when the merged value equals the old one. The two warrowing rules answer with a new value whenever an origin's contribution changes, even when the merge over all origins stays the same, and the solver destabilizes every reader of a global it is handed a value for. Recording the contribution and reporting nothing keeps those readers stable.

```
definition changed :: "'d ⇒ 'd option ⇒ 'd option" where
  "changed d res = (case res of Some d'' ⇒ if d'' = d then None else Some d'' | None ⇒ None)"

lemma changed_SomeD: "changed d res = Some d'' ⟹ res = Some d''"
  by (cases res) (auto simp: changed_def split: if_splits)

lemma changed_NoneD: "changed d res = None ⟹ res = None ∨ res = Some d"
  by (cases res) (auto simp: changed_def split: if_splits)

definition lift_basic_rule ::
    "('d ⇒ 'x ⇒ 'g ⇒ 'd ⇒ ('x, 'g, 'd) ug_state ⇒ 'd option × ('x, 'g, 'd) ug_state)
     ⇒ 'd ⇒ 'x ⇒ 'g ⇒ 'd ⇒ ('x, 'g, 'd) ug_state_with_gas
     ⇒ 'd option × ('x, 'g, 'd) ug_state_with_gas" where
  "lift_basic_rule f d orig g d' state =
     (let (res, st) = f d orig g d' (ug_state.truncate state)
      in (changed d res, ug_state.ϱ_update (λ_. ug_state.ϱ st) state))"

lemma lift_basic_ruleE:
  assumes "(res, state') = lift_basic_rule f d orig g d' state"
  obtains res0 st where "(res0, st) = f d orig g d' (ug_state.truncate state)"
    and "res = changed d res0" and "ug_state.ϱ state' = ug_state.ϱ st"
  using assms by (auto simp: lift_basic_rule_def split: prod.splits)

lemma update_rule_lift_basic_rule:
  assumes f: "update_rule init_basic_ug_state f"
  shows "update_rule init_ug_state_with_gas (lift_basic_rule f)"
proof
  show "∀g orig. rho_lookup (ug_state.ϱ init_ug_state_with_gas) g orig ≤ ⊥"
    by (simp add: init_ug_state_with_gas_def fmlookup_default_def)
next
  fix res state' d orig g d' state g' orig'
  assume step: "(res, state') = lift_basic_rule f d orig g d' state" and ne: "g' ≠ g"
  from step obtain res0 st where e: "(res0, st) = f d orig g d' (ug_state.truncate state)"
    and r: "ug_state.ϱ state' = ug_state.ϱ st" by (rule lift_basic_ruleE)
  show "rho_lookup (ug_state.ϱ state') g' orig' = rho_lookup (ug_state.ϱ state) g' orig'"
    using update_rule.update_global_untouched(1)[OF f e ne] by (simp add: r)
next
  fix res state' d orig g d' state g' orig'
  assume step: "(res, state') = lift_basic_rule f d orig g d' state" and ne: "orig' ≠ orig"
  from step obtain res0 st where e: "(res0, st) = f d orig g d' (ug_state.truncate state)"
```

```
    and r: "ug_state.ϱ state' = ug_state.ϱ st" by (rule lift_basic_ruleE)
  show "rho_lookup (ug_state.ϱ state') g' orig' = rho_lookup (ug_state.ϱ state) g' orig'"
    using update_rule.update_global_untouched(2)[OF f e ne] by (simp add: r)
next
  fix res state' d orig g d' state
  assume step: "(res, state') = lift_basic_rule f d orig g d' state"
  from step obtain res0 st where e: "(res0, st) = f d orig g d' (ug_state.truncate state)"
    and r: "ug_state.ϱ state' = ug_state.ϱ st" by (rule lift_basic_ruleE)
  show "d' ≤ rho_lookup (ug_state.ϱ state') g orig"
    using update_rule.update_global_recorded_in_rho[OF f e] by (simp add: r)
next
  fix state' d orig g d' state
  assume step: "(None, state') = lift_basic_rule f d orig g d' state"
    and inv: "∀orig. rho_lookup (ug_state.ϱ state) g orig ≤ d"
  from step obtain res0 st where e: "(res0, st) = f d orig g d' (ug_state.truncate state)"
    and c: "None = changed d res0" and r: "ug_state.ϱ state' = ug_state.ϱ st"
    by (rule lift_basic_ruleE)
  from changed_NoneD[OF c[symmetric]] show "rho_lookup (ug_state.ϱ state') g orig ≤ d"
  proof
    assume "res0 = None"
    then show ?thesis
      using update_rule.update_global_preserves_rho_invariant(1)[OF f] e inv by (simp add: r)
  next
    assume "res0 = Some d"
    then show ?thesis
      using update_rule.update_global_preserves_rho_invariant(2)[OF f] e inv by (simp add: r)
  qed
next
  fix d'' state' d orig g d' state orig'
  assume step: "(Some d'', state') = lift_basic_rule f d orig g d' state"
    and inv: "∀orig. rho_lookup (ug_state.ϱ state) g orig ≤ d"
  from step obtain res0 st where e: "(res0, st) = f d orig g d' (ug_state.truncate state)"
    and c: "Some d'' = changed d res0" and r: "ug_state.ϱ state' = ug_state.ϱ st"
    by (rule lift_basic_ruleE)
  have "res0 = Some d''" by (rule changed_SomeD) (use c in simp)
  then show "rho_lookup (ug_state.ϱ state') g orig' ≤ d''"
    using update_rule.update_global_preserves_rho_invariant(2)[OF f] e inv by (simp add: r)
qed
```

The rule as a value

`update_global_of` maps each `globals_rule` to its update rule, so one interpretation of `TD_side_upd_rule` covers every rule a configuration can select.

```
definition update_global_of ::
    "globals_rule ⇒ 'd ⇒ 'x ⇒ 'g ⇒ 'd::{bounded_semilattice_sup_bot,warrowing}
       ⇒ ('x, 'g, 'd) ug_state_with_gas ⇒ 'd option × ('x, 'g, 'd) ug_state_with_gas" where
  "update_global_of r =
     (case r of
        Globals_Join ⇒ lift_basic_rule update_global_always_join
      | Globals_Per_Origin ⇒ lift_basic_rule update_global_per_origin
      | Globals_Warrow ⇒ lift_basic_rule update_global_warrowing_apinis
      | Globals_Warrow_Per_Origin ⇒ lift_basic_rule update_global_warrowing_per_origin
      | Globals_Bounded_Narrowing n ⇒ update_global_bounded_narrowing n)"

lemma update_rule_update_global_of:
  "update_rule init_ug_state_with_gas (update_global_of r)"
  by (cases r)
     (simp_all add: update_global_of_def update_rule_lift_basic_rule
        always_join.update_rule_axioms per_origin.update_rule_axioms
        warrowing_apinis.update_rule_axioms warrowing_per_origin.update_rule_axioms
        bounded_narrowing.update_rule_axioms)

global_interpretation TD_side_rule_Interp:
  TD_side_upd_rule init_ug_state_with_gas "update_global_of r" T for r T
  defines TD_side_rule_Interp_solve = TD_side_rule_Interp.solve
    and TD_side_rule_Interp_solve_c = TD_side_rule_Interp.solve_c
    and TD_side_rule_Interp_solve_rec_c = TD_side_rule_Interp.solve_rec_c
  by (simp add: TD_side_upd_rule.intro update_rule_update_global_of)
```

The solver contract of the pipeline, once for every rule.

```
lemma td_certified_solver:
  "certified_solver (TD_side_rule_Interp_solve r)
     (TD_side_rule_Interp.solve_dom TYPE('g) TYPE('d::{bounded_semilattice_sup_bot,warrowing}) r)
     (TD_side_rule_Interp_solve_c r)"
  by unfold_locales
```

```
     (erule TD_side_rule_Interp.partial_post_solution[OF _ surjective_pairing],
      erule TD_side_rule_Interp.finite_stabl_solve,
      erule TD_side_rule_Interp.solve_dom_of_solve_c,
      simp add: TD_side_rule_Interp.solve_code_equation)

end
```

3.7 Solver_Trace

```
theory Solver_Trace
  imports Globals_Rule
begin
```

A solver trace inside the exported code

The vendored interface hides the solver's state fields (`TD_side_Interface`). The traced equations below restate the solver's own terms, so this theory names those fields again, as input abbreviations of the hidden constants.

```
abbreviation (input) point where "point ≡ TD_side_upd_rule.state.point"
abbreviation (input) point_update where "point_update ≡ TD_side_upd_rule.state.point_update"
abbreviation (input) σ where "σ ≡ TD_side.state.σ"
abbreviation (input) σ_update where "σ_update ≡ TD_side.state.σ_update"
```

The executable solver can report its steps. `trace_event` is the identity on the logic's side: it takes a channel name and a suspended event and returns `()`, so every equation below is proved equal to the untraced one by unfolding it. Code export maps the constant to a target-language hook (`Trace_Run`); the hook forces the suspension only when tracing is on, so an untraced run never builds an event. The Eval target keeps the equation, so proofs by evaluation run the same code without any hook.

The traced solver is a copy of the vendored solver's executable functions under new constants, each defined as the function it copies and given the traced equation as its code equation. The vendored functions keep their own code equations, so a caller that names them runs no hook; only a caller that names the traced constants reports events.

```
definition trace_event :: "String.literal ⇒ (unit ⇒ 'e) ⇒ unit" where
  "trace_event channel e = ()"
```

One event per solver step that Goblint's tracing of its top-down solvers reports (https://github.com/goblint/analyzer/blob/master/src/solver/td_simplified.ml, https://github.com/goblint/analyzer/blob/master/src/solver/td3.ml), plus two with no counterpart there: `Ev_Rhs`, the value a right-hand side returns, and the start and end of one solve, and `Ev_Wpoint_Clear`, an iteration of an already stable unknown dropping it from the widening points. A local unknown is `'x`, a global one `'g`, a value `'d`. Booleans are the solver's own sets read at the step: `called`, `stabl`, and `point`, the widening points, and whether an old value is ⊥.

```
datatype ('x, 'g, 'd) solver_event =
    Ev_Start 'x
  | Ev_Stop
  | Ev_Query 'x 'x bool bool
  | Ev_Query_Wpoint 'x bool
  | Ev_Iterate_From_Query 'x
  | Ev_Add_Infl "'x + 'g" 'x
  | Ev_Answer 'x 'x 'd
  | Ev_Query_Global 'x 'g
  | Ev_Answer_Global 'x 'g 'd
  | Ev_Iterate 'x bool bool bool
  | Ev_Eq 'x
  | Ev_Rhs 'x 'd
  | Ev_Still_Unstable 'x
  | Ev_Widen 'x bool
  | Ev_Sol 'x bool 'd 'd 'd
  | Ev_Wpoint_Remove 'x bool
  | Ev_Wpoint_Clear 'x bool
  | Ev_Update 'x bool bool 'd 'd
  | Ev_Iterate_Changed 'x
  | Ev_Side 'x 'g 'd
  | Ev_Update_Global 'x 'g bool 'd 'd
  | Ev_Destabilize "'x + 'g"
  | Ev_Stable_Remove 'x
```

Destabilization. `destab_opt x` clears the unknowns that read `x` from the stable set and continues through each one not currently being solved.

```
lemma destab_iter_opt_traced:
  fixes i :: "('x + 'g, 'x list) fmap"
  shows
    "destab_iter_opt [] i s c = (i, s)"
    "destab_iter_opt (y # ys) i s c =
      (let _ = trace_event STR ''solver''
                 (λ_. (Ev_Stable_Remove y :: ('x, 'g, unit) solver_event)) in
       let (i, s) = (if y ∈ c then (i, s - {y}) else destab_opt (Inl y) i (s - {y}) c) in
       destab_iter_opt ys i s c)"
```

```
  by (simp_all only: destab_iter_opt.simps trace_event_def Let_def)

lemma destab_opt_traced:
  fixes x :: "'x + 'g"
  shows
    "destab_opt x i s c =
      (let _ = trace_event STR ''solver''
                (λ_. (Ev_Destabilize x :: ('x, 'g, unit) solver_event)) in
       destab_iter_opt (fmlookup_default i [] x) (fmdrop x i) s c)"
  by (simp only: destab_opt.simps trace_event_def Let_def)

definition traced_destab_iter_opt ::
    "'x list ⇒ ('x + 'g, 'x list) fmap ⇒ 'x set ⇒ 'x set ⇒ ('x + 'g, 'x list) fmap × 'x set"
  where [code del]: "traced_destab_iter_opt ys i s c = destab_iter_opt ys i s c"

definition traced_destab_opt ::
    "'x + 'g ⇒ ('x + 'g, 'x list) fmap ⇒ 'x set ⇒ 'x set ⇒ ('x + 'g, 'x list) fmap × 'x set"
  where [code del]: "traced_destab_opt x i s c = destab_opt x i s c"

lemmas traced_destab_code [code] =
  destab_iter_opt_traced[folded traced_destab_iter_opt_def traced_destab_opt_def]
  destab_opt_traced[folded traced_destab_iter_opt_def traced_destab_opt_def]
```

The solver. The equation below is the vendored `solve_rec_c` equation with events added at the steps Goblint traces: `Q` is a query, `I` an iteration, `R` one evaluation of the right-hand side, `E` one instruction of it.

```
unbundle lattice_syntax

lemma solve_rec_c_traced:
  fixes T :: "('x, 'g, 'd::{bounded_semilattice_sup_bot,warrowing}) eqsT"
  shows "TD_side_rule_Interp_solve_rec_c r T s = (case s of
    TD_side_upd_rule.func_state.Q (y, x, state, ug_state) ⇒
      (let _ = trace_event STR ''solver'' (λ_. (Ev_Query y x (x ∈ stabl state) (x ∈ called state)
                :: ('x, 'g, 'd) solver_event)) in
       Option.bind (
        if x ∈ called state then
          (let _ = trace_event STR ''solver''
                    (λ_. (Ev_Query_Wpoint x (x ∈ point state) :: ('x, 'g, 'd) solver_event)) in
           Some ((σ state) (Inl x), state⦇ point := insert x (point state) ⦈, ug_state))
        else
          (let _ = trace_event STR ''solver''
                    (λ_. (Ev_Iterate_From_Query x :: ('x, 'g, 'd) solver_event)) in
           TD_side_rule_Interp_solve_rec_c r T
             (TD_side_upd_rule.func_state.I (x, state⦇ called := insert x (called state) ⦈, ug_state))))
        (λ(xd, state, ug_state).
          let _ = trace_event STR ''solver'' (λ_. (Ev_Add_Infl (Inl x) y :: ('x, 'g, 'd) solver_event)) in
          let _ = trace_event STR ''solver'' (λ_. (Ev_Answer y x xd :: ('x, 'g, 'd) solver_event)) in
          Some (xd, state⦇infl := fminsert (infl state) (Inl x) y⦈, ug_state)))
  | TD_side_upd_rule.func_state.I (x, state, ug_state) ⇒
      (let _ = trace_event STR ''solver''
                (λ_. (Ev_Iterate x (x ∈ called state) (x ∈ stabl state) (x ∈ point state)
                  :: ('x, 'g, 'd) solver_event)) in
       if x ∉ stabl state then Option.bind (
        TD_side_rule_Interp_solve_rec_c r T (TD_side_upd_rule.func_state.R (x, state, ug_state)))
        (λ(d_new, state1, ug_state1).
          let _ = trace_event STR ''solver''
                    (λ_. (Ev_Widen x (x ∈ point state) :: ('x, 'g, 'd) solver_event)) in
          let d_new' = (if x ∈ point state then (σ state1) (Inl x) ∇Δ d_new else d_new) in
          let _ = trace_event STR ''solver''
                    (λ_. (Ev_Sol x (x ∈ point state) ((σ state1) (Inl x)) d_new d_new'
                      :: ('x, 'g, 'd) solver_event)) in
          if (σ state1) (Inl x) = d_new' then
            (let _ = trace_event STR ''solver''
                      (λ_. (Ev_Wpoint_Remove x (x ∈ point state1) :: ('x, 'g, 'd) solver_event)) in
             Some (d_new', state1⦇ called := called state1 - {x}, point := (point state1) - {x}⦈,
               ug_state1))
          else
            (let _ = trace_event STR ''solver''
                      (λ_. (Ev_Update x (x ∈ point state1) ((σ state1) (Inl x) = ⊥)
                              ((σ state1) (Inl x)) d_new' :: ('x, 'g, 'd) solver_event)) in
             let (infl1, stabl1) = destab_opt (Inl x) (infl state1) (stabl state1) (called state1) in
             let _ = trace_event STR ''solver''
                      (λ_. (Ev_Iterate_Changed x :: ('x, 'g, 'd) solver_event)) in
             TD_side_rule_Interp_solve_rec_c r T (TD_side_upd_rule.func_state.I (x, state1⦇ infl := infl1, stabl :=
stabl1,
               σ := (σ state1)(Inl x := d_new') ⦈, ug_state1))))
      else
```

```
       (let _ = trace_event STR ''solver''
                  (λ_. (Ev_Wpoint_Clear x (x ∈ point state) :: ('x, 'g, 'd) solver_event)) in
        Some ((σ state) (Inl x), state(| called := called state - {x}, point := (point state) - {x}|),
         ug_state)))
  | TD_side_upd_rule.func_state.R (x, state, ug_state) ⇒
     (let _ = trace_event STR ''solver'' (λ_. (Ev_Eq x :: ('x, 'g, 'd) solver_event)) in
      Option.bind (
       TD_side_rule_Interp_solve_rec_c r T
         (TD_side_upd_rule.func_state.E (x, T x, (λ_. ⊥)), state (|stabl := insert x (stabl state)|), ug_state)))
       (λ(xd, state, ug_state). if x ∈ stabl state then Some (xd, state, ug_state)
         else (let _ = trace_event STR ''solver''
                       (λ_. (Ev_Still_Unstable x :: ('x, 'g, 'd) solver_event)) in
               TD_side_rule_Interp_solve_rec_c r T (TD_side_upd_rule.func_state.R (x, state, ug_state)))))
  | TD_side_upd_rule.func_state.E (x, t, sides, state, ug_state) ⇒ (case t of
       Answer d ⇒
         (let _ = trace_event STR ''solver'' (λ_. (Ev_Rhs x d :: ('x, 'g, 'd) solver_event)) in
          Some (d, state, ug_state))
     | QueryL y g ⇒ Option.bind (TD_side_rule_Interp_solve_rec_c r T (TD_side_upd_rule.func_state.Q (x, y, state,
ug_state)))
           (λ(yd, state, ug_state).
             TD_side_rule_Interp_solve_rec_c r T (TD_side_upd_rule.func_state.E (x, g yd, sides, state, ug_state)))
     | QueryG y g ⇒
         (let _ = trace_event STR ''solver'' (λ_. (Ev_Query_Global x y :: ('x, 'g, 'd) solver_event)) in
          let _ = trace_event STR ''solver'' (λ_. (Ev_Add_Infl (Inr y) x :: ('x, 'g, 'd) solver_event)) in
          let _ = trace_event STR ''solver''
                    (λ_. (Ev_Answer_Global x y ((σ state) (Inr y)) :: ('x, 'g, 'd) solver_event)) in
          TD_side_rule_Interp_solve_rec_c r T (TD_side_upd_rule.func_state.E (x, g ((σ state) (Inr y)), sides,
            state(| infl := fminsert (infl state) (Inr y) x|), ug_state)))
     | Side y d t ⇒
         (let _ = trace_event STR ''solver'' (λ_. (Ev_Side x y d :: ('x, 'g, 'd) solver_event)) in
          let d = sides y ⊔ d in
          let sides = sides(y := d) in
          let (upd, ug_state) = update_global_of r (σ state (Inr y)) x y d ug_state in
          case upd of None ⇒ TD_side_rule_Interp_solve_rec_c r T (TD_side_upd_rule.func_state.E (x, t, sides, state,
ug_state))
           | Some d ⇒
             let _ = trace_event STR ''solver''
                       (λ_. (Ev_Update_Global x y ((σ state) (Inr y) = ⊥) ((σ state) (Inr y)) d
                         :: ('x, 'g, 'd) solver_event)) in
             let (infl', stabl') = destab_opt (Inr y) (infl state) (stabl state) (called state) in
             TD_side_rule_Interp_solve_rec_c r T (TD_side_upd_rule.func_state.E (x, t, sides, state(| infl := infl',
stabl := stabl',
               σ := (σ state)(Inr y := d)|), ug_state)))))"
  by (rule trans[OF TD_side_rule_Interp.solve_rec_c.simps])
     (simp only: trace_event_def Let_def)

definition traced_solve_rec_c
  where [code del]: "traced_solve_rec_c r T s = TD_side_rule_Interp_solve_rec_c r T s"

lemmas traced_solve_rec_c_code [code] =
  solve_rec_c_traced[folded traced_solve_rec_c_def traced_destab_opt_def]

definition traced_solve_c ::
    "globals_rule ⇒ ('x, 'g, 'd::{bounded_semilattice_sup_bot,warrowing}) eqsT ⇒ 'x
       ⇒ ('x set × ('x + 'g ⇒ 'd)) option"
  where [code del]: "traced_solve_c r T x = TD_side_rule_Interp_solve_c r T x"

lemmas traced_solve_c_code [code] =
  TD_side_rule_Interp.solve_c_def[folded traced_solve_c_def traced_solve_rec_c_def]
```

One solve, from its root unknown to its result, between a start and a stop event. It is `solve_c` in the logic; a traced caller names it in place of `solve_c`.

```
definition solve_c_traced ::
    "globals_rule ⇒ ('x, 'g, 'd::{bounded_semilattice_sup_bot,warrowing}) eqsT ⇒ 'x
       ⇒ ('x set × ('x + 'g ⇒ 'd)) option" where
  "solve_c_traced r T x =
    (let _ = trace_event STR ''solver'' (λ_. (Ev_Start x :: ('x, 'g, 'd) solver_event)) in
     let res = TD_side_rule_Interp_solve_c r T x in
     let _ = trace_event STR ''solver'' (λ_. (Ev_Stop :: ('x, 'g, 'd) solver_event)) in
     res)"

lemma solve_c_traced_eq: "solve_c_traced r T x = TD_side_rule_Interp_solve_c r T x"
  by (simp add: solve_c_traced_def trace_event_def)

lemma solve_c_traced_code [code]:
  fixes T :: "('x, 'g, 'd::{bounded_semilattice_sup_bot,warrowing}) eqsT"
```

```
  shows "solve_c_traced r T x =
    (let _ = trace_event STR ''solver'' (λ_. (Ev_Start x :: ('x, 'g, 'd) solver_event)) in
     let res = traced_solve_c r T x in
     let _ = trace_event STR ''solver'' (λ_. (Ev_Stop :: ('x, 'g, 'd) solver_event)) in
     res)"
  by (simp only: solve_c_traced_def traced_solve_c_def)

end
```

Chapter 4

Voblint_CFG

4.1 CFG_Def

```
theory CFG_Def
  imports "Voblint_VIMP.VIMP_Proc" "HOL-Library.Countable"
begin
```

Procedure-aware control-flow graph

A `cfg` is an interprocedural control-flow graph over two disjoint transition relations, one per phenomenon:

- `intra` — ordinary context-preserving flow, labelled by an `edge_action` that is a total within-context store transformer. Executed by `edge_step`.
- `calls` — context-crossing calls, labelled by a `call_action` and carrying the call-site, the callee entry node, and the continuation node.

Nodes are procedure-aware: `FunctionEntry p` and `FunctionResult p` are first-class nodes, not keyed side slots. There is no global exit node and no call/return matching side relation: a call edge already names its callee entry and its continuation, and a return is an ordinary intra edge into `FunctionResult p`. Calls carry incompatible typing from intra flow, so `edge_step` has no call case and `intra` can never carry a call — by typing, not by a side condition.

Translation from VIMP to CFG and the collecting/equation layers ride on this structure.

Nodes

```
datatype cfg_node =
    Statement (node_stmt: nat)
  | FunctionEntry (entry_proc: pname)
  | FunctionResult (result_proc: pname)

instance cfg_node :: countable
  by countable_datatype
```

The solver unknown for a program point is a CFG node. Analysis-facing code keeps the short name `pp` for it; a return node is `FunctionResult p`, a callee entry `FunctionEntry p`, an ordinary location `Statement n`.

```
type_synonym pp = cfg_node
```

Edge actions and call actions

`edge_action` labels intra flow. It has no call constructor: every action is a total store transformer within one activation, so an intra edge can never denote a call. A `EA_Ret e p` writes the return value into `ret_var` in the callee's own context; its graph target is `FunctionResult p` (enforced by `wf_cfg`), which is why return summarisation is ordinary predecessor folding over `FunctionResult p`.

`EA_Body p` labels the one edge leaving `FunctionEntry p`, so an activation's first step inside the callee is distinguishable from an ordinary `EA_Nop`. Concretely it is the identity: a VIMP procedure declares no locals beyond its formals, which the call already bound. It exists so that an analysis has a transition to attach procedure-entry work to — Goblint runs `body` on exactly this edge — and it carries the procedure name because that work is per-procedure.

`call_action` labels call edges. `CallEdge dst formals args` records the caller destination variable, the callee's formal parameter names, and the actual arguments; the callee identity and the continuation are the two target nodes of the `calls` tuple, not payload of the action. Carrying `formals` on the edge lets the caller-side entry transfer `call_enter` bind actuals to formals without consulting a procedure table, keeping the trace kernel independent of the source declaration environment.

```
datatype edge_action =
    EA_Nop
  | EA_Assign   (ea_var: vname) (ea_rhs: exp)
  | EA_Special  (ea_special_op: special_call) (ea_special_dst: vname)
  | EA_Assume    (ea_cond: exp)
  | EA_AssumeNot (ea_cond: exp)
  | EA_Body     (ea_body_proc: pname)
  | EA_Ret      (ea_ret_val: "exp option") (ea_ret_proc: pname)
  | EA_Check    (ea_check_label: check_label) (ea_check_cond: exp)

datatype call_action =
    CallEdge (ce_dst: "vname option") (ce_formals: "vname list") (ce_args: "exp list")
```

```
instance edge_action :: countable
  by countable_datatype

instance call_action :: countable
  by countable_datatype
```

`call_info` is the call-boundary metadata an interprocedural transfer may consult: the caller destination, the callee, its formals, and the actual arguments. Goblint's `Analyses.Spec` hands the same four to both halves of its return protocol – `combine_env man lval fexp f args fc au ask` and `combine_assign` with the identical argument shape – so a transfer that only ever sees the caller and callee-exit states is strictly weaker than that interface. The call site and the callee-exit point are the combine tree's own two program-point arguments, so they are not duplicated here.

Goblint's `fexp` has no separate counterpart: VIMP calls a statically named procedure, so `ci_callee` already carries what the call expression identifies. The callee context `fc` belongs to the routed-analysis layer rather than to a transfer-record field, and `Queries.ask` over the callee exit has none: there is no query bus.

```
record call_info =
  ci_dst     :: "vname option"
  ci_callee  :: pname
  ci_formals :: "vname list"
  ci_args    :: "exp list"

definition call_info_of :: "call_action ⇒ pname ⇒ call_info" where
  "call_info_of ca p =
     (| ci_dst = ce_dst ca, ci_callee = p,
       ci_formals = ce_formals ca, ci_args = ce_args ca |)"

lemma call_info_of_simps [simp]:
  "ci_dst (call_info_of ca p) = ce_dst ca"
  "ci_callee (call_info_of ca p) = p"
  "ci_formals (call_info_of ca p) = ce_formals ca"
  "ci_args (call_info_of ca p) = ce_args ca"
  by (simp_all add: call_info_of_def)
```

CFG record: two relations

`calls` is a four-place relation `(call_site, act, callee_entry, continuation)`: the callee entry and the continuation are ordinary nodes, kept explicit so no matching side relation is needed. The flat (call-free) CFG is exactly the `calls = {}` fragment, so every fact quantified over `intra` is verbatim a flat-CFG fact.

```
record cfg =
  intra     :: "(cfg_node × edge_action × cfg_node) set"
  calls     :: "(cfg_node × call_action × cfg_node × cfg_node) set"
  cfg_entry :: cfg_node
```

Intra edge execution

`special_step` is `special_call`'s own step function, factored out of `edge_step` so that extending the special-call vocabulary only touches this function (via `special_result`, shared with the source small-step semantics) and `special_step_nonempty` below, not every generic lemma stated over `edge_action`. `Nondet_Int` havocs the destination; `Min`/`Max` write exactly one, deterministic value, so no single equation characterizes every `sc` uniformly any more – callers reasoning about an as-yet-unclassified `sc` case-split on it instead.

```
fun special_step :: "special_call ⇒ vname ⇒ store ⇒ store set" where
  "special_step sc x s = {s(x := v) |v. special_result sc s v}"

lemma special_step_nonempty [simp]: "special_step sc x s ≠ {}"
  by (cases sc) auto
```

`edge_step` is the single primitive semantics of an intra action. It is defined for every constructor and has no call case; guards are the only source of an empty result.

```
fun edge_step :: "edge_action ⇒ store ⇒ store set" where
  "edge_step EA_Nop s = {s}"
| "edge_step (EA_Assign x a) s = {s(x := ⟦a⟧e s)}"
| "edge_step (EA_Special sc x) s = special_step sc x s"
| "edge_step (EA_Assume b) s = (if truthy (⟦b⟧e s) then {s} else {})"
| "edge_step (EA_AssumeNot b) s = (if truthy (⟦b⟧e s) then {} else {s})"
| "edge_step (EA_Body p) s = {s}"
| "edge_step (EA_Ret e p) s =
     {s(ret_var := (case e of None ⇒ s ret_var | Some a ⇒ ⟦a⟧e s))}"
| "edge_step (EA_Check l c) s = {s}"
```

Intra-only execution paths

`intra_path` is the reflexive-transitive closure of `edge_step` along `intra` edges, on `(node, store)` pairs. It deliberately excludes `calls`: an intra path never pushes or pops an activation, so it lifts to the collecting semantics without any stack-representation side condition, and to a stack-preserving CFG run at any stack.

One intra transition on a `(node, store)` pair. `intra_path` is its `star`, so reflexivity, transitivity and the induction rule come from `HOL-IMP.Star` rather than from a bespoke closure. The name avoids `intra_step`, which is the *source*-level `pstep` fragment.

```
definition cfg_intra_step :: "cfg ⇒ cfg_node × store ⇒ cfg_node × store ⇒ bool" where
  "cfg_intra_step g p q ⟷
     (∃a. (fst p, a, fst q) ∈ intra g ∧ snd q ∈ edge_step a (snd p))"

abbreviation intra_path :: "cfg ⇒ cfg_node × store ⇒ cfg_node × store ⇒ bool" where
  "intra_path g ≡ star (cfg_intra_step g)"

lemma cfg_intra_stepI [intro]:
  "(u, a, v) ∈ intra g ⟹ s' ∈ edge_step a s ⟹ cfg_intra_step g (u, s) (v, s')"
  by (auto simp: cfg_intra_step_def)

lemma cfg_intra_stepE [elim]:
  assumes "cfg_intra_step g (u, s) (v, s')"
  obtains a where "(u, a, v) ∈ intra g" "s' ∈ edge_step a s"
  using assms by (auto simp: cfg_intra_step_def)

lemma intra_path_single:
  "(u, a, v) ∈ intra g ⟹ s' ∈ edge_step a s ⟹ intra_path g (u, s) (v, s')"
  by (intro star_step1 cfg_intra_stepI)

lemma intra_path_nop:
  "(u, EA_Nop, v) ∈ intra g ⟹ intra_path g (u, s) (v, s)"
  by (rule intra_path_single[where a = EA_Nop]) simp_all
```

Node set (derived, not stored)

`cfg_nodes` is reconstructed from the endpoints of `intra`, every node occurring in `calls`, and `cfg_entry`. It is not a record field.

```
definition cfg_nodes :: "cfg ⇒ cfg_node set" where
  "cfg_nodes g =
     {u. ∃a v. (u, a, v) ∈ intra g}
     ∪ {v. ∃u a. (u, a, v) ∈ intra g}
     ∪ {u. ∃act ce after. (u, act, ce, after) ∈ calls g}
     ∪ {ce. ∃u act after. (u, act, ce, after) ∈ calls g}
     ∪ {after. ∃u act ce. (u, act, ce, after) ∈ calls g}
     ∪ {cfg_entry g}"
```

Whole-program completion is the entry procedure's `FunctionResult`. A graph whose entry is not a procedure entry is never compiled; the fallback keeps the function total and aborts in generated code.

```
definition cfg_exit :: "cfg ⇒ cfg_node" where
  "cfg_exit g =
    (case cfg_entry g of
       FunctionEntry p ⇒ FunctionResult p
     | n ⇒ Code.abort (STR ''cfg_exit: entry is not a procedure entry'') (λ_. n))"
```

Well-formedness

`wf_cfg` is structural only: call edges enter procedure-entry nodes; no intra edge enters a procedure-entry node (so a callee is reached only across a call); and a return action lands on the matching procedure result. No compiler-correctness or source-semantic property is included.

```
definition wf_cfg :: "cfg ⇒ bool" where
  "wf_cfg g ⟷
     (∀u act ce after. (u, act, ce, after) ∈ calls g ⟶ (∃p. ce = FunctionEntry p))
   ∧ (∀u a v. (u, a, v) ∈ intra g ⟶ (∀p. v ≠ FunctionEntry p))
   ∧ (∀u e p v. (u, EA_Ret e p, v) ∈ intra g ⟶ v = FunctionResult p)"
```

`calls_source_unique` is a separate, optional side condition: one call edge per call site. `wf_cfg` does not include it, and the collecting semantics never needs it — both the return clause and the context semantics name an edge by its effect. A solved routed table does need it, to match the edge a return reads against the one its callee was entered through; compiled programs satisfy it by construction.

```
definition calls_source_unique :: "cfg ⇒ bool" where
```

```
  "calls_source_unique g ⟷
     (∀u ca1 ce1 af1 ca2 ce2 af2.
        (u, ca1, ce1, af1) ∈ calls g ⟶ (u, ca2, ce2, af2) ∈ calls g
        ⟶ ca1 = ca2 ∧ ce1 = ce2 ∧ af1 = af2)"

lemma calls_source_unique_edgesD:
  assumes "calls_source_unique g"
    and "(u, ca1, ce1, af1) ∈ calls g" and "(u, ca2, ce2, af2) ∈ calls g"
  shows "ca1 = ca2" and "ce1 = ce2" and "af1 = af2"
  using assms unfolding calls_source_unique_def by blast+
```

Structural invariants

What holds of a graph's node set no matter where the graph came from: an edge's endpoints are nodes, and finitely many edges give finitely many nodes. Both are read off the definition of `cfg_nodes`, which is why they need no well-formedness assumption.

```
lemma intra_endpoints_in_nodes:
  assumes "(u, a, v) ∈ intra g"
  shows "u ∈ cfg_nodes g" and "v ∈ cfg_nodes g"
  using assms by (auto simp: cfg_nodes_def)

lemma call_endpoints_in_nodes:
  assumes "(u, act, ce, after) ∈ calls g"
  shows "u ∈ cfg_nodes g" and "ce ∈ cfg_nodes g" and "after ∈ cfg_nodes g"
  using assms by (auto simp: cfg_nodes_def)
```

Each of the five `cfg_nodes` disjuncts is a projection of `intra g` or `calls g`, so it inherits their finiteness as a finite image; the sixth is a singleton.

```
lemma cfg_nodes_finite:
  assumes "finite (intra g)" and "finite (calls g)"
  shows "finite (cfg_nodes g)"
proof -
  have "{u. ∃a v. (u, a, v) ∈ intra g} = (λ(u, a, v). u) ‘ intra g"
    and "{v. ∃u a. (u, a, v) ∈ intra g} = (λ(u, a, v). v) ‘ intra g"
    by force+
  moreover
  have "{u. ∃act ce after. (u, act, ce, after) ∈ calls g} = (λ(u, act, ce, after). u) ‘ calls g"
    and "{ce. ∃u act after. (u, act, ce, after) ∈ calls g} = (λ(u, act, ce, after). ce) ‘ calls g"
    and "{after. ∃u act ce. (u, act, ce, after) ∈ calls g}
           = (λ(u, act, ce, after). after) ‘ calls g"
    by force+
  ultimately show ?thesis
    unfolding cfg_nodes_def using assms by simp
qed
```

Executable orders

Structural orders make `sorted_list_of_set` a deterministic executable enumeration of intra and call relations. They affect only solver representation, not CFG semantics. `exp` already derives `linorder` in `Voblint_VIMP.VIMP_Syntax`.

```
derive linorder special_call
derive linorder edge_action
derive linorder call_action
derive linorder cfg_node
```

Executable enumeration

Stable list views for the TD bridge: the intra and call edge sets sorted by their structural order. Both guard on `finite (intra g)`/`finite (calls g)`: every compiled graph satisfies it, so the `Code.abort` branch never fires in practice, and names the violation instead of failing on an uninformative `sorted_list_of_set` pattern-match error. Placed here rather than in the framework session's own equation-generation enumerations, so any consumer of this session alone — a compiled graph's `finite_cfg` interpretation among them — gets them for free.

```
definition cfg_intra_list :: "cfg ⇒ (cfg_node × edge_action × cfg_node) list" where
  "cfg_intra_list g =
     (if finite (intra g) then sorted_list_of_set (intra g)
      else Code.abort (STR ''cfg_intra_list: infinite intra edge set'') (λ_. []))"

lemma cfg_intra_list_code [code]:
  "cfg_intra_list g = sorted_list_of_set (intra g)"
  unfolding cfg_intra_list_def by (cases "finite (intra g)") auto

definition cfg_calls_list ::
```

```
    "cfg ⇒ (cfg_node × call_action × cfg_node × cfg_node) list" where
  "cfg_calls_list g =
     (if finite (calls g) then sorted_list_of_set (calls g)
      else Code.abort (STR ''cfg_calls_list: infinite calls edge set'') (λ_. []))"

lemma cfg_calls_list_code [code]:
  "cfg_calls_list g = sorted_list_of_set (calls g)"
  unfolding cfg_calls_list_def by (cases "finite (calls g)") auto

lemma set_cfg_intra_list [simp]:
  "finite (intra g) ⟹ set (cfg_intra_list g) = intra g"
  unfolding cfg_intra_list_def by simp

lemma set_cfg_calls_list [simp]:
  "finite (calls g) ⟹ set (cfg_calls_list g) = calls g"
  unfolding cfg_calls_list_def by simp
```

Every `cfg_nodes` element, listed once: the intra endpoints, the call-site/ callee-entry/continuation triples, and the distinguished entry, deduplicated. This is the canonical finite node enumeration a public per-node result table draws its key domain from, distinct from `cfg_intra_list`'s edge-indexed view.

```
definition cfg_node_list :: "cfg ⇒ cfg_node list" where
  "cfg_node_list g =
     remdups
       (concat (map (λ(u, a, v). [u, v]) (cfg_intra_list g)) @
        concat (map (λ(u, act, ce, after). [u, ce, after]) (cfg_calls_list g)) @
        [cfg_entry g])"

lemma set_cfg_node_list [simp]:
  assumes "finite (intra g)" and "finite (calls g)"
  shows "set (cfg_node_list g) = cfg_nodes g"
  unfolding cfg_node_list_def cfg_nodes_def
  using set_cfg_intra_list[OF assms(1)] set_cfg_calls_list[OF assms(2)]
  by force
```

A `cfg` whose two transition relations are finite. Bundles the two finiteness facts every enumeration and every compiled-graph interpretation needs, so a caller states one assumption instead of two, and `finite_nodes` becomes derivable rather than re-proved at each instance.

```
locale finite_cfg =
  fixes g :: cfg
  assumes finite_intra [intro, simp]: "finite (intra g)"
    and finite_calls [intro, simp]: "finite (calls g)"
begin

lemma finite_nodes [simp]: "finite (cfg_nodes g)"
  by (rule cfg_nodes_finite) simp_all

end
```

Expressions observed by an edge, shared by diagnostics and transfer footprints.

```
fun edge_expressions :: "edge_action ⇒ exp list" where
  "edge_expressions (EA_Assign x e) = [e]"
| "edge_expressions (EA_Assume e) = [e]"
| "edge_expressions (EA_AssumeNot e) = [e]"
| "edge_expressions (EA_Check l e) = [e]"
| "edge_expressions (EA_Ret (Some e) p) = [e]"
| "edge_expressions (EA_Special (Min a b) x) = [a, b]"
| "edge_expressions (EA_Special (Max a b) x) = [a, b]"
| "edge_expressions _ = []"

end
```

4.2 CFG_Transfer

```
theory CFG_Transfer
  imports CFG_Def
begin
```

What each kind of edge does to the store

Three transfers, one per thing an edge can do. `edge_collect` runs an ordinary edge on a whole set of stores at once, by lifting the single-store `edge_step` pointwise — derived from the concrete step rather than restated, so the two cannot drift apart. `call_enter` builds the callee's opening store: evaluate the actuals in the caller, reset the locals, bind the formals. `combine_collect` does the reverse at a return: keep the callee's globals, restore the caller's locals, and write the callee's return value into the destination.

All the payload comes from the edge itself, so none of these needs the procedure table.

```
definition edge_collect :: "edge_action ⇒ store set ⇒ store set" where
  "edge_collect a S = {t. ∃s∈S. t ∈ edge_step a s}"

lemma edge_collect_simps [simp]:
  "edge_collect EA_Nop S = S"
  "edge_collect (EA_Assign x a) S = {s(x := ⟦a⟧e s) | s. s ∈ S}"
  "edge_collect (EA_Special sc x) S = {t. ∃s∈S. t ∈ special_step sc x s}"
  "edge_collect (EA_Assume b) S = {s. s ∈ S ∧ truthy (⟦b⟧e s)}"
  "edge_collect (EA_AssumeNot b) S = {s. s ∈ S ∧ ¬ truthy (⟦b⟧e s)}"
  "edge_collect (EA_Body p) S = S"
  "edge_collect (EA_Ret e p) S =
     {s(ret_var := (case e of None ⇒ s ret_var | Some a ⇒ ⟦a⟧e s)) | s. s ∈ S}"
  "edge_collect (EA_Check l c) S = S"
  unfolding edge_collect_def by (auto split: if_splits)

lemma edge_collect_single:
  "edge_collect a {s} = edge_step a s"
  by (cases a) (auto split: if_splits)

lemma edge_collect_empty_set [simp]: "edge_collect a {} = {}"
  by (cases a) auto

lemma edge_collect_mono:
  assumes "S ⊆ T"
  shows "edge_collect a S ⊆ edge_collect a T"
  using assms by (cases a) auto
```

Return-value transfer

Return-value rehydration at the caller: write the callee's `ret_var` into the destination over the combined store (callee globals, caller locals). It is fixed by the call's destination `dst`, which the `CallEdge` already records — no side lookup.

```
definition combine_collect :: "(vname ⇒ bool) ⇒ vname option ⇒ store ⇒ store ⇒ store" where
  "combine_collect 𝒢 dst s t = combine_assign dst (t ret_var) (combine_env 𝒢 s t)"
```

Call-entry transfer

Caller-side entry transfer at a call. The actuals are evaluated in the caller store, the callee locals are reset (`enter_state`, globals preserved), and the resulting values are bound to the callee formals. All payload comes from the `CallEdge`, so the transfer needs no procedure table. This is exactly the entered store produced by the source `pstep Call` rule.

```
definition call_enter :: "(vname ⇒ bool) ⇒ call_action ⇒ store ⇒ store" where
  "call_enter 𝒢 ca s =
     (case ca of CallEdge dst pars actuals ⇒
        enter_binding 𝒢 0 aval pars actuals s)"

lemma call_enter_CallEdge:
  "call_enter 𝒢 (CallEdge dst pars actuals) s
     = bind_formals pars (map (λe. ⟦e⟧e s) actuals) (enter_state 𝒢 s)"
  by (simp add: call_enter_def enter_binding_def enter_state_def)
```

A parameterless call is exactly `enter_state`: no actuals to evaluate and no formals to bind.

```
lemma call_enter_Nil [simp]:
  "call_enter 𝒢 (CallEdge dst [] []) s = enter_state 𝒢 s"
  by (simp add: call_enter_CallEdge)
```

What an edge reads and what it may change

An edge reads the variables of its expressions. It may change only the variable it assigns, and a return only its result slot.

```
definition edge_reads :: "edge_action ⇒ vname list" where
  "edge_reads a = sorted_list_of_set (⋃ (exp_vnames ` set (edge_expressions a)))"

fun edge_writes :: "edge_action ⇒ vname list" where
  "edge_writes (EA_Assign x a) = [x]"
| "edge_writes (EA_Special sc x) = [x]"
| "edge_writes (EA_Ret e p) = [ret_var]"
| "edge_writes _ = []"
```

The frame each concrete step respects: a name outside its write set keeps its value. A call entry may only bind its formals, and a return takes every global from the callee except the destination it assigns.

```
lemma edge_step_frame:
  "t ∈ edge_step a s ⟹ x ∉ set (edge_writes a) ⟹ t x = s x"
  by (cases a) (auto split: if_splits)

lemma call_enter_frame:
  "𝒢 x ⟹ x ∉ set pars ⟹ call_enter 𝒢 (CallEdge dst pars args) s x = s x"
  by (simp add: call_enter_CallEdge bind_formals_other enter_state_def)

lemma combine_collect_frame:
  "𝒢 x ⟹ dst ≠ Some x ⟹ combine_collect 𝒢 dst s t x = t x"
  by (cases dst) (auto simp: combine_collect_def)
```

Finite global footprints

The declared globals among those names, as duplicate-free lists a transfer folds over to read or publish them.

```
definition global_names_in :: "(vname ⇒ bool) ⇒ vname list ⇒ vname list" where
  "global_names_in G xs = remdups (filter G xs)"

lemma set_global_names_in [simp]:
  "set (global_names_in G xs) = {x ∈ set xs. G x}"
  by (simp add: global_names_in_def)

lemma distinct_global_names_in [simp]: "distinct (global_names_in G xs)"
  by (simp add: global_names_in_def)

definition globals_in_exp :: "(vname ⇒ bool) ⇒ exp ⇒ vname list" where
  "globals_in_exp G e = global_names_in G (sorted_list_of_set (exp_vnames e))"

definition edge_global_reads :: "(vname ⇒ bool) ⇒ edge_action ⇒ vname list" where
  "edge_global_reads G a = global_names_in G (edge_reads a)"

definition edge_global_writes :: "(vname ⇒ bool) ⇒ edge_action ⇒ vname list" where
  "edge_global_writes G a = global_names_in G (edge_writes a)"

definition call_global_reads :: "(vname ⇒ bool) ⇒ exp list ⇒ vname list" where
  "call_global_reads G es = global_names_in G (sorted_list_of_set (⋃ (exp_vnames ` set es)))"

lemma set_edge_global_writes [simp]:
  "set (edge_global_writes G a) = {x ∈ set (edge_writes a). G x}"
  by (simp add: edge_global_writes_def)

lemma distinct_global_footprints [simp]:
  "distinct (globals_in_exp G e)"
  "distinct (edge_global_reads G a)"
  "distinct (edge_global_writes G a)"
  "distinct (call_global_reads G es)"
  by (simp_all add: globals_in_exp_def edge_global_reads_def edge_global_writes_def
    call_global_reads_def)

end
```

4.3 CFG_Prune

```
theory CFG_Prune
  imports CFG_Transfer
begin
```

Procedure-aware structural reachability

`cfg_succ_rel` is the derived structural dependency graph the analysis pruning and cone proofs run on — not the concrete execution relation. It has four sources, induced by `intra g` and `calls g`:

- INTRA: an ordinary edge `(u, a, v) ∈ intra g` gives `u → v` (ordinary flow).
- ENTRY: a call `(c, ca, FunctionEntry p, k) ∈ calls g` gives `c → FunctionEntry p` — the callee entry's abstract state depends on the caller state routed through the analysis's own enter operation.
- `COMB_CALLER`: the same call gives `c → k` — the continuation depends on the saved caller state via `combine_collect`. This is not a concrete execution edge; execution does not skip the callee.
- `COMB_RESULT`: the same call gives `FunctionResult p → k` — the continuation depends on the callee's result, `combine_collect`'s callee-exit argument.

  `c → k` and `FunctionResult p → k` are combine dependencies of the analysis, kept visibly separate from `intra g`. They are never added to `intra g`.

The structural successor relation

```
definition cfg_succ_rel :: "cfg ⇒ (cfg_node × cfg_node) set" where
  "cfg_succ_rel g =
     {(u, v) | u a v. (u, a, v) ∈ intra g}
   ∪ {(c, ce) | c ca ce k. (c, ca, ce, k) ∈ calls g}
   ∪ {(c, k) | c ca ce k. (c, ca, ce, k) ∈ calls g}
   ∪ {(FunctionResult p, k) | c ca p k. (c, ca, FunctionEntry p, k) ∈ calls g}"

lemma cfg_succ_rel_intra:
  "(u, a, v) ∈ intra g ⟹ (u, v) ∈ cfg_succ_rel g"
  unfolding cfg_succ_rel_def by blast

lemma cfg_succ_rel_entry:
  "(c, ca, ce, k) ∈ calls g ⟹ (c, ce) ∈ cfg_succ_rel g"
  unfolding cfg_succ_rel_def by blast

lemma cfg_succ_rel_comb_caller:
  "(c, ca, ce, k) ∈ calls g ⟹ (c, k) ∈ cfg_succ_rel g"
  unfolding cfg_succ_rel_def by blast

lemma cfg_succ_rel_comb_result:
  "(c, ca, FunctionEntry p, k) ∈ calls g ⟹ (FunctionResult p, k) ∈ cfg_succ_rel g"
  unfolding cfg_succ_rel_def by blast

lemma cfg_succ_rel_cases:
  assumes "(y, z) ∈ cfg_succ_rel g"
  obtains (INTRA) a where "(y, a, z) ∈ intra g"
    | (ENTRY) ca k where "(y, ca, z, k) ∈ calls g"
    | (COMB_CALLER) ca ce where "(y, ca, ce, z) ∈ calls g"
    | (COMB_RESULT) c ca p k where "(c, ca, FunctionEntry p, k) ∈ calls g"
                                   "y = FunctionResult p" "z = k"
  using assms unfolding cfg_succ_rel_def by blast
```

Reachability: reflexive-transitive closure of the successor relation

Reachability over that successor relation. Because the relation already relates a caller to its continuation and a callee result to the same continuation, this closure crosses procedure boundaries without any separate interprocedural notion.

```
definition cfg_reaches :: "cfg ⇒ cfg_node ⇒ cfg_node ⇒ bool" where
  "cfg_reaches g v v0 ⟷ (v, v0) ∈ (cfg_succ_rel g)*"
```

Reachability inside one activation

The same closure without entering callees: a call site steps straight to its continuation. These are the steps a routed equation reads backwards, so this is the reachability the solver's coverage argument needs.

```
definition local_succ_rel :: "cfg ⇒ (cfg_node × cfg_node) set" where
  "local_succ_rel g =
     {(u, v). ∃a. (u, a, v) ∈ intra g}
     ∪ {(u, k). ∃ca q. (u, ca, FunctionEntry q, k) ∈ calls g}"

definition local_reaches :: "cfg ⇒ cfg_node ⇒ cfg_node ⇒ bool" where
  "local_reaches g u v ⟷ (u, v) ∈ (local_succ_rel g)*"

lemma local_reaches_refl [simp]: "local_reaches g v v"
  by (simp add: local_reaches_def)

lemma local_reaches_trans:
  "local_reaches g u v ⟹ local_reaches g v w ⟹ local_reaches g u w"
  unfolding local_reaches_def by (rule rtrancl_trans)

lemma local_reaches_intra_step:
  "(u, a, v) ∈ intra g ⟹ local_reaches g v w ⟹ local_reaches g u w"
  unfolding local_reaches_def local_succ_rel_def
  by (rule converse_rtrancl_into_rtrancl) blast+

lemma local_reaches_comb_step:
  "(u, ca, FunctionEntry q, k) ∈ calls g ⟹ local_reaches g k w ⟹ local_reaches g u w"
  unfolding local_reaches_def local_succ_rel_def
  by (rule converse_rtrancl_into_rtrancl) blast+

lemma local_reaches_cfg_reaches:
  "local_reaches g u v ⟹ cfg_reaches g u v"
  unfolding local_reaches_def cfg_reaches_def
proof (induction rule: rtrancl_induct)
  case (step y z)
  then have "(y, z) ∈ cfg_succ_rel g"
    unfolding local_succ_rel_def by (auto simp: cfg_succ_rel_intra cfg_succ_rel_comb_caller)
  with step.IH show ?case by (rule rtrancl_into_rtrancl)
qed simp

end
```

4.4 CFG_Exec

```
theory CFG_Exec
  imports CFG_Transfer
begin
```

How a control-flow graph runs

`cstep` executes an arbitrary CFG over a node, a store, and a stack of suspended activations. Each `cframe` records where to resume, which variable receives the result, and the caller's store. There are three rules, one per graph phenomenon: follow an `intra` edge and apply its transfer; follow a `calls` edge, enter the callee and push a frame; at a `FunctionResult`, pop the top frame and combine the callee's store into the caller's.

A return follows no edge. The stack supplies the continuation, so one `FunctionResult` node serves every caller of a procedure and recursion needs no duplicated nodes. Nothing here mentions the compiler: this is the execution of any graph, just as `valid_activation_trace` is the activation-trace semantics of any graph.

```
type_synonym cframe = "cfg_node × vname option × store"
type_synonym cconf = "cfg_node × store × cframe list"

inductive cstep :: "(vname ⇒ bool) ⇒ cfg ⇒ cconf ⇒ cconf ⇒ bool"
    ("(_,_ ⊢/ _ →c/ _)" [51, 51, 51, 51] 50) for 𝒢 and g where
  Intra [intro]:
    "(u, a, v) ∈ intra g ⟹ s' ∈ edge_step a s ⟹
     𝒢, g ⊢ (u, s, stk) →c (v, s', stk)"
| Call [intro]:
    "(u, CallEdge dst pars actuals, FunctionEntry q, cont) ∈ calls g ⟹
     𝒢, g ⊢ (u, s, stk)
       →c (FunctionEntry q, call_enter 𝒢 (CallEdge dst pars actuals) s, (cont, dst, s) # stk)"
| Return [intro]:
    "𝒢, g ⊢ (FunctionResult q, t, (cont, dst, caller) # stk)
       →c (cont, combine_collect 𝒢 dst caller t, stk)"


abbreviation csteps :: "(vname ⇒ bool) ⇒ cfg ⇒ cconf ⇒ cconf ⇒ bool"
    ("(_,_ ⊢/ _ →c*/ _)" [51, 51, 51, 51] 50)
  where "csteps 𝒢 g x y ≡ star (cstep 𝒢 g) x y"
```

One inversion rule, because the three clauses are told apart by the edge taken rather than by the shape of the configuration. It stays plain `[elim]` so the classical reasoner does not split every `cstep` hypothesis three ways before trying anything else.

```
inductive_cases cstep_E [elim]: "𝒢, g ⊢ (u, s, stk) →c y"
```

Single-step and small-step lemmas

Introduction lemmas for the store-preserving edge actions the simulation steps through: body, assume and assume-not.

```
lemma cstep_body:
  assumes "(u, EA_Body p, v) ∈ intra g"
  shows "𝒢, g ⊢ (u, s, stk) →c (v, s, stk)"
  by (rule cstep.Intra[OF assms]) simp

lemma cstep_assume:
  assumes "(u, EA_Assume b, v) ∈ intra g" and "truthy (⟦b⟧e s)"
  shows "𝒢, g ⊢ (u, s, stk) →c (v, s, stk)"
  by (rule cstep.Intra[OF assms(1)]) (use assms(2) in simp)

lemma cstep_assume_not:
  assumes "(u, EA_AssumeNot b, v) ∈ intra g" and "¬ truthy (⟦b⟧e s)"
  shows "𝒢, g ⊢ (u, s, stk) →c (v, s, stk)"
  by (rule cstep.Intra[OF assms(1)]) (use assms(2) in simp)
```

Intra-only paths as stack-preserving runs

An `intra_path` is a `cstep` run at any stack: every `cfg_intra_step` is a `cstep.Intra`, which passes the stack through untouched. Callers that only need to move along local edges can therefore reason on `(node, store)` pairs and lift the result here.

```
lemma intra_path_imp_cstep_star:
  "intra_path g x y ⟹ 𝒢, g ⊢ (fst x, snd x, stk) →c* (fst y, snd y, stk)"
proof (induction rule: star.induct)
  case (refl a) show ?case by simp
```

```
next
  case (step a b c)
  obtain ua sa where a: "a = (ua, sa)" by (cases a)
  obtain ub sb where b: "b = (ub, sb)" by (cases b)
  from step.hyps(1) a b obtain e where "(ua, e, ub) ∈ intra g" "sb ∈ edge_step e sa" by auto
  hence "𝒢, g ⊢ (ua, sa, stk) →c (ub, sb, stk)" by (rule cstep.Intra)
  with step.IH a b show ?case by (auto intro: star.step)
qed
```

Activation-stack matching

Both stacks record procedure activations. `act_frames` removes the CFG-only continuation component and retains the caller store and destination used by the source.

```
fun act_frames :: "frame list ⇒ (store × vname option) list" where
  "act_frames [] = []"
| "act_frames (Frame s d # frs) = (s, d) # act_frames frs"

definition cframe_act :: "cframe ⇒ store × vname option" where
  "cframe_act cf = (case cf of (cont, d, s) ⇒ (s, d))"
```

`frames_match` ties a source frame stack to a CFG activation stack: the activation frames agree, in order, on caller store and destination (the continuation node is the CFG's own bookkeeping and is not visible to the source).

```
definition frames_match :: "frame list ⇒ cframe list ⇒ bool" where
  "frames_match frs stk = (act_frames frs = map cframe_act stk)"

lemma frames_match_Nil [simp]: "frames_match [] []"
  by (simp add: frames_match_def)
```

An activation frame matches the top CFG activation on caller store and destination.

```
lemma frames_match_activation:
  "frames_match (Frame s d # frs) ((cont, d, s) # stk)
     = frames_match frs stk"
  by (simp add: frames_match_def cframe_act_def)

end
```

4.5 Activation_Trace_Def

```
theory Activation_Trace_Def
  imports CFG_Transfer
begin
```

Activation traces

An `activation_trace` represents one procedure activation of a sequential run and its concrete ancestry. Its own path contains `(cfg_node, store)` pairs; structural constructors record the suspended caller and, after resumption, the completed callee. Activation contexts are projections of this structure rather than fields stored in the trace.

Local flow follows `intra`, including `EA_Ret` into the matching `FunctionResult`. A `calls` tuple supplies the call site, callee entry, and continuation. The call rule builds the parameter-bound entered store with `call_enter`. The resume rule combines caller locals, callee globals, and the return value with `combine_collect`.

The shape is adapted from the local traces of Schwarz et al., *Improving Thread-Modular Abstract Interpretation* (SAS 2021), and Schwarz and Erhard, *Data Race Detection by Digest-Driven Abstract Interpretation* (arXiv:2511.11055, 2025). Theirs is a multithreaded semantics: a local trace is one thread's view, in which operations of different threads are only partially ordered. An activation trace is one activation of a sequential run, and a return composes the completed callee into its suspended caller.

The datatype

- `Root p` — the main activation, with local path `p`.
- `Call caller p` — a callee whose local path `p` starts at the entered store; `caller` is the exact suspended caller, frozen at the call node.
- `Resume current callee p` — the activation continued past a completed call. `current` is that activation frozen at its call node (the value that spawned `callee`); `callee` is the retained completed callee subtree; `p` is the continued path.

An `activation_path` lists the nodes one activation passed and the store it held at each. An activation trace carries the path of its own activation, with the surrounding activations beside it rather than on it.

```
type_synonym activation_path = "(cfg_node × store) list"

datatype activation_trace =
    Root activation_path
  | Call (activation_trace_caller: activation_trace) activation_path
  | Resume (activation_trace_current: activation_trace) (activation_trace_callee: activation_trace)
    activation_path
```

Observers

`path_of` is the activation path. `sink_node` and `sink_store` return its final program point and final store.

```
fun path_of :: "activation_trace ⇒ activation_path" where
  "path_of (Root p)       = p"
| "path_of (Call _ p)     = p"
| "path_of (Resume _ _ p) = p"

definition entry_store :: "activation_trace ⇒ store" where
  "entry_store t = snd (hd (path_of t))"

definition sink_node :: "activation_trace ⇒ cfg_node" where
  "sink_node t = fst (last (path_of t))"

definition sink_store :: "activation_trace ⇒ store" where
  "sink_store t = snd (last (path_of t))"
```

`caller_of` recovers the creating caller of an activation. It descends the frozen `caller` field of a `Resume`, so it works uniformly for a returned callee of any constructor — this is what makes nested and recursive returns compose.

```
fun caller_of :: "activation_trace ⇒ activation_trace option" where
  "caller_of (Root _)             = None"
| "caller_of (Call caller _)      = Some caller"
| "caller_of (Resume current _ _) = caller_of current"
```

Extension and context projection

`extend` appends one step to the innermost local path; it never touches an outer constructor's caller/callee fields.

```
fun extend :: "activation_trace ⇒ (cfg_node * store) ⇒ activation_trace" where
  "extend (Root p) x       = Root (p @ [x])"
| "extend (Call c p) x     = Call c (p @ [x])"
| "extend (Resume c d p) x = Resume c d (p @ [x])"
```

The closure relation

`valid_activation_trace` is the least set closed under four concrete operations: an initial main activation at `cfg_entry`; an `intra` step; a call; and a return. Each rule reads exactly the relation for its phenomenon. `intra` carries no side condition — calls are not `intra` members, so they are untraversable by typing. `call` enters the callee named by the `calls` edge at the entered store `call_enter`. `ret` matches the callee's `FunctionResult p` against the `FunctionEntry p` of a concrete `calls` edge leaving the caller's node, and resumes at the continuation stored in that same edge; the resumed state is `combine_collect`. There is no `combines` lookup and no scan for a compatible context.

```
inductive_set valid_activation_trace ::
    "(vname ⇒ bool) ⇒ cfg ⇒ store set ⇒ activation_trace set"
    ("𝒯_,_,_")
  for 𝒢 and g and S where
  root:
    "s ∈ S
     ⟹ Root [(cfg_entry g, s)] ∈ 𝒯𝒢,g,S"
| intra:
    "t ∈ 𝒯𝒢,g,S
     ⟹ (sink_node t, a, v) ∈ intra g
     ⟹ s' ∈ edge_step a (sink_store t)
     ⟹ extend t (v, s') ∈ 𝒯𝒢,g,S"
| call:
    "caller ∈ 𝒯𝒢,g,S
     ⟹ (sink_node caller, CallEdge dst pars args, FunctionEntry p, cont)
          ∈ calls g
     ⟹ Call caller
          [(FunctionEntry p,
            call_enter 𝒢 (CallEdge dst pars args) (sink_store caller))]
        ∈ 𝒯𝒢,g,S"
| ret:
    "callee ∈ 𝒯𝒢,g,S
     ⟹ caller_of callee = Some caller
     ⟹ sink_node callee = FunctionResult p
     ⟹ (sink_node caller, CallEdge dst pars args, FunctionEntry p, cont)
          ∈ calls g
     ⟹ Resume caller callee
          (path_of caller
             @ [(cont, combine_collect 𝒢 dst
                         (sink_store caller) (sink_store callee))])
        ∈ 𝒯𝒢,g,S"

inductive_cases valid_activation_trace_RootE [elim]:
  "Root p ∈ 𝒯𝒢,g,S"

inductive_cases valid_activation_trace_CallE [elim]:
  "Call caller p ∈ 𝒯𝒢,g,S"

inductive_cases valid_activation_trace_ResumeE [elim]:
  "Resume caller callee p ∈ 𝒯𝒢,g,S"
```

Structural lemmas

How the three trace shapes answer the projections every later proof reads them through – path, sink, caller, entry store. Kept as `simp` rules so an induction over `valid_activation_trace` never has to case on the constructor merely to look up a sink.

```
lemma extend_simps [simp]:
  "path_of (extend t x) = path_of t @ [x]"
  "caller_of (extend t x) = caller_of t"
  "sink_node (extend t x) = fst x"
  "sink_store (extend t x) = snd x"
  by (cases t; auto simp: sink_node_def sink_store_def)+

lemma sink_single_simps [simp]:
```

```
  "sink_node (Root [(n, s)]) = n"
  "sink_store (Root [(n, s)]) = s"
  "sink_node (Call c [(n, s)]) = n"
  "sink_store (Call c [(n, s)]) = s"
  by (simp_all add: sink_node_def sink_store_def)
```

The lemmas up to the caller-chain closure all speak about the traces of one graph from one set of initial stores, so the block fixes $\mathcal{G}$, `g` and `S` and writes $\mathcal{T}$ for $\mathcal{T}_{\mathcal{G},\mathtt{g},\mathtt{S}}$. Outside the block every lemma holds for arbitrary $\mathcal{G}$, `g` and `S`.

```
context
  fixes 𝒢 :: "vname ⇒ bool" and g :: cfg and S :: "store set"
begin

private abbreviation (input) traces :: "activation_trace set" where "traces ≡ 𝒯⟨𝒢,g,S⟩"
notation traces ("𝒯")

lemma valid_activation_trace_path_nonempty:
  "t ∈ 𝒯 ⟹ path_of t ≠ []"
  by (induction t rule: valid_activation_trace.induct) auto

lemma entry_store_extend [simp]:
  assumes "path_of t ≠ []"
  shows "entry_store (extend t x) = entry_store t"
  using assms by (simp add: entry_store_def)
```

Design invariants

A valid `activation_trace.Call` activation has a valid caller, even after intra steps have extended its local path.

```
lemma valid_activation_trace_Call_caller_valid:
  "u ∈ 𝒯 ⟹ u = Call cc q ⟹ cc ∈ 𝒯"
proof (induction arbitrary: cc q rule: valid_activation_trace.induct)
  case (intra t a v s')
  from intra.prems obtain q' where "t = Call cc q'"
    by (cases t) auto
  then show ?case using intra.IH by simp
qed auto
```

A valid `Resume` retains its callee as a valid activation trace, and its frozen caller is forced to be exactly `caller_of callee` — a return cannot invent a caller.

```
lemma valid_activation_trace_Resume_fields:
  "u ∈ 𝒯 ⟹ u = Resume cc dd q
   ⟹ dd ∈ 𝒯 ∧ caller_of dd = Some cc"
proof (induction arbitrary: cc dd q rule: valid_activation_trace.induct)
  case (intra t a v s')
  from intra.prems obtain q' where "t = Resume cc dd q'"
    by (cases t) auto
  then show ?case using intra.IH by simp
qed auto
```

Every caller recovered from a valid activation trace by `caller_of` is itself valid.

```
lemma valid_activation_trace_caller_valid:
  "t ∈ 𝒯 ⟹ caller_of t = Some c ⟹ c ∈ 𝒯"
proof (induction t arbitrary: c)
  case (Root x)
  then show ?case by simp
next
  case (Call caller p)
  have "caller ∈ 𝒯"
    using valid_activation_trace_Call_caller_valid[OF Call.prems(1) refl] .
  with Call.prems(2) show ?case by simp
next
  case (Resume caller callee p)
  from valid_activation_trace_Resume_fields[OF Resume.prems(1) refl]
  have cd: "callee ∈ 𝒯" "caller_of callee = Some caller" by auto
  have cv: "caller ∈ 𝒯" using Resume.IH(2)[OF cd(1)] cd(2) by simp
  from Resume.prems(2) have "caller_of caller = Some c" by simp
  then show ?case using Resume.IH(1)[OF cv] by simp
qed
```

A `Root` activation starts at `cfg_entry`: `root` creates it there and `intra` only appends.

```
lemma valid_activation_trace_Root_entry:
  "u ∈ 𝒯 ⟹ u = Root p ⟹ fst (hd p) = cfg_entry g"
```

```
proof (induction arbitrary: p rule: valid_activation_trace.induct)
  case (intra t a v s')
  from intra.prems obtain p' where t: "t = Root p'" and p: "p = p' @ [(v, s')]"
    by (cases t) auto
  have "fst (hd p') = cfg_entry g" using intra.IH[OF t] .
  moreover have "p' ≠ []" using valid_activation_trace_path_nonempty[OF intra.hyps(1)] t by simp
  ultimately show ?case using p by simp
qed auto
```

A `Resume` extends its caller's own local path, so both share a head node.

```
lemma valid_activation_trace_Resume_path:
  "u ∈ 𝒯 ⟹ u = Resume cc dd q ⟹ ∃xs. q = path_of cc @ xs ∧ xs ≠ []"
proof (induction arbitrary: cc dd q rule: valid_activation_trace.induct)
  case (intra t a v s')
  from intra.prems obtain q' where t: "t = Resume cc dd q'" and q: "q = q' @ [(v, s')]"
    by (cases t) auto
  from intra.IH[OF t] obtain xs where "q' = path_of cc @ xs" by blast
  then show ?case using q by auto
qed auto
```

A callerless activation is the root one, and its local `path_of` starts at `cfg_entry`. The `Resume` case needs the caller's own entry, which term induction supplies (the caller is a subterm) — rule induction would only offer the callee.

```
lemma valid_activation_trace_caller_None_entry:
  "t ∈ 𝒯 ⟹ caller_of t = None ⟹ fst (hd (path_of t)) = cfg_entry g"
proof (induction t)
  case (Root p)
  then show ?case using valid_activation_trace_Root_entry[OF Root.prems(1) refl] by simp
next
  case (Call caller p)
  then show ?case by simp
next
  case (Resume caller callee q)
  from valid_activation_trace_Resume_fields[OF Resume.prems(1) refl]
  have cd: "callee ∈ 𝒯" "caller_of callee = Some caller" by auto
  have cv: "caller ∈ 𝒯" using Resume.IH(2)[OF cd(1)] cd(2)
    using valid_activation_trace_caller_valid[OF cd(1) cd(2)] by simp
  from Resume.prems(2) have "caller_of caller = None" by simp
  with Resume.IH(1)[OF cv] have hcaller: "fst (hd (path_of caller)) = cfg_entry g" by simp
  from valid_activation_trace_Resume_path[OF Resume.prems(1) refl] obtain xs where
    q: "q = path_of caller @ xs" by blast
  show ?case using hcaller q valid_activation_trace_path_nonempty[OF cv] by simp
qed
```

Caller ancestry

`ancestors t` is the `caller_of` chain above `t`; `callers t` adds `t` itself and is the set a caller-chain invariant ranges over.

```
fun ancestors :: "activation_trace ⇒ activation_trace set" where
  "ancestors (Root _) = {}"
| "ancestors (Call caller _) = insert caller (ancestors caller)"
| "ancestors (Resume current _ _) = ancestors current"

abbreviation callers :: "activation_trace ⇒ activation_trace set" where
  "callers t ≡ insert t (ancestors t)"

lemma ancestors_extend [simp]: "ancestors (extend t x) = ancestors t"
  by (cases t) simp_all

lemma callers_refl: "t ∈ callers t"
  by simp

lemma ancestors_caller: "caller_of t = Some c ⟹ callers c ⊆ ancestors t"
  by (induction t) auto

lemma callers_caller_subset: "caller_of t = Some c ⟹ callers c ⊆ callers t"
  using ancestors_caller by blast
```

Generic caller-chain closure

A caller-chain-quantified predicate `P` holds along the whole `callers` chain of every valid activation trace once its four `valid_activation_trace` obligations are discharged. Each obligation reads its own generating trace's

induction hypothesis as the whole-chain fact ∀u ∈ callers _. P u, not a single-node fact — this is exactly what ret needs to recover its caller's own chain fact from the callee's.

```
lemma caller_chain_closure:
  fixes P :: "activation_trace ⇒ bool"
  assumes Root: "⋀s. s ∈ S ⟹ P (Root [(cfg_entry g, s)])"
    and Intra: "⋀t a v s'. t ∈ 𝒯 ⟹ (∀u ∈ callers t. P u)
        ⟹ (sink_node t, a, v) ∈ intra g ⟹ s' ∈ edge_step a (sink_store t)
        ⟹ P (extend t (v, s'))"
    and Call: "⋀caller dst pars args p cont. caller ∈ 𝒯
        ⟹ (∀u ∈ callers caller. P u)
        ⟹ (sink_node caller, CallEdge dst pars args, FunctionEntry p, cont)
           ∈ calls g
        ⟹ P (Call caller
                  [(FunctionEntry p, call_enter 𝒢 (CallEdge dst pars args) (sink_store caller))])"
    and Ret: "⋀callee caller p dst pars args cont. callee ∈ 𝒯
        ⟹ (∀u ∈ callers callee. P u)
        ⟹ caller_of callee = Some caller ⟹ sink_node callee = FunctionResult p
        ⟹ (sink_node caller, CallEdge dst pars args, FunctionEntry p, cont)
           ∈ calls g
        ⟹ P (Resume caller callee (path_of caller
                  @ [(cont, combine_collect 𝒢 dst (sink_store caller) (sink_store callee))]))"
  shows "t ∈ 𝒯 ⟹ ∀u ∈ callers t. P u"
proof (induction rule: valid_activation_trace.induct)
  case (root s)
  then show ?case using Root[OF root] by simp
next
  case (intra t a v s')
  then show ?case using Intra[OF intra.hyps(1) intra.IH intra.hyps(2,3)] by auto
next
  case (call caller dst pars args p cont)
  then show ?case using Call[OF call.hyps(1) call.IH call.hyps(2)] by auto
next
  case (ret callee caller p dst pars args cont)
  then show ?case
    using Ret[OF ret.hyps(1) ret.IH ret.hyps(2,3,4)] ancestors_caller[OF ret.hyps(2)] by auto
qed

end
```

Stable context entry invariant

An activation's entry store is fixed once it starts: a call names the store the callee begins with, and resuming a caller keeps the caller's own first store rather than adopting the callee's. Context policies that read the entry store rely on this – otherwise the context a trace carries could change under it as the trace grows.

```
definition call_enter_store :: "(vname ⇒ bool) ⇒ cfg ⇒ cfg_node ⇒ store ⇒ store ⇒ bool" where
  "call_enter_store 𝒢 g c s t ⟷
     (∃dst pars args p cont. (c, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
        ∧ t = call_enter 𝒢 (CallEdge dst pars args) s)"

lemma entry_store_Resume_caller:
  "path_of caller ≠ [] ⟹
     entry_store (Resume caller callee (path_of caller @ [x])) = entry_store caller"
  by (simp add: entry_store_def hd_append)

end
```

4.6 Activation_Trace_Context

```
theory Activation_Trace_Context
  imports Activation_Trace_Def
begin
```

Activation context selection and context-indexed collecting

The activation context is computed from the concrete trace after the fact and is activation-stable: fixed when the activation is created and unchanged by the calls it later makes and returns from. `Root` carries the seed; a `activation_trace.Call` routes the caller context on the call site and the entered store — `sink_node parent` is exactly the call site, since `Activation_Trace_Def.extend` only ever appends to the callee path and leaves `parent` frozen there; a `Resume` keeps the resumed caller's own context, so a completed call does not repartition the caller. Exposing the call site to `enterc` is what makes a call-site-keyed context (a k-call-string) expressible as an instance rather than only as a generator-side hook.

```
fun activation_context_of :: "(cfg_node ⇒ 'c ⇒ store ⇒ 'c) ⇒ 'c ⇒ activation_trace ⇒ 'c" where
  "activation_context_of enterc initial_ctx (Root _)                  = initial_ctx"
| "activation_context_of enterc initial_ctx (Call parent p)           =
     enterc (sink_node parent) (activation_context_of enterc initial_ctx parent) (snd (hd p))"
| "activation_context_of enterc initial_ctx (Resume current callee _) = activation_context_of enterc initial_ctx current"
```

`activation_context_of` fixes one context per trace by decoding the entered store exactly: a `activation_trace.Call` activation's context is derived from its caller's OWN selected context, not re-derived from the concrete store independently at each node. This is what keeps a chosen context stable across a CALL/RETURN pair: the callee's context in `Resume` resumes through `current`, so a completed call cannot repartition the caller.

`activation_context_of` is the deterministic special case: a total function from the entered store to a context, never a relation admitting several contexts for one concrete activation. The unit and call-site/k-call-string routing instances are exactly this shape, and the executable routed solver's own `route` is typed to return a single context, not a set. The entry-state instance is not: a call may answer several overlapping alternatives, each covering the same concrete activation and routing it to its own context, so its concrete semantics is stated directly over `activation_context_rel`/`context_policy` rather than over `activation_context_of`. `activation_context_of` is therefore not the primary semantics; it is the deterministic compatibility witness for the functional case, connected to `activation_context_rel` via `context_policy_of_fun` and `activation_context_rel_of_fun_iff`.

```
lemma activation_context_extend_nonempty:
  "path_of t ≠ [] ⟹ activation_context_of enterc initial_ctx (extend t x) = activation_context_of enterc initial_ctx
t"
  by (cases t) (auto simp: hd_append)
```

Admissible callee contexts, as a set per call

An analysis's entry operation answers a *list* of alternatives and its context selector is applied to each, so one concrete call may be admitted in several contexts. `context_policy` is the semantic shape of that whole-call protocol: given the call site, the caller's own context, the call's information, the caller store and the entered store, it returns the set of callee contexts that may describe this transition. The set may be empty or have several elements; `call_context_total_on` below requires it to be non-empty for every covered call. Both stores are arguments so an instance can tie the chosen context to one alternative whose continuation half covers the caller and whose entry half covers the entered store, never mixing two.

A functional policy — the unit context, a k-call-string — returns a singleton, `context_policy_of_fun`. The abstract selector an analysis actually runs stays a function of the abstract entry state; only the activation-trace semantics is set-valued.

```
type_synonym 'c context_policy =
  "cfg_node ⇒ 'c ⇒ call_info ⇒ store ⇒ store ⇒ 'c set"

definition context_policy_of_fun ::
  "(cfg_node ⇒ 'c ⇒ store ⇒ 'c) ⇒ 'c context_policy"
where
  "context_policy_of_fun f u ctx ci caller entered = {f u ctx entered}"

lemma context_policy_of_fun_iff [simp]:
  "ctx' ∈ context_policy_of_fun f u ctx ci caller entered ⟷ ctx' = f u ctx entered"
  by (simp add: context_policy_of_fun_def)

lemma context_policy_of_fun_nonempty [simp]:
  "context_policy_of_fun f u ctx ci caller entered ≠ {}"
  by (simp add: context_policy_of_fun_def)
```

`admits_call_context 𝒢 g adm u ctx p s es ctx'`: from node `u` in context `ctx` with store `s`, some `calls` edge of `g` enters `p` with exactly the store `es`, and `adm` admits `ctx'` for that edge. It is `call_enter_store` with the callee and the admitted context made explicit. Naming the edge by its effect rather than assuming it unique keeps the semantics honest on a graph with two call edges leaving one node, exactly as `valid_activation_trace.ret` does.

```
definition admits_call_context ::
  "(vname ⇒ bool) ⇒ cfg ⇒ 'c context_policy
     ⇒ cfg_node ⇒ 'c ⇒ pname ⇒ store ⇒ store ⇒ 'c ⇒ bool"
where
  "admits_call_context 𝒢 g adm u ctx p s es ctx' ⟷
     (∃dst pars args cont.
        (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
        ∧ es = call_enter 𝒢 (CallEdge dst pars args) s
        ∧ ctx' ∈ adm u ctx (call_info_of (CallEdge dst pars args) p) s es)"

lemma admits_call_contextI:
  assumes "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and "ctx' ∈ adm u ctx (call_info_of (CallEdge dst pars args) p) s
           (call_enter 𝒢 (CallEdge dst pars args) s)"
  shows "admits_call_context 𝒢 g adm u ctx p s (call_enter 𝒢 (CallEdge dst pars args) s) ctx'"
  using assms unfolding admits_call_context_def by blast

lemma admits_call_contextE:
  assumes "admits_call_context 𝒢 g adm u ctx p s es ctx'"
  obtains dst pars args cont
    where "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
      and "es = call_enter 𝒢 (CallEdge dst pars args) s"
      and "ctx' ∈ adm u ctx (call_info_of (CallEdge dst pars args) p) s es"
  using assms unfolding admits_call_context_def by blast

lemma admits_call_context_call_enter_storeD:
  "admits_call_context 𝒢 g adm u ctx p s es ctx' ⟹ call_enter_store 𝒢 g u s es"
  unfolding admits_call_context_def call_enter_store_def by blast

lemma admits_call_context_of_fun:
  "admits_call_context 𝒢 g (context_policy_of_fun f) u ctx p s es ctx'
     ⟷ (∃dst pars args cont.
           (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
           ∧ es = call_enter 𝒢 (CallEdge dst pars args) s)
       ∧ ctx' = f u ctx es"
  unfolding admits_call_context_def by auto
```

`activation_context_rel 𝒢 adm c₀ g t ctx`: the trace `t` may carry context `ctx`. A root carries `c₀`; a call carries any context admitted for its creating transition, computed from the caller's own carried context; a resumed caller keeps what it carried.

```
inductive activation_context_rel ::
  "(vname ⇒ bool) ⇒ 'c context_policy ⇒ 'c ⇒ cfg ⇒ activation_trace ⇒ 'c ⇒ bool"
  for 𝒢 :: "vname ⇒ bool" and adm :: "'c context_policy" and c₀ :: 'c and g :: cfg
where
  Root [intro]: "activation_context_rel 𝒢 adm c₀ g (Root xs) c₀"
| Call [intro]:
    "activation_context_rel 𝒢 adm c₀ g caller c
     ⟹ admits_call_context 𝒢 g adm (sink_node caller) c p (sink_store caller) es c'
     ⟹ activation_context_rel 𝒢 adm c₀ g (Call caller ((FunctionEntry p, es) # xs)) c'"
| Resume [intro]:
    "activation_context_rel 𝒢 adm c₀ g current c
     ⟹ activation_context_rel 𝒢 adm c₀ g (Resume current callee xs) c"


inductive_cases activation_context_rel_RootE [elim!]:
  "activation_context_rel 𝒢 adm c₀ g (Root xs) ctx"

inductive_cases activation_context_rel_CallE [elim]:
  "activation_context_rel 𝒢 adm c₀ g (Call parent xs) ctx"

inductive_cases activation_context_rel_ResumeE [elim]:
  "activation_context_rel 𝒢 adm c₀ g (Resume current callee xs) ctx"

lemma activation_context_rel_Root_iff [simp]:
  "activation_context_rel 𝒢 adm c₀ g (Root xs) ctx ⟷ ctx = c₀"
  by blast

lemma activation_context_rel_Resume_iff [simp]:
  "activation_context_rel 𝒢 adm c₀ g (Resume current callee xs) ctx
```

```
    ⟷ activation_context_rel 𝒢 adm c₀ g current ctx"
  by blast

lemma activation_context_rel_Call_Nil [simp]:
  "¬ activation_context_rel 𝒢 adm c₀ g (Call parent []) ctx"
  by blast
```

The carried context of a called activation depends on its caller and its entry record only: the rest of the local path is irrelevant, which is what makes an intra step context-preserving.

```
lemma activation_context_rel_Call_iff:
  "activation_context_rel 𝒢 adm c₀ g (Call parent ((n, es) # xs)) ctx'
    ⟷ (∃ctx p. n = FunctionEntry p
         ∧ activation_context_rel 𝒢 adm c₀ g parent ctx
         ∧ admits_call_context 𝒢 g adm (sink_node parent) ctx p (sink_store parent) es ctx')"
  by blast

lemma activation_context_rel_extend:
  assumes "path_of t ≠ []"
  shows "activation_context_rel 𝒢 adm c₀ g (extend t x) ctx
         ⟷ activation_context_rel 𝒢 adm c₀ g t ctx"
proof (cases t)
  case (Call parent xs)
  then obtain n es ys where "xs = (n, es) # ys" using assms by (cases xs) auto
  then show ?thesis using Call by (simp add: activation_context_rel_Call_iff)
qed simp_all
```

Context entry invariant

The invariants below concern the traces of one graph from one set of initial stores, so the block fixes $\mathcal{G}$, g and S and writes $\mathcal{T}$ for $\mathcal{T}_{\mathcal{G},g,S}$.

```
context
  fixes 𝒢 :: "vname ⇒ bool" and g :: cfg and S :: "store set"
begin

private abbreviation (input) traces :: "activation_trace set" where "traces ≡ 𝒯𝒢,g,S"
notation traces ("𝒯")
```

A called activation carries exactly the contexts admitted for the transition from its immediate caller, each computed from a context the caller itself carries. Stated as an equivalence: forwards, a RETURN argument recovers the caller's slot from the callee's; backwards, it re-enters the callee at a context derived from whichever slot the resumed caller is being placed in.

```
lemma activation_context_rel_caller_entry:
  assumes "callee ∈ 𝒯"
  shows "∀u ∈ callers callee. ∀c. caller_of u = Some c
      ⟶ (∃p es. hd (path_of u) = (FunctionEntry p, es)
           ∧ (∀ctx'. activation_context_rel 𝒢 adm c₀ g u ctx'
                ⟷ (∃ctx. activation_context_rel 𝒢 adm c₀ g c ctx
                     ∧ admits_call_context 𝒢 g adm (sink_node c) ctx p (sink_store c) es ctx')))"
proof -
  define P where "P u ⟷ (∀c. caller_of u = Some c
      ⟶ (∃p es. hd (path_of u) = (FunctionEntry p, es)
           ∧ (∀ctx'. activation_context_rel 𝒢 adm c₀ g u ctx'
                ⟷ (∃ctx. activation_context_rel 𝒢 adm c₀ g c ctx
                     ∧ admits_call_context 𝒢 g adm (sink_node c) ctx p (sink_store c) es ctx'))))"
    for u
  have "∀u ∈ callers callee. P u"
  proof (rule caller_chain_closure[OF _ _ _ _ assms])
    fix s show "P (Root [(cfg_entry g, s)])" by (simp add: P_def)
  next
    fix t a v s' assume t: "t ∈ 𝒯" and IH: "∀u ∈ callers t. P u"
    have "path_of t ≠ []" using valid_activation_trace_path_nonempty[OF t] .
    with IH[THEN bspec, OF callers_refl] show "P (extend t (v, s'))"
      by (simp add: P_def activation_context_rel_extend hd_append)
  next
    fix caller dst pars args p cont
    show "P (Call caller
        [(FunctionEntry p, call_enter 𝒢 (CallEdge dst pars args) (sink_store caller))])"
      unfolding P_def by (auto simp: activation_context_rel_Call_iff)
  next
    fix callee caller p dst pars args cont
    assume cv: "callee ∈ 𝒯" and IH: "∀u ∈ callers callee. P u"
      and cof: "caller_of callee = Some caller"
    have "path_of caller ≠ []"
```

```
    using valid_activation_trace_path_nonempty[OF valid_activation_trace_caller_valid[OF cv cof]]
    .
  moreover have "P caller" using IH ancestors_caller[OF cof] by blast
  ultimately show "P (Resume caller callee
      (path_of caller @ [(cont, combine_collect 𝒢 dst (sink_store caller) (sink_store callee))]))"
    by (simp add: P_def hd_append)
 qed
 then show ?thesis unfolding P_def .
qed

lemma activation_context_rel_caller_entryE:
 assumes "callee ∈ 𝒯" and "caller_of callee = Some caller"
   and "activation_context_rel 𝒢 adm c₀ g callee ctx'"
 obtains ctx p where "hd (path_of callee) = (FunctionEntry p, entry_store callee)"
   and "activation_context_rel 𝒢 adm c₀ g caller ctx"
   and "admits_call_context 𝒢 g adm (sink_node caller) ctx p (sink_store caller)
        (entry_store callee) ctx'"
proof -
 from activation_context_rel_caller_entry[OF assms(1), THEN bspec, OF callers_refl,
     rule_format, OF assms(2)]
 obtain p es where hd: "hd (path_of callee) = (FunctionEntry p, es)"
   and iff: "∀ctx'. activation_context_rel 𝒢 adm c₀ g callee ctx'
       ⟷ (∃ctx. activation_context_rel 𝒢 adm c₀ g caller ctx
            ∧ admits_call_context 𝒢 g adm (sink_node caller) ctx p (sink_store caller) es ctx')"
   by force
 have "es = entry_store callee" using hd by (simp add: entry_store_def)
 with hd iff assms(3) show thesis using that by blast
qed

lemma activation_context_rel_caller_entryI:
 assumes "callee ∈ 𝒯" and "caller_of callee = Some caller"
   and "hd (path_of callee) = (FunctionEntry p, entry_store callee)"
   and "activation_context_rel 𝒢 adm c₀ g caller ctx"
   and "admits_call_context 𝒢 g adm (sink_node caller) ctx p (sink_store caller)
        (entry_store callee) ctx'"
 shows "activation_context_rel 𝒢 adm c₀ g callee ctx'"
proof -
 from activation_context_rel_caller_entry[OF assms(1), THEN bspec, OF callers_refl,
     rule_format, OF assms(2)]
 obtain p' es where hd: "hd (path_of callee) = (FunctionEntry p', es)"
   and iff: "∀ctx'. activation_context_rel 𝒢 adm c₀ g callee ctx'
       ⟷ (∃ctx. activation_context_rel 𝒢 adm c₀ g caller ctx
            ∧ admits_call_context 𝒢 g adm (sink_node caller) ctx p' (sink_store caller) es ctx')"
   by force
 have "p' = p" and "es = entry_store callee" using hd assms(3) by simp_all
 with iff assms(4,5) show ?thesis by blast
qed
```

A functional policy carries exactly `activation_context_of`'s context on every valid activation trace. The restriction to valid activation traces is what supplies the call edge the relational clause names.

```
lemma activation_context_rel_of_fun_iff:
 assumes "t ∈ 𝒯"
 shows "activation_context_rel 𝒢 (context_policy_of_fun f) c₀ g t ctx
        ⟷ activation_context_of f c₀ t = ctx"
proof -
 define P where "P u ⟷ (∀ctx. activation_context_rel 𝒢 (context_policy_of_fun f) c₀ g u ctx
                          ⟷ activation_context_of f c₀ u = ctx)" for u
 have "∀u ∈ callers t. P u"
 proof (rule caller_chain_closure[OF _ _ _ _ assms])
   fix s show "P (Root [(cfg_entry g, s)])" by (simp add: P_def eq_commute)
 next
   fix t a v s' assume t: "t ∈ 𝒯" and IH: "∀u ∈ callers t. P u"
   have "path_of t ≠ []" using valid_activation_trace_path_nonempty[OF t] .
   with IH[THEN bspec, OF callers_refl] show "P (extend t (v, s'))"
     by (simp add: P_def activation_context_extend_nonempty activation_context_rel_extend)
 next
   fix caller dst pars args p cont
   assume IH: "∀u ∈ callers caller. P u"
     and e: "(sink_node caller, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
   have Pc: "P caller" using IH callers_refl by blast
   show "P (Call caller
       [(FunctionEntry p, call_enter 𝒢 (CallEdge dst pars args) (sink_store caller))])"
     unfolding P_def activation_context_rel_Call_iff admits_call_context_of_fun
     using Pc[unfolded P_def] e by auto
 next
   fix callee caller p dst pars args cont
```

```
    assume IH: "∀u ∈ callers callee. P u" and cof: "caller_of callee = Some caller"
    have "P caller" using IH ancestors_caller[OF cof] by blast
    then show "P (Resume caller callee
        (path_of caller @ [(cont, combine_collect 𝒢 dst (sink_store caller) (sink_store callee))]))"
      by (simp add: P_def)
  qed
  then show ?thesis unfolding P_def using callers_refl by blast
qed
```

Functional context entry invariant

Every ancestor of a valid activation trace is itself valid: `caller_chain_closure`'s own four cases are exactly `valid_activation_trace`'s four introduction rules, so the closure fact and validity coincide.

```
lemma callers_valid:
  assumes "callee ∈ 𝒯" and "u ∈ callers callee"
  shows "u ∈ 𝒯"
proof -
  have "∀u ∈ callers callee. u ∈ 𝒯"
  proof (rule caller_chain_closure[OF _ _ _ _ assms(1)], goal_cases Root Intra Call Ret)
    case (Root s) then show ?case by (rule valid_activation_trace.root)
  next
    case (Intra t a v s') then show ?case by (blast intro: valid_activation_trace.intra)
  next
    case (Call caller dst pars args p cont) then show ?case
      by (blast intro: valid_activation_trace.call)
  next
    case (Ret callee caller p dst pars args cont) then show ?case
      by (blast intro: valid_activation_trace.ret)
  qed
  then show ?thesis using assms(2) by blast
qed
```

The context of an activation is exactly what `enterc` computes from its immediate caller's own context and the entered store. This is what lets a RETURN argument construct the callee's context directly from the caller's chosen one, rather than rediscovering it after the fact. A special case of `activation_context_rel_caller_entry` at the functional policy `context_policy_of_fun`, where `activation_context_rel_of_fun_iff` pins both sides' witness context to `activation_context_of` and `admits_call_context_of_fun` reads off the `enterc`-equation and the call-edge witness in one step.

```
lemma activation_context_entry_invariant:
  assumes "callee ∈ 𝒯"
  shows "∀u ∈ callers callee. ∀c. caller_of u = Some c ⟶
      activation_context_of enterc initial_ctx u = enterc (sink_node c) (activation_context_of enterc initial_ctx
c) (entry_store u)
      ∧ call_enter_store 𝒢 g (sink_node c) (sink_store c) (entry_store u)"
proof (intro ballI allI impI)
  fix u c assume mem: "u ∈ callers callee" and cof: "caller_of u = Some c"
  have uv: "u ∈ 𝒯" using callers_valid[OF assms mem] .
  have cv: "c ∈ 𝒯" using valid_activation_trace_caller_valid[OF uv cof] .
  have tc_u: "activation_context_rel 𝒢 (context_policy_of_fun enterc) initial_ctx g u
                (activation_context_of enterc initial_ctx u)"
    by (simp only: activation_context_rel_of_fun_iff[OF uv])
  obtain p ctx where "hd (path_of u) = (FunctionEntry p, entry_store u)"
    and tc_c: "activation_context_rel 𝒢 (context_policy_of_fun enterc) initial_ctx g c ctx"
    and adm: "admits_call_context 𝒢 g (context_policy_of_fun enterc) (sink_node c) ctx p
                (sink_store c) (entry_store u) (activation_context_of enterc initial_ctx u)"
    by (rule activation_context_rel_caller_entryE[OF uv cof tc_u])
  have activation_context_c: "activation_context_of enterc initial_ctx c = ctx"
    using tc_c by (simp only: activation_context_rel_of_fun_iff[OF cv])
  have adm': "activation_context_of enterc initial_ctx u = enterc (sink_node c) ctx (entry_store u)"
    using adm unfolding admits_call_context_of_fun by simp
  have activation_context_eq: "activation_context_of enterc initial_ctx u
                 = enterc (sink_node c) (activation_context_of enterc initial_ctx c) (entry_store u)"
    unfolding activation_context_c using adm' .
  show "activation_context_of enterc initial_ctx u
          = enterc (sink_node c) (activation_context_of enterc initial_ctx c) (entry_store u)
        ∧ call_enter_store 𝒢 g (sink_node c) (sink_store c) (entry_store u)"
    using activation_context_eq admits_call_context_call_enter_storeD[OF adm] by (intro conjI)
qed

lemma key_entry_invariant_call_enterD [dest]:
  assumes "callee ∈ 𝒯" and "caller_of callee = Some caller"
  shows "call_enter_store 𝒢 g (sink_node caller) (sink_store caller) (entry_store callee)"
  using activation_context_entry_invariant[OF assms(1), THEN bspec, OF callers_refl, rule_format, OF
```

```
    assms(2)]
  by (rule conjunct2)

end
```

Conditional totality

A policy may admit no context for a call. The callee activation then carries no context and drops out of every bucket, but the resumed caller keeps its own, so the continuation is still collected. CALL and RETURN hold vacuously at such a call, and a claim may leave the continuation uncovered although a valid activation trace reaches it there (`total_dropped_unsound` in `Example_Non_Vacuity`). `call_context_total_on cover adm` $\mathcal{G}$ `g` rules that out exactly where it matters: at every call edge, every store the claimed `cover` admits at the call site has some admissible callee context. A dead call site owes nothing. The graph of a function is total outright.

```
definition call_context_total_on ::
  "(cfg_node ⇒ 'c ⇒ store set) ⇒ 'c context_policy ⇒ (vname ⇒ bool) ⇒ cfg ⇒ bool"
where
  "call_context_total_on cover adm 𝒢 g ⟷
     (∀u dst pars args p cont ctx s.
        (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
        ⟶ s ∈ cover u ctx
        ⟶ adm u ctx (call_info_of (CallEdge dst pars args) p) s
              (call_enter 𝒢 (CallEdge dst pars args) s) ≠ {})"

lemma call_context_total_onE:
  assumes "call_context_total_on cover adm 𝒢 g"
    and "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and "s ∈ cover u ctx"
  obtains ctx' where "admits_call_context 𝒢 g adm u ctx p s
                        (call_enter 𝒢 (CallEdge dst pars args) s) ctx'"
  using assms unfolding call_context_total_on_def
  by (blast intro: admits_call_contextI)

lemma call_context_total_on_of_fun [simp]:
  "call_context_total_on cover (context_policy_of_fun f) 𝒢 g"
  unfolding call_context_total_on_def context_policy_of_fun_def by blast
```

The activation-indexed context collecting

The activation-sensitive collecting is the sink stores of valid activation traces reaching `v` that may carry the queried context `c`. One trace may carry several contexts, so the buckets of one node cover its stores without partitioning them.

```
definition activation_collect ::
  "(vname ⇒ bool) ⇒ 'c context_policy ⇒ 'c ⇒ cfg
     ⇒ store set ⇒ cfg_node ⇒ 'c ⇒ store set"
    ("𝒜_,_,_,_,_") where
  "𝒜𝒢,adm,c₀,g,S v c =
     {sink_store t | t. t ∈ 𝒯𝒢,g,S ∧ sink_node t = v
                        ∧ activation_context_rel 𝒢 adm c₀ g t c}"

lemma activation_collect_I [intro]:
  "t ∈ 𝒯𝒢,g,S ⟹ sink_node t = v ⟹ activation_context_rel 𝒢 adm c₀ g t c
   ⟹ sink_store t ∈ 𝒜𝒢,adm,c₀,g,S v c"
  unfolding activation_collect_def by blast
```

Every collected state has a valid activation trace witness carrying the queried `c`.

```
lemma activation_collect_E [elim]:
  assumes "s ∈ 𝒜𝒢,adm,c₀,g,S v c"
  obtains t where "t ∈ 𝒯𝒢,g,S" "sink_node t = v"
    "activation_context_rel 𝒢 adm c₀ g t c" "sink_store t = s"
  using assms unfolding activation_collect_def by blast
```

Under a functional policy the buckets are `activation_context_of`'s fibres.

```
lemma activation_collect_of_fun:
  "𝒜𝒢,context_policy_of_fun f,c₀,g,S v c =
     {sink_store t | t. t ∈ 𝒯𝒢,g,S ∧ sink_node t = v ∧ activation_context_of f c₀ t = c}"
  unfolding activation_collect_def
  by (auto simp: activation_context_rel_of_fun_iff)


end
```

4.7 Activation_Trace_Collect

```
theory Activation_Trace_Collect
  imports Activation_Trace_Context
begin
```

Which stores can occur at each node

$\mathcal{C}_{\mathcal{G},g,S}$ v is the set of stores a run can actually have at node v: take every valid activation trace that ends there and keep its final store. This is the concrete set an analysis has to over-approximate, so it is the target of every soundness statement downstream.

valid_activation_trace has four clauses — root, intra step, call, return — each reading exactly the relation for its phenomenon: intra for local extension, calls for entering a callee and for recovering a continuation. The return clause recovers the caller from the completed callee's own ancestry through caller_of, never by an independent choice.

There is no global exit node: whole-program completion is collection at FunctionResult main, and a procedure result is an ordinary collected node rather than a separate summary mechanism.

Forgetful projections

node_collect is the concrete collecting view: the sink stores of valid activation traces reaching node v. activation_collect is the context-indexed collector, keeping those whose trace may carry the queried context; the context type is not required finite, and a bucket is not claimed to be an exact activation identity.

```
definition node_collect ::
    "(vname ⇒ bool) ⇒ cfg ⇒ store set ⇒ cfg_node ⇒ store set"
    ("C_,_,_") where
  "C_,g,S v =
     {sink_store t | t. t ∈ T_,g,S ∧ sink_node t = v}"
```

Up to the inhabitation witness, every lemma concerns one graph and one set of initial stores, so the block fixes $\mathcal{G}$, g and S and writes $\mathcal{T}$ and $\mathcal{C}$ v for $\mathcal{T}_{\mathcal{G},g,S}$ and $\mathcal{C}_{\mathcal{G},g,S}$ v.

```
context
  fixes G :: "vname ⇒ bool" and g :: cfg and S :: "store set"
begin

private abbreviation (input) traces :: "activation_trace set" where "traces ≡ T_,g,S"
private abbreviation (input) collect :: "cfg_node ⇒ store set"
  where "collect ≡ C_,g,S"
notation traces ("T") and collect ("C")

lemma node_collect_I [intro]:
  "t ∈ T ⟹ sink_store t ∈ C (sink_node t)"
  unfolding node_collect_def by blast
```

Every collected state has a valid activation trace witness.

```
lemma node_collect_E [elim]:
  assumes "s ∈ C v"
  obtains t where "t ∈ T" "sink_node t = v" "sink_store t = s"
  using assms unfolding node_collect_def by blast
```

node_collect is closed under intra flow: extending a witness trace by one intra edge is valid_activation_trace.intra. Closure under *call* and *return* steps is deliberately absent here — valid_activation_trace.ret requires the popped activation to be the witness trace's own caller, which a bare CFG step does not supply.

```
lemma node_collect_intra_step:
  assumes s: "s ∈ C u"
    and e: "(u, a, v) ∈ intra g"
    and st: "s' ∈ edge_step a s"
  shows "s' ∈ C v"
proof -
  from s obtain t where t: "t ∈ T" and n: "sink_node t = u" and w: "sink_store t = s"
    by (rule node_collect_E)
  have "extend t (v, s') ∈ T"
    by (rule valid_activation_trace.intra[OF t]) (use e st n w in simp_all)
  from node_collect_I[OF this] show ?thesis by simp
qed
```

The activation-preserving form. Following an intra_path only extends the trace, so the resulting witness is the *same* activation: same caller and same entry node. Callers that must know *which* activation reached a node need this: the collection at a node merges every activation that reaches it.

```
lemma valid_activation_trace_intra_path_extend_pair:
  "intra_path g x y ⟹ t ∈ 𝒯 ⟹ sink_node t = fst x ⟹ sink_store t = snd x
   ⟹ ∃t'. t' ∈ 𝒯 ∧ sink_node t' = fst y ∧ sink_store t' = snd y
            ∧ caller_of t' = caller_of t ∧ fst (hd (path_of t')) = fst (hd (path_of t))"
proof (induction arbitrary: t rule: star.induct)
  case (refl x)
  then show ?case by blast
next
  case (step x y z)
  obtain u s where x: "x = (u, s)" by (cases x)
  obtain w s1 where y: "y = (w, s1)" by (cases y)
  from step.hyps(1) x y obtain a where
    e: "(u, a, w) ∈ intra g" and st: "s1 ∈ edge_step a s"
    by auto
  have ext: "extend t (w, s1) ∈ 𝒯"
    by (rule valid_activation_trace.intra[OF step.prems(1)])
       (use e st step.prems(2,3) x in simp_all)
  have hd_ext: "fst (hd (path_of (extend t (w, s1)))) = fst (hd (path_of t))"
    using valid_activation_trace_path_nonempty[OF step.prems(1)] by simp
  from step.IH[OF ext] y obtain t' where
    "t' ∈ 𝒯" "sink_node t' = fst z" "sink_store t' = snd z"
    "caller_of t' = caller_of (extend t (w, s1))"
    "fst (hd (path_of t')) = fst (hd (path_of (extend t (w, s1))))" by auto
  then show ?case using hd_ext by auto
qed

lemma valid_activation_trace_intra_path_extend:
  assumes p: "intra_path g (sink_node t, sink_store t) (v, s')"
    and t: "t ∈ 𝒯"
  obtains t' where "t' ∈ 𝒯" "sink_node t' = v" "sink_store t' = s'"
    "caller_of t' = caller_of t" "fst (hd (path_of t')) = fst (hd (path_of t))"
  using valid_activation_trace_intra_path_extend_pair[OF p t] by auto
```

A call and its matching return *do* compose, which is what makes the missing single-step closure above harmless in practice. The callee trace is built from the caller's by `valid_activation_trace.call`, so `valid_activation_trace.ret`'s side condition — that the activation being popped is this trace's own caller — holds by construction, and running the callee to its `FunctionResult` along `intra_path` preserves it. What a caller supplies is the call edge and an intra path through the callee; what it gets back is the continuation node with the combined store.

```
lemma node_collect_call_return_step:
  assumes s: "s ∈ 𝒞 u"
      and ce: "(u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls g"
      and body: "intra_path g (FunctionEntry q, call_enter 𝒢 (CallEdge dst pars args) s)
                   (FunctionResult q, t)"
  shows "combine_collect 𝒢 dst s t ∈ 𝒞 cont"
proof -
  from s obtain caller where cv: "caller ∈ 𝒯"
    and cn: "sink_node caller = u" and cs: "sink_store caller = s"
    by (rule node_collect_E)
  let ?entered = "Call caller [(FunctionEntry q, call_enter 𝒢 (CallEdge dst pars args) s)]"
  have ev: "?entered ∈ 𝒯"
    using valid_activation_trace.call [OF cv, where dst = dst and pars = pars and args = args
                            and p = q and cont = cont]
      ce cn cs by simp
  obtain callee where dv: "callee ∈ 𝒯"
    and dn: "sink_node callee = FunctionResult q" and ds: "sink_store callee = t"
    and dc: "caller_of callee = caller_of ?entered"
    by (rule valid_activation_trace_intra_path_extend [OF _ ev]) (use body in simp_all)
  have "Resume caller callee
          (path_of caller @ [(cont, combine_collect 𝒢 dst (sink_store caller) (sink_store callee))])
          ∈ 𝒯"
    by (rule valid_activation_trace.ret [OF dv _ dn]) (use dc ce cn in simp_all)
  from node_collect_I [OF this] show ?thesis
    using cs ds by (simp add: sink_node_def sink_store_def)
qed
```

Collecting equations

(1) Initial stores are collected at `cfg_entry`.

```
lemma node_collect_init:
  "s ∈ S ⟹ s ∈ 𝒞 (cfg_entry g)"
  using node_collect_I[OF valid_activation_trace.root] by simp
```

Context-sensitive / context-insensitive bridge

Bridge (1): every context bucket, and so their union, is included in the context-insensitive collection.

```
theorem activation_collect_le_node_collect:
  "𝒜𝒢,adm,c₀,g,S v c ⊆ 𝒞 v"
  unfolding activation_collect_def node_collect_def by blast

theorem Union_activation_collect_le_node_collect:
  "(⋃c. 𝒜𝒢,adm,c₀,g,S v c) ⊆ 𝒞 v"
  using activation_collect_le_node_collect by blast
```

Bridge (2): the converse needs every valid activation trace to carry some context. That premise is the whole content of the direction – a policy that leaves some trace uncontexted loses the stores on it, and the union then falls short. A functional policy has it outright, since `activation_context_of` is total; a relational one earns it from conditional totality, which is `Activation_Trace_Abstract`'s business. No finiteness assumption either way.

```
theorem node_collect_eq_Union_activation_of_has_context:
  assumes has_ctx: "⋀t. t ∈ 𝒯 ⟹ ∃c. activation_context_rel 𝒢 adm c₀ g t c"
  shows "𝒞 v = (⋃c. 𝒜𝒢,adm,c₀,g,S v c)"
proof
  show "𝒞 v ⊆ (⋃c. 𝒜𝒢,adm,c₀,g,S v c)"
  proof
    fix x assume "x ∈ 𝒞 v"
    then obtain t where t: "t ∈ 𝒯" "sink_node t = v" "sink_store t = x"
      by (rule node_collect_E)
    from has_ctx [OF t(1)] obtain c where "activation_context_rel 𝒢 adm c₀ g t c" ..
    with t show "x ∈ (⋃c. 𝒜𝒢,adm,c₀,g,S v c)"
      by blast
  qed
next
  show "(⋃c. 𝒜𝒢,adm,c₀,g,S v c) ⊆ 𝒞 v"
    by (rule Union_activation_collect_le_node_collect)
qed

theorem node_collect_eq_Union_activation_of_fun:
  "𝒞 v
     = (⋃c. 𝒜𝒢,context_policy_of_fun f,c₀,g,S v c)"
  by (rule node_collect_eq_Union_activation_of_has_context)
     (simp add: activation_context_rel_of_fun_iff)

end
```

The collecting semantics is inhabited

`valid_activation_trace` and `node_collect` are defined over an arbitrary graph, and every soundness statement downstream is an over-approximation claim about them; if they were empty, all of it would hold vacuously. A two-node graph with one assignment edge settles that inside this session, without borrowing a graph from the compiler: the initial store is collected at the entry, and the assigned store is collected one edge later.

```
definition activation_trace_witness_cfg :: cfg where
  "activation_trace_witness_cfg =
     (| intra = {(Statement 0, EA_Assign (STR ''x'') (N 1), Statement 1)},
       calls = {},
       cfg_entry = Statement 0 |)"

lemma activation_trace_witness_entry [simp]: "cfg_entry activation_trace_witness_cfg = Statement 0"
  by (simp add: activation_trace_witness_cfg_def)

lemma activation_trace_witness_edge [simp]:
  "(Statement 0, EA_Assign (STR ''x'') (N 1), Statement 1) ∈ intra activation_trace_witness_cfg"
  by (simp add: activation_trace_witness_cfg_def)

lemma activation_trace_witness_root:
  "Root [(Statement 0, s)] ∈ 𝒯𝒢,activation_trace_witness_cfg,{s}"
proof -
  have "Root [(cfg_entry activation_trace_witness_cfg, s)] ∈ 𝒯𝒢,activation_trace_witness_cfg,{s}"
    by (rule valid_activation_trace.root) simp
  then show ?thesis by simp
qed

theorem node_collect_witness:
  "s ∈ 𝒞𝒢,activation_trace_witness_cfg,{s} (Statement 0)"
  "s(STR ''x'' := 1) ∈ 𝒞𝒢,activation_trace_witness_cfg,{s} (Statement 1)"
```

```
proof -
  have root: "Root [(Statement 0, s)] ∈ 𝒯_{𝒢,activation_trace_witness_cfg,{s}}"
    by (rule activation_trace_witness_root)
  show "s ∈ 𝒞_{𝒢,activation_trace_witness_cfg,{s}} (Statement 0)"
    using node_collect_I[OF root] by (simp add: sink_node_def sink_store_def)
  have "extend (Root [(Statement 0, s)]) (Statement 1, s(STR ''x'' := 1))
          ∈ 𝒯_{𝒢,activation_trace_witness_cfg,{s}}"
    by (rule valid_activation_trace.intra[OF root, where a = "EA_Assign (STR ''x'') (N 1)"
                                          and v = "Statement 1"])
       (simp_all add: activation_trace_witness_cfg_def sink_node_def sink_store_def)
  from node_collect_I[OF this]
  show "s(STR ''x'' := 1) ∈ 𝒞_{𝒢,activation_trace_witness_cfg,{s}} (Statement 1)"
    by (simp add: sink_node_def sink_store_def)
qed

end
```

4.8 Activation_Trace_Abstract

```
theory Activation_Trace_Abstract
  imports Activation_Trace_Collect
begin
```

What an analysis must satisfy to be sound

An analysis claims: "at node v, in context c, only these stores can occur". This theory says what such a claim must satisfy to be believed, and proves that satisfying it is enough — every valid activation trace's final store really does lie in the set claimed for its own node and every context it carries. No domain, solver or dependency graph appears here; they instantiate this.

The claim is a function `cover` from a node and a context to a set of stores, and there are five obligations on it: the seed stores are covered at the entry (`INIT`), each intra edge keeps them covered (`INTRA`), a call covers the entered store at the callee in every context the policy admits (`CALL`), a return covers the combined store back at the caller (`RETURN`), and every covered call admits at least one context (`TOTAL`).

`RETURN` is the load-bearing one. The callee is read at a context admitted for the transition from the caller's own carried context, so a return may only compose a callee whose context came from the caller it is resuming — not any context that happens to cover that callee's entry. This correlation is not an extra parameter to assume: `activation_context_rel_caller_entry` makes it a theorem about `valid_activation_trace`.

`TOTAL` is what makes the buckets meaningful rather than merely safe. A resumed caller may carry a context under which its callee was never assigned one; the callee's exit store would then be bounded by no bucket, and the combined store by nothing. With it, every valid activation trace carries some context, so the context-insensitive collection is exactly the union of the buckets.

A context is not claimed to identify an activation: two activations may carry the same context, and one activation may carry several. The theorem is that each trace lands in every slot it carries.

The abstract interface

`cover` is the claim itself, `adm` the context policy it is indexed by, and c_0 the context the seed stores arrive under. An analysis interprets this once, with its own solved table in place of `cover`; nothing below asks how that table was computed.

```
locale activation_coverage =
  fixes g :: cfg and S :: "store set"
    and cover :: "cfg_node ⇒ 'c ⇒ store set"
    and adm :: "'c context_policy"
    and c₀ :: 'c
    and 𝒢 :: "vname ⇒ bool"
  assumes INIT[intro]: "⋀s. s ∈ S ⟹ s ∈ cover (cfg_entry g) c₀"
    and INTRA[intro]: "⋀u a v c s s'. (u, a, v) ∈ intra g
        ⟹ s ∈ cover u c ⟹ s' ∈ edge_step a s ⟹ s' ∈ cover v c"
    and CALL[intro]: "⋀u dst pars args p cont c c' s.
        (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
        ⟹ s ∈ cover u c
        ⟹ c' ∈ adm u c (call_info_of (CallEdge dst pars args) p) s
             (call_enter 𝒢 (CallEdge dst pars args) s)
        ⟹ call_enter 𝒢 (CallEdge dst pars args) s ∈ cover (FunctionEntry p) c'"
    and RETURN[intro]: "⋀u dst pars args p cont c c' p' s t es.
        (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
        ⟹ s ∈ cover u c
        ⟹ admits_call_context 𝒢 g adm u c p' s es c'
        ⟹ t ∈ cover (FunctionResult p) c'
        ⟹ combine_collect 𝒢 dst s t ∈ cover cont c"
    and TOTAL: "call_context_total_on cover adm 𝒢 g"
begin
```

The trace-level objects at the locale's fixed arguments: the collection $\mathcal{C}$, the valid activation traces $\mathcal{T}$, the context relation `carries`, the buckets $\mathcal{A}$, and the admitted call contexts `admits`.

```
abbreviation collect :: "cfg_node ⇒ store set" ("𝒞")
  where "𝒞 ≡ 𝒞_{𝒢,g,S}"
abbreviation traces :: "activation_trace set" ("𝒯")
  where "𝒯 ≡ 𝒯_{𝒢,g,S}"
abbreviation carries :: "activation_trace ⇒ 'c ⇒ bool"
  where "carries ≡ activation_context_rel 𝒢 adm c₀ g"
abbreviation buckets :: "cfg_node ⇒ 'c ⇒ store set" ("𝒜")
  where "𝒜 ≡ 𝒜_{𝒢,adm,c₀,g,S}"
abbreviation admits :: "cfg_node ⇒ 'c ⇒ pname ⇒ store ⇒ store ⇒ 'c ⇒ bool"
```

```
  where "admits ≡ admits_call_context 𝒢 g adm"
```

`trace_covered u`: `u` carries some context, and its sink store is admitted at its own node under every context it carries.

```
definition trace_covered :: "activation_trace ⇒ bool" where
  "trace_covered u ⟷
     (∃c. carries u c)
     ∧ (∀c. carries u c ⟶ sink_store u ∈ cover (sink_node u) c)"

lemma trace_coveredI:
  assumes "carries u c0"
    and "⋀c. carries u c ⟹ sink_store u ∈ cover (sink_node u) c"
  shows "trace_covered u"
  using assms unfolding trace_covered_def by blast

lemma trace_covered_someE:
  assumes "trace_covered u"
  obtains c where "carries u c" "sink_store u ∈ cover (sink_node u) c"
  using assms unfolding trace_covered_def by blast

lemma trace_coveredD:
  "trace_covered u ⟹ carries u c
   ⟹ sink_store u ∈ cover (sink_node u) c"
  unfolding trace_covered_def by blast
```

Closure under the four constructor clauses

`init_covered`: the main activation's seed store is admitted at c_0, the one context every `Root` carries.

```
lemma init_covered: "s ∈ S ⟹ trace_covered (Root [(cfg_entry g, s)])"
  by (auto simp: trace_covered_def)
```

`intra_covered`: an intra edge preserves the carried contexts (`activation_context_rel_extend`) and covers the concrete step through `INTRA` at each of them.

```
lemma intra_covered:
  assumes e: "(sink_node t, a, v) ∈ intra g"
    and st: "s' ∈ edge_step a (sink_store t)" and pne: "path_of t ≠ []" and iht: "trace_covered t"
  shows "trace_covered (extend t (v, s'))"
  using iht unfolding trace_covered_def
  by (auto simp: activation_context_rel_extend[OF pne] intro: INTRA[OF e _ st])
```

`call_covered`: `TOTAL` gives the new activation some context, from a context its caller carries; `CALL` covers the entered store at every context it carries, each of which names its own edge out of the call site.

```
lemma call_covered:
  assumes e: "(sink_node caller, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and ihc: "trace_covered caller"
  shows "trace_covered (Call caller
           [(FunctionEntry p, call_enter 𝒢 (CallEdge dst pars args) (sink_store caller))])"
proof -
  let ?es = "call_enter 𝒢 (CallEdge dst pars args) (sink_store caller)"
  from ihc obtain c where c: "carries caller c"
    and cov: "sink_store caller ∈ cover (sink_node caller) c"
    by (rule trace_covered_someE)
  obtain c' where
    adm: "admits (sink_node caller) c p (sink_store caller) ?es c'"
    by (rule call_context_total_onE[OF TOTAL e cov])
  have some: "carries (Call caller [(FunctionEntry p, ?es)]) c'"
    using c adm by (rule activation_context_rel.Call)
  show ?thesis
  proof (rule trace_coveredI[OF some])
    fix c' assume "carries (Call caller [(FunctionEntry p, ?es)]) c'"
    then obtain c0 where c0: "carries caller c0"
      and adm0: "admits (sink_node caller) c0 p (sink_store caller) ?es c'"
      by (auto simp: activation_context_rel_Call_iff)
    from adm0 obtain dst' pars' args' cont'
      where e': "(sink_node caller, CallEdge dst' pars' args', FunctionEntry p, cont') ∈ calls g"
        and es: "?es = call_enter 𝒢 (CallEdge dst' pars' args') (sink_store caller)"
        and Rc: "c' ∈ adm (sink_node caller) c0 (call_info_of (CallEdge dst' pars' args') p)
                   (sink_store caller) ?es"
      by (rule admits_call_contextE)
    have "sink_store caller ∈ cover (sink_node caller) c0"
      using ihc c0 by (rule trace_coveredD)
    from CALL[OF e' this Rc[unfolded es]] es
    show "sink_store (Call caller [(FunctionEntry p, ?es)])
```

```
          ∈ cover (sink_node (Call caller [(FunctionEntry p, ?es)])) c'" by simp
  qed
qed
```

`return_covered`: the matched return. For each context `c` the resumed caller carries, `TOTAL` re-enters the callee at a context `c'` admitted from `c` along the callee's own entry edge (`activation_context_rel_caller_entry`, both directions), the callee's hypothesis bounds its exit at `c'`, and `RETURN` composes the two at `c` — never at a context rediscovered independently.

```
lemma return_covered:
  assumes callee_val: "callee ∈ 𝒯"
    and cof: "caller_of callee = Some caller"
    and res: "sink_node callee = FunctionResult p"
    and comb: "(sink_node caller, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and ih_caller: "trace_covered caller"
    and ih_callee: "trace_covered callee"
  shows "trace_covered (Resume caller callee (path_of caller
              @ [(cont, combine_collect 𝒢 dst (sink_store caller) (sink_store callee))]))"
proof -
  let ?u = "Resume caller callee (path_of caller
    @ [(cont, combine_collect 𝒢 dst (sink_store caller) (sink_store callee))])"
  have sn: "sink_node ?u = cont"
    and ss: "sink_store ?u = combine_collect 𝒢 dst (sink_store caller) (sink_store callee)"
    by (simp_all add: sink_node_def sink_store_def)
  from ih_caller obtain c1 where c1: "carries caller c1"
    by (rule trace_covered_someE)
  have some: "carries ?u c1" using c1 by simp
  from ih_callee obtain ce where ce: "carries callee ce"
    by (rule trace_covered_someE)
  obtain c0 p' where hd: "hd (path_of callee) = (FunctionEntry p', entry_store callee)"
    and adm: "admits (sink_node caller) c0 p' (sink_store caller)
                (entry_store callee) ce"
    by (rule activation_context_rel_caller_entryE[OF callee_val cof ce])
  from adm obtain dst' pars' args' cont'
    where e': "(sink_node caller, CallEdge dst' pars' args', FunctionEntry p', cont') ∈ calls g"
      and es: "entry_store callee = call_enter 𝒢 (CallEdge dst' pars' args') (sink_store caller)"
    by (rule admits_call_contextE)
  show ?thesis
  proof (rule trace_coveredI[OF some])
    fix c assume "carries ?u c"
    then have cc: "carries caller c" by simp
    have s_cov: "sink_store caller ∈ cover (sink_node caller) c"
      using ih_caller cc by (rule trace_coveredD)
    obtain c' where adm': "admits (sink_node caller) c p' (sink_store caller)
                             (call_enter 𝒢 (CallEdge dst' pars' args') (sink_store caller)) c'"
      by (rule call_context_total_onE[OF TOTAL e' s_cov])
    have tc': "carries callee c'"
      by (rule activation_context_rel_caller_entryI[OF callee_val cof hd cc adm'[folded es]])
    have t_cov: "sink_store callee ∈ cover (FunctionResult p) c'"
      using trace_coveredD[OF ih_callee tc'] res by simp
    have "combine_collect 𝒢 dst (sink_store caller) (sink_store callee) ∈ cover cont c"
      by (rule RETURN[OF comb s_cov adm'[folded es] t_cov])
    then show "sink_store ?u ∈ cover (sink_node ?u) c" by (simp add: sn ss)
  qed
qed
```

The generic soundness theorem

The bound holds along the whole caller chain, by `valid_activation_trace` rule induction feeding the four closure lemmas. The chain, not a per-node statement, is what the `RETURN` case needs: it recovers its caller structurally, and the caller lies in the callee's chain.

```
lemma caller_chain_covered:
  "t ∈ 𝒯 ⟹ ∀u ∈ callers t. trace_covered u"
proof (rule caller_chain_closure)
  fix s assume "s ∈ S"
  then show "trace_covered (Root [(cfg_entry g, s)])" by (rule init_covered)
next
  fix t a v s' assume ht: "t ∈ 𝒯" and ch: "∀u ∈ callers t. trace_covered u"
    and e: "(sink_node t, a, v) ∈ intra g" and st: "s' ∈ edge_step a (sink_store t)"
  have pt: "path_of t ≠ []" using ht valid_activation_trace_path_nonempty by blast
  have iht: "trace_covered t" using ch callers_refl by blast
  show "trace_covered (extend t (v, s'))" by (rule intra_covered[OF e st pt iht])
next
  fix caller dst pars args p cont
  assume ch: "∀u ∈ callers caller. trace_covered u"
```

```
    and e: "(sink_node caller, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
  have ihc: "trace_covered caller" using ch callers_refl by blast
  show "trace_covered (Call caller
          [(FunctionEntry p, call_enter 𝒢 (CallEdge dst pars args) (sink_store caller))])"
    by (rule call_covered[OF e ihc])
next
  fix callee caller p dst pars args cont
  assume cv: "callee ∈ 𝒯" and ch: "∀u ∈ callers callee. trace_covered u"
    and cof: "caller_of callee = Some caller" and res: "sink_node callee = FunctionResult p"
    and e: "(sink_node caller, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
  have ih_caller: "trace_covered caller" using ch cof callers_caller_subset callers_refl by blast
  have ih_callee: "trace_covered callee" using ch callers_refl by blast
  show "trace_covered (Resume caller callee (path_of caller
            @ [(cont, combine_collect 𝒢 dst (sink_store caller) (sink_store callee))]))"
    by (rule return_covered[OF cv cof res e ih_caller ih_callee])
qed
```

Every valid activation trace carries a context, and its sink lies in every slot it carries: `valid_activation_trace` is soundly over-approximated by `trace_covered`.

```
theorem valid_activation_trace_covered: "t ∈ 𝒯 ⟹ trace_covered t"
proof -
  assume "t ∈ 𝒯"
  then have "∀u ∈ callers t. trace_covered u" by (rule caller_chain_covered)
  then show "trace_covered t" using callers_refl by blast
qed

theorem valid_activation_trace_has_context:
  assumes "t ∈ 𝒯"
  obtains c where "carries t c"
  using valid_activation_trace_covered[OF assms] by (rule trace_covered_someE)

theorem valid_activation_trace_covered_at:
  "t ∈ 𝒯 ⟹ carries t c
   ⟹ sink_store t ∈ cover (sink_node t) c"
  using valid_activation_trace_covered by (rule trace_coveredD)
```

Bridge (2) of `Voblint_CFG.Activation_Trace_Collect`, earned: under `TOTAL` the context-insensitive collection is the union of the buckets.

```
theorem node_collect_eq_Union_activation_collect:
  "𝒞 v = (⋃c. 𝒜 v c)"
proof
  show "𝒞 v ⊆ (⋃c. 𝒜 v c)"
  proof
    fix x assume "x ∈ 𝒞 v"
    then obtain t where t: "t ∈ 𝒯" "sink_node t = v" "sink_store t = x"
      by (rule node_collect_E)
    from valid_activation_trace_has_context[OF t(1)] obtain c where "carries t c" .
    with t show "x ∈ (⋃c. 𝒜 v c)" by blast
  qed
next
  show "(⋃c. 𝒜 v c) ⊆ 𝒞 v"
    by (rule Union_activation_collect_le_node_collect)
qed
```

The two results together: the claims of all contexts contain the context-insensitive collection.

```
corollary node_collect_covered: "𝒞 v ⊆ (⋃c. cover v c)"
proof
  fix x assume "x ∈ 𝒞 v"
  then obtain t where t: "t ∈ 𝒯" "sink_node t = v" "sink_store t = x"
    by (rule node_collect_E)
  from valid_activation_trace_has_context[OF t(1)] obtain c where "carries t c" .
  then have "x ∈ cover v c"
    using valid_activation_trace_covered_at[OF t(1)] t(2,3) by blast
  then show "x ∈ (⋃c. cover v c)" by blast
qed

end
```

Non-vacuity

The interface is satisfiable for every graph, seed set, and initial context: choosing the top cover `cover = (λ_ _. UNIV)` together with any total functional context policy discharges all five obligations, `TOTAL` included, since a functional policy is total (`call_context_total_on_of_fun`). It is not satisfiable at an arbitrary `adm`: taking

`adm = (λ_ _ _ _ _. {})` fails `TOTAL` for any graph with a reachable call, however wide `cover` is chosen — `TOTAL` genuinely constrains the context policy, not merely the cover.

```
lemma activation_coverage_exists:
  "activation_coverage g S (λ_ _. UNIV) (context_policy_of_fun (λ_ _ _. ())) () 𝒢"
  by unfold_locales auto
```

Monovariant semantic post-fixpoint

Set-valued entry, edge, call, and combine closure of a candidate node map `B` bounds the stack-faithful `node_collect` at every node. The proof interprets `activation_coverage` at `cover v _ = B v` with the trivial one-point context, whose policy is total outright; its combine obligation follows one caller and one callee exit.

```
lemma node_collect_semantic_postfix:
  fixes g :: cfg and B :: "cfg_node ⇒ store set"
    and S0 :: "store set" and v :: cfg_node and 𝒢 :: "vname ⇒ bool"
  assumes entry: "S0 ⊆ B (cfg_entry g)"
    and edge: "⋀u a w s s'. (u, a, w) ∈ intra g
        ⟹ s ∈ B u ⟹ s' ∈ edge_step a s ⟹ s' ∈ B w"
    and call: "⋀u dst pars args p cont s.
        (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
        ⟹ s ∈ B u
        ⟹ call_enter 𝒢 (CallEdge dst pars args) s ∈ B (FunctionEntry p)"
    and combine: "⋀cl dst pars args p cont s t.
        (cl, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
        ⟹ s ∈ B cl ⟹ t ∈ B (FunctionResult p)
        ⟹ combine_collect 𝒢 dst s t ∈ B cont"
  shows "𝒞⟨𝒢,g,S0⟩ v ⊆ B v"
proof -
  interpret G: activation_coverage g S0 "λv _. B v" "context_policy_of_fun (λ_ _ _. ())" "()" 𝒢
  proof (standard, goal_cases INIT INTRA CALL RETURN TOTAL)
    case (INIT s) then show ?case using entry by auto
  next
    case (INTRA u a w c s s') then show ?case using edge by simp
  next
    case (CALL u dst pars args p cont c c' s) then show ?case using call by simp
  next
    case (RETURN cl dst pars args p cont c1 c' p' s t es) then show ?case using combine by simp
  next
    case TOTAL then show ?case by simp
  qed
  show ?thesis
  proof (rule subsetI)
    fix x assume "x ∈ 𝒞⟨𝒢,g,S0⟩ v"
    then obtain u where u: "u ∈ 𝒯⟨𝒢,g,S0⟩" "sink_node u = v" "sink_store u = x"
      by (rule node_collect_E)
    from G.valid_activation_trace_has_context[OF u(1)] obtain c
      where c: "activation_context_rel 𝒢 (context_policy_of_fun (λ_ _ _. ())) () g u c" .
    have "sink_store u ∈ B (sink_node u)" using G.valid_activation_trace_covered_at[OF u(1) c]
      by simp
    then show "x ∈ B v" using u(2,3) by simp
  qed
qed


end
```

Chapter 5

Voblint_Compile

5.1 VIMP_Proc_to_CFG

```
theory VIMP_Proc_to_CFG
  imports "Voblint_CFG.CFG_Def"
begin
```

Procedure-aware CFG compilation for `com` programs

Compile a `com` program directly into the two-relation procedure-aware CFG. Nodes are `Statement n` (allocated from a counter), `FunctionEntry p`, and `FunctionResult p`. The compiler threads the enclosing procedure name `p`, so a source `Return` and the normal fall-through both reach `FunctionResult p` through an `EA_Ret` intra edge.

Compilation is continuation passing: the node reached when a fragment falls through is an input, not a result. A node therefore denotes a program point between transfer functions, and no fragment invents a node to hold a control flow that its own source construct does not have.

Layout of one procedure `p` with body `b`:

- `FunctionEntry p --EA_Body p--> entry(b)`;
- the compiled body edges, with the epilogue node as the body's continuation;
- `epilogue --EA_Ret None p--> FunctionResult p` (normal fall-through returns without a value);
- each source `Return e` inside `b` compiles to `--EA_Ret e p--> FunctionResult p` and ignores its continuation, so an early return leaves no node behind.

Control effects and their source counterparts:

- source `Return` becomes an explicit `EA_Ret` edge into `FunctionResult`, which also does the work the runtime `Unwind` does at source level;
- source activation handling (`Restore` and the frame pop) becomes `FunctionResult` + the `calls`-edge continuation.

Calls are emitted only into `calls`; a call never appears as an executable intra edge. `Restore` / `Unwind` are runtime-only (excluded by `source_com`); their compile clauses are trivial one-node stubs that never occur for a source program.

Static allocation size

`csize c` is the number of `Statement` indices the fragment of `c` allocates, fixed by the source syntax alone. `compile` needs it before it recurses: `Seq` names the entry node of `c2` — the continuation it hands to `c1` — and that node is `Statement (n + csize c1)`. `compile_next_id` below ties the two together, so the two definitions cannot drift apart silently.

```
fun csize :: "'a acom ⇒ nat" where
  "csize (SKIP i) = 1"
| "csize (Assign i x a) = 1"
| "csize (Check i l c) = 1"
| "csize (Seq c1 c2) = csize c1 + csize c2"
| "csize (If i b c1 c2) = 1 + csize c1 + csize c2"
| "csize (While i b c) = 1 + csize c"
| "csize (Call i dst q actuals) = 1"
| "csize (Return i e) = 1"
| "csize (Restore i) = 1"
| "csize (Unwind i) = 1"

lemma csize_pos: "0 < csize c"
  by (induction c) auto
```

Command compilation

`compile Π p c k n` returns `(n', entry, intra_edges, call_edges)`, allocating `Statement` indices in the counter range `[n, n')`. The continuation `k` is an *input*: it is the node normal execution reaches when the fragment falls through. A fragment therefore never reports an exit, and a command that cannot fall through simply ignores `k`.

Every fragment enters at `Statement n`, so the entry is determined by the counter and is reported only for convenience. The procedure table Π is threaded so a `Call` can label its `CallEdge` with the callee's declared formals; every other clause ignores it.

Consequences of handing the continuation down rather than reporting an exit:

- `Seq` needs no connecting `EA_Nop`: `c1`'s continuation *is* `c2`'s entry;
- `gcom.If` allocates no merge node: both branches receive the same `k`, and the join happens at `k`, which is the program point after the conditional;
- `While` needs no back-edge node: the body's continuation is the loop head;
- `Return` ignores `k` and emits only its `EA_Ret` edge, so no unreachable continuation node is invented;
- `Call` resumes at `k`, so the resume point is the following program point. `Restore` and `Unwind` are runtime-only (excluded by `source_com`), so their clauses are never exercised for a source program. They stay transparent: one node and a `EA_Nop` to the continuation. Making them dead ends instead would cost the unconditional form of `compile_prog_entry_cfg_reaches_exit` for no gain, since no source program reaches these clauses.

A call edge into an undeclared procedure carries no formals: `Call` compiles against whatever Π `q` says at compile time, declared or not, so its formal list is this classifier rather than a lookup that could fail. Naming it lets lemma statements about compiled call edges cite `call_formals` instead of restating the case split; the `[simp]` tag keeps it transparent to existing automation.

```
definition call_formals :: "'a ptable ⇒ pname ⇒ vname list" where
  [simp]: "call_formals Π q =
    (case Π q of
       Some decl ⇒ formals decl
     | None ⇒ Code.abort (STR ''call_formals: call to an undeclared procedure'') (λ_. []))"

fun compile ::
  "'a ptable ⇒ pname ⇒ 'a acom ⇒ cfg_node ⇒ nat
   ⇒ nat × cfg_node
       × (cfg_node × edge_action × cfg_node) set
       × (cfg_node × call_action × cfg_node × cfg_node) set"
where
  "compile Π p (SKIP i) k n =
     (Suc n, Statement n, {(Statement n, EA_Nop, k)}, {})"
| "compile Π p (Assign i x a) k n =
     (Suc n, Statement n, {(Statement n, EA_Assign x a, k)}, {})"
| "compile Π p (Check i l c) k n =
     (Suc n, Statement n, {(Statement n, EA_Check l c, k)}, {})"
| "compile Π p (Seq c1 c2) k n =
     (let (n1, en1, E1, K1) = compile Π p c1 (Statement (n + csize c1)) n;
          (n2, en2, E2, K2) = compile Π p c2 k (n + csize c1)
      in (n2, en1, E1 ∪ E2, K1 ∪ K2))"
| "compile Π p (If i b c1 c2) k n =
     (let (n1, en1, E1, K1) = compile Π p c1 k (Suc n);
          (n2, en2, E2, K2) = compile Π p c2 k n1
      in (n2, Statement n,
          {(Statement n, EA_Assume b, if is_SKIP c1 then k else en1),
           (Statement n, EA_AssumeNot b, if is_SKIP c2 then k else en2)}
            ∪ (if is_SKIP c1 then {} else E1) ∪ (if is_SKIP c2 then {} else E2),
          K1 ∪ K2))"
| "compile Π p (While i b c) k n =
     (let (n1, en1, E1, K1) = compile Π p c (Statement n) (Suc n)
      in (n1, Statement n,
          {(Statement n, EA_Assume b, en1), (Statement n, EA_AssumeNot b, k)} ∪ E1,
          K1))"
| "compile Π p (Call i dst q actuals) k n =
     (case special_table q of
        Some desc ⇒
          (case classify_special desc actuals of
             None ⇒ (Suc n, Statement n, {(Statement n, EA_Nop, k)}, {})
           | Some sc ⇒
               (case dst of
                  Some x ⇒
                    (Suc n, Statement n, {(Statement n, EA_Special sc x, k)}, {})
                | None ⇒
                    (Suc n, Statement n, {(Statement n, EA_Nop, k)}, {})))
      | None ⇒
          (Suc n, Statement n, {},
           {(Statement n,
             CallEdge dst (call_formals Π q) actuals,
             FunctionEntry q, k)}))"
| "compile Π p (Return i e) k n =
     (Suc n, Statement n, {(Statement n, EA_Ret e p, FunctionResult p)}, {})"
| "compile Π p (Restore i) k n =
     (Suc n, Statement n, {(Statement n, EA_Nop, k)}, {})"
```

```
| "compile Π p (Unwind i) k n =
     (Suc n, Statement n, {(Statement n, EA_Nop, k)}, {})"
```

Procedure and program compilation

A procedure wraps its body between `FunctionEntry p` and `FunctionResult p`. The body is compiled with the *epilogue* node `Statement r` as its continuation, and the epilogue carries the implicit `EA_Ret None p`: normal fall-through returns without a value. A value is published only by an explicit source `Return e`, which compiles to its own `EA_Ret (Some e) p` edge inside the body.

The epilogue exists so that every edge into `FunctionResult p` is a return edge. Passing `FunctionResult p` directly as the body continuation would let a trailing assignment target the result node. It is allocated *after* the body, so statement indices stay in source order and `r` lies inside the procedure's own counter range — which is what keeps edge locality a statement about one fragment.

The epilogue edge is emitted for every body, also for one all of whose paths end in an explicit `Return`; there the epilogue is dead code. Emitting it unconditionally makes every node of a procedure reach its result, which the solver's coverage argument relies on (`prog_node_reaches`).

```
definition compile_proc ::
  "'a ptable ⇒ pname ⇒ 'a gproc_decl ⇒ nat
   ⇒ nat × (cfg_node × edge_action × cfg_node) set
       × (cfg_node × call_action × cfg_node × cfg_node) set"
where
  "compile_proc Π p decl n =
     (let r = n + csize (body decl);
          (n', ben, E, K) = compile Π p (body decl) (Statement r) n
      in (Suc r,
          insert (FunctionEntry p, EA_Body p, ben)
            (insert (Statement r, EA_Ret None p, FunctionResult p) E),
          K))"

fun compile_procs ::
  "'a ptable ⇒ pname list ⇒ nat
   ⇒ nat × (cfg_node × edge_action × cfg_node) set
       × (cfg_node × call_action × cfg_node × cfg_node) set"
where
  "compile_procs Π [] n = (n, {}, {})"
| "compile_procs Π (p # ps) n =
     (case Π p of
        None ⇒ compile_procs Π ps n
      | Some decl ⇒
          (let (n1, E, K) = compile_proc Π p decl n;
               (n2, E', K') = compile_procs Π ps n1
           in (n2, E ∪ E', K ∪ K')))"
```

The whole program: every declared procedure, plus `main_body Π` compiled under `prog_main_name` (which the well-formedness predicate keeps disjoint from `ps`). There is no global exit: whole-program completion is `FunctionResult prog_main_name`, and the entry is `FunctionEntry prog_main_name`.

```
definition compile_prog ::
  "'a ptable ⇒ pname list ⇒ cfg"
where
  "compile_prog Π ps =
     (let (n1, Eprocs, Kprocs) = compile_procs Π ps 0;
          (n2, Emain, Kmain) =
            compile_proc Π prog_main_name (|formals = [], body = main_body Π|) n1
      in (| intra = Eprocs ∪ Emain, calls = Kprocs ∪ Kmain,
            cfg_entry = FunctionEntry prog_main_name |))"
```

Decomposition of a whole compiled program into the callee pass and the `main` fragment.

```
lemma compile_prog_intra_split:
  obtains n1 Eprocs Kprocs n2 Emain Kmain where
    "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    "compile_proc Π prog_main_name (|formals = [], body = (main_body Π)|) n1 = (n2, Emain, Kmain)"
    "intra (compile_prog Π ps) = Eprocs ∪ Emain"
    "calls (compile_prog Π ps) = Kprocs ∪ Kmain"
proof -
  obtain n1 Eprocs Kprocs where procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    by (rule prod_cases3)
  obtain n2 Emain Kmain
    where mainc: "compile_proc Π prog_main_name (|formals = [], body = (main_body Π)|) n1
                    = (n2, Emain, Kmain)"
    by (rule prod_cases3)
  show ?thesis
```

```
    by (rule that[OF procs mainc]) (simp_all add: compile_prog_def procs mainc Let_def)
qed

lemma cfg_entry_compile_prog [simp]:
  "cfg_entry (compile_prog Π ps) = FunctionEntry prog_main_name"
  by (simp add: compile_prog_def Let_def split: prod.splits)

lemma cfg_exit_compile_prog [simp]:
  "cfg_exit (compile_prog Π ps) = FunctionResult prog_main_name"
  by (simp add: cfg_exit_def)
```

Allocation arithmetic and statement ranges

A fragment allocates exactly `csize c` indices. This is the obligation the forward numbering rests on: `Seq` names `c2`'s entry before compiling `c1`, so a clause allocating more or fewer nodes than its `csize` claims would leave that continuation dangling. The equation subsumes the counter monotonicity used throughout the ownership lemmas.

```
lemma compile_next_id:
  "compile Π p c k n = (n', en, E, K) ⟹ n' = n + csize c"
proof (induction c arbitrary: k n n' en E K rule: gcom.induct)
  case (Seq c1 c2)
  then obtain n1 en1 E1 K1 n2 en2 E2 K2 where
    c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, en1, E1, K1)"
    and c2: "compile Π p c2 k (n + csize c1) = (n2, en2, E2, K2)"
    and n': "n' = n2"
    by (auto simp: Let_def split: prod.splits)
  from Seq.IH(2)[OF c2] n' show ?case by simp
next
  case (If i b c1 c2)
  then obtain n1 en1 E1 K1 n2 en2 E2 K2 where
    c1: "compile Π p c1 k (Suc n) = (n1, en1, E1, K1)"
    and c2: "compile Π p c2 k n1 = (n2, en2, E2, K2)" and n': "n' = n2"
    by (auto split: prod.splits)
  from If.IH(1)[OF c1] If.IH(2)[OF c2] n' show ?case by simp
next
  case (While i b c)
  then obtain n1 en1 E1 K1 where
    c1: "compile Π p c (Statement n) (Suc n) = (n1, en1, E1, K1)" and n': "n' = n1"
    by (auto split: prod.splits)
  from While.IH[OF c1] n' show ?case by simp
qed (auto split: prod.splits option.splits)
```

Every fragment enters at its base counter. Handing the continuation down does not change this, which is why `SKIP` keeps a node of its own instead of reporting the continuation as its entry.

```
lemma compile_entry:
  "compile Π p c k n = (n', en, E, K) ⟹ en = Statement n"
  by (induction c arbitrary: k n n' en E K rule: gcom.induct)
     (auto simp: Let_def split: prod.splits option.splits)
```

Fragment decomposition

The composite clauses, packaged as elimination rules. Every consumer of `compile` needs the same destructuring — sub-fragments, their base counters, and the edge split — and `csize` makes the sub-counters explicit, so no auxiliary equation has to be carried along.

```
lemma compile_SeqE [elim]:
  assumes "compile Π p (Seq c1 c2) k n = (n', en, E, K)"
  obtains n1 E1 K1 n2 E2 K2 where
    "compile Π p c1 (Statement (n + csize c1)) n = (n1, Statement n, E1, K1)"
    "compile Π p c2 k (n + csize c1) = (n2, Statement (n + csize c1), E2, K2)"
    "en = Statement n" "E = E1 ∪ E2" "K = K1 ∪ K2"
proof -
  obtain n1 en1 E1 K1 where
    c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, en1, E1, K1)"
    by (cases "compile Π p c1 (Statement (n + csize c1)) n") auto
  obtain n2 en2 E2 K2 where
    c2: "compile Π p c2 k (n + csize c1) = (n2, en2, E2, K2)"
    by (cases "compile Π p c2 k (n + csize c1)") auto
  have e1: "en1 = Statement n" using compile_entry[OF c1] .
  have e2: "en2 = Statement (n + csize c1)" using compile_entry[OF c2] .
  from assms c1 c2 have "en = Statement n" "E = E1 ∪ E2" "K = K1 ∪ K2"
    using e1 by (auto simp: Let_def)
  with c1 c2 e1 e2 show ?thesis by (auto intro: that)
```

```
qed

lemma compile_IfE [elim]:
  assumes "compile Π p (If i b c1 c2) k n = (n', en, E, K)"
  obtains n1 E1 K1 n2 E2 K2 where
    "compile Π p c1 k (Suc n) = (n1, Statement (Suc n), E1, K1)"
    "compile Π p c2 k (Suc n + csize c1) = (n2, Statement (Suc n + csize c1), E2, K2)"
    "en = Statement n"
    "E = {(Statement n, EA_Assume b, if is_SKIP c1 then k else Statement (Suc n)),
          (Statement n, EA_AssumeNot b, if is_SKIP c2 then k else Statement (Suc n + csize c1))} ∪ (if is_SKIP c1 then
{} else E1) ∪ (if is_SKIP c2 then {} else E2)"
    "K = K1 ∪ K2"
proof -
  obtain n1 en1 E1 K1 where c1: "compile Π p c1 k (Suc n) = (n1, en1, E1, K1)"
    by (cases "compile Π p c1 k (Suc n)") auto
  have e1: "en1 = Statement (Suc n)" using compile_entry[OF c1] .
  have i1: "n1 = Suc n + csize c1" using compile_next_id[OF c1] by simp
  obtain n2 en2 E2 K2 where c2: "compile Π p c2 k n1 = (n2, en2, E2, K2)"
    by (cases "compile Π p c2 k n1") auto
  have e2: "en2 = Statement (Suc n + csize c1)" using compile_entry[OF c2] i1 by simp
  from assms c1 c2 e1 e2
  have "en = Statement n"
    "E = {(Statement n, EA_Assume b, if is_SKIP c1 then k else Statement (Suc n)),
          (Statement n, EA_AssumeNot b, if is_SKIP c2 then k else Statement (Suc n + csize c1))} ∪ (if is_SKIP c1 then
{} else E1) ∪ (if is_SKIP c2 then {} else E2)"
    "K = K1 ∪ K2"
    by (auto simp: Let_def)
  with c1 c2 e1 e2 i1 show ?thesis by (auto intro: that)
qed

lemma compile_WhileE [elim]:
  assumes "compile Π p (While i b c) k n = (n', en, E, K)"
  obtains n1 E1 K1 where
    "compile Π p c (Statement n) (Suc n) = (n1, Statement (Suc n), E1, K1)"
    "en = Statement n"
    "E = {(Statement n, EA_Assume b, Statement (Suc n)),
          (Statement n, EA_AssumeNot b, k)} ∪ E1"
    "K = K1"
proof -
  obtain n1 en1 E1 K1 where c1: "compile Π p c (Statement n) (Suc n) = (n1, en1, E1, K1)"
    by (cases "compile Π p c (Statement n) (Suc n)") auto
  have e1: "en1 = Statement (Suc n)" using compile_entry[OF c1] .
  from assms c1 e1
  have "en = Statement n"
    "E = {(Statement n, EA_Assume b, Statement (Suc n)),
          (Statement n, EA_AssumeNot b, k)} ∪ E1"
    "K = K1"
    by (auto simp: Let_def)
  with c1 e1 show ?thesis by (auto intro: that)
qed
```

The sub-fragment projections. Descending into one branch needs only that branch's compile tuple and the fact that its edges are among the enclosing fragment's; the full rules above additionally hand back the sibling's counter and edge sets, which such a proof binds and never uses — leaving their types unconstrained — and the exact `E = ...` equation, which it then has to weaken to a subset by hand. These state the weaker fact directly.

```
lemma compile_Seq_leftE [elim]:
  assumes "compile Π p (Seq c1 c2) k n = (n', en, E, K)"
  obtains m F L where
    "compile Π p c1 (Statement (n + csize c1)) n = (m, Statement n, F, L)"
    "F ⊆ E" "L ⊆ K"
proof -
  from assms obtain n1 E1 K1 n2 E2 K2 where
    "compile Π p c1 (Statement (n + csize c1)) n = (n1, Statement n, E1, K1)"
    "compile Π p c2 k (n + csize c1) = (n2, Statement (n + csize c1), E2, K2)"
    "en = Statement n" "E = E1 ∪ E2" "K = K1 ∪ K2"
    by (rule compile_SeqE)
  then show ?thesis by (intro that) auto
qed

lemma compile_Seq_rightE [elim]:
  assumes "compile Π p (Seq c1 c2) k n = (n', en, E, K)"
  obtains m F L where
    "compile Π p c2 k (n + csize c1) = (m, Statement (n + csize c1), F, L)"
    "F ⊆ E" "L ⊆ K"
proof -
```

```
  from assms obtain n1 E1 K1 n2 E2 K2 where
    "compile Π p c1 (Statement (n + csize c1)) n = (n1, Statement n, E1, K1)"
    "compile Π p c2 k (n + csize c1) = (n2, Statement (n + csize c1), E2, K2)"
    "en = Statement n" "E = E1 ∪ E2" "K = K1 ∪ K2"
    by (rule compile_SeqE)
  then show ?thesis by (intro that) auto
qed

lemma compile_If_leftE [elim]:
  assumes "compile Π p (If i b c1 c2) k n = (n', en, E, K)"
      and "¬ is_SKIP c1"
  obtains m F L where
    "compile Π p c1 k (Suc n) = (m, Statement (Suc n), F, L)"
    "F ⊆ E" "L ⊆ K"
proof -
  from assms(1) obtain n1 E1 K1 n2 E2 K2 where
    "compile Π p c1 k (Suc n) = (n1, Statement (Suc n), E1, K1)"
    "compile Π p c2 k (Suc n + csize c1) = (n2, Statement (Suc n + csize c1), E2, K2)"
    "en = Statement n"
    "E = {(Statement n, EA_Assume b, if is_SKIP c1 then k else Statement (Suc n)),
          (Statement n, EA_AssumeNot b, if is_SKIP c2 then k else Statement (Suc n + csize c1))} ∪ (if is_SKIP c1 then
{} else E1) ∪ (if is_SKIP c2 then {} else E2)"
    "K = K1 ∪ K2"
    by (rule compile_IfE)
  with assms(2) show ?thesis by (intro that) auto
qed

lemma compile_If_rightE [elim]:
  assumes "compile Π p (If i b c1 c2) k n = (n', en, E, K)"
      and "¬ is_SKIP c2"
  obtains m F L where
    "compile Π p c2 k (Suc n + csize c1) = (m, Statement (Suc n + csize c1), F, L)"
    "F ⊆ E" "L ⊆ K"
proof -
  from assms(1) obtain n1 E1 K1 n2 E2 K2 where
    "compile Π p c1 k (Suc n) = (n1, Statement (Suc n), E1, K1)"
    "compile Π p c2 k (Suc n + csize c1) = (n2, Statement (Suc n + csize c1), E2, K2)"
    "en = Statement n"
    "E = {(Statement n, EA_Assume b, if is_SKIP c1 then k else Statement (Suc n)),
          (Statement n, EA_AssumeNot b, if is_SKIP c2 then k else Statement (Suc n + csize c1))} ∪ (if is_SKIP c1 then
{} else E1) ∪ (if is_SKIP c2 then {} else E2)"
    "K = K1 ∪ K2"
    by (rule compile_IfE)
  with assms(2) show ?thesis by (intro that) auto
qed

lemma compile_While_bodyE [elim]:
  assumes "compile Π p (While i b c) k n = (n', en, E, K)"
  obtains m F L where
    "compile Π p c (Statement n) (Suc n) = (m, Statement (Suc n), F, L)"
    "F ⊆ E" "L ⊆ K"
proof -
  from assms obtain n1 E1 K1 where
    "compile Π p c (Statement n) (Suc n) = (n1, Statement (Suc n), E1, K1)"
    "en = Statement n"
    "E = {(Statement n, EA_Assume b, Statement (Suc n)),
          (Statement n, EA_AssumeNot b, k)} ∪ E1"
    "K = K1"
    by (rule compile_WhileE)
  then show ?thesis by (intro that) auto
qed
```

A branching fragment's own two edges, for the callers that want them rather than a sub-fragment: the guard edges out of `Statement n`, which are all the fragment adds beyond its branches.

```
lemma compile_If_assume_edges:
  assumes "compile Π p (If i b c1 c2) k n = (n', en, E, K)"
  shows "(Statement n, EA_Assume b, if is_SKIP c1 then k else Statement (Suc n)) ∈ E"
        "(Statement n, EA_AssumeNot b, if is_SKIP c2 then k else Statement (Suc n + csize c1)) ∈ E"
proof -
  from assms obtain n1 E1 K1 n2 E2 K2 where
    "compile Π p c1 k (Suc n) = (n1, Statement (Suc n), E1, K1)"
    "compile Π p c2 k (Suc n + csize c1) = (n2, Statement (Suc n + csize c1), E2, K2)"
    "en = Statement n"
    and E: "E = {(Statement n, EA_Assume b, if is_SKIP c1 then k else Statement (Suc n)),
                 (Statement n, EA_AssumeNot b, if is_SKIP c2 then k else Statement (Suc n + csize c1))} ∪ (if is_SKIP
c1 then {} else E1) ∪ (if is_SKIP c2 then {} else E2)"
```

```
    "K = K1 ∪ K2"
    by (rule compile_IfE)
  show "(Statement n, EA_Assume b, if is_SKIP c1 then k else Statement (Suc n)) ∈ E" using E
    by blast
  show "(Statement n, EA_AssumeNot b, if is_SKIP c2 then k else Statement (Suc n + csize c1)) ∈ E"
    using E by blast
qed

lemma compile_While_assume_edges:
  assumes "compile Π p (While i b c) k n = (n', en, E, K)"
  shows "(Statement n, EA_Assume b, Statement (Suc n)) ∈ E"
        "(Statement n, EA_AssumeNot b, k) ∈ E"
proof -
  from assms obtain n1 E1 K1 where
    "compile Π p c (Statement n) (Suc n) = (n1, Statement (Suc n), E1, K1)"
    "en = Statement n"
    and E: "E = {(Statement n, EA_Assume b, Statement (Suc n)),
                (Statement n, EA_AssumeNot b, k)} ∪ E1"
    "K = K1"
    by (rule compile_WhileE)
  show "(Statement n, EA_Assume b, Statement (Suc n)) ∈ E" using E by blast
  show "(Statement n, EA_AssumeNot b, k) ∈ E" using E by blast
qed
```

The procedure layout, packaged the same way: the body is compiled at `n` with the epilogue as its continuation, the entry bracket points at the body entry `Statement n`, and the epilogue `Statement (n + csize (body decl))` carries the implicit return.

```
lemma compile_procE [elim]:
  assumes "compile_proc Π p decl n = (n', E, K)"
  obtains Eb where
    "compile Π p (body decl) (Statement (n + csize (body decl))) n
       = (n + csize (body decl), Statement n, Eb, K)"
    "E = insert (FunctionEntry p, EA_Body p, Statement n)
          (insert (Statement (n + csize (body decl)), EA_Ret None p, FunctionResult p) Eb)"
    "n' = Suc (n + csize (body decl))"
proof -
  define r where "r = n + csize (body decl)"
  obtain m ben Eb Kb where
    cb: "compile Π p (body decl) (Statement r) n = (m, ben, Eb, Kb)"
    by (rule prod_cases4)
  have e: "ben = Statement n" using compile_entry[OF cb] .
  have i: "m = r" using compile_next_id[OF cb] unfolding r_def by simp
  from assms cb e i
  have "E = insert (FunctionEntry p, EA_Body p, Statement n)
             (insert (Statement r, EA_Ret None p, FunctionResult p) Eb)"
    "K = Kb" "n' = Suc r"
    unfolding compile_proc_def r_def by (auto simp: Let_def)
  with cb e i show ?thesis unfolding r_def by (auto intro: that)
qed

end
```

5.2 Compile_Wellformed

```
theory Compile_Wellformed
  imports VIMP_Proc_to_CFG
begin
```

What is true of the graph the compiler produces

The compiler theory says how a program becomes nodes and edges; this one says what holds of the result. Three facts leave the session. A compiled `calls` set never has two edges out of one node, because each fragment claims fresh `Statement` indices for its call leaves (`compile_prog_calls_source_unique`) — the context-sensitive analyses need it to read a call site's callee off the graph. A compiled graph is finite (`compile_prog_finite`), which every solver instantiation requires. And a compiled graph satisfies `wf_cfg` (`compile_prog_wf`), the compiler-agnostic contract the D/G layer is stated against.

`frag_stmts` is the local vocabulary: the set of `Statement` indices an edge set mentions, which is how a fragment's claim on the counter is expressed and how two procedures are shown not to overlap.

Call-source uniqueness

Each compiled fragment allocates its `Call` leaves at distinct, freshly claimed `Statement` indices, so a compiled `calls` set never has two different edges out of the same source node. This is a structural fact about the compiler, not about `wf_cfg` (which is deliberately compiler-agnostic) or about any particular program: a hand-built CFG whose source node has two distinct outgoing call edges can still violate it, but no `compile_prog` output can, and it is what lets a context-routing hook resolve the one call at a node from the source node alone.

```
lemma compile_calls_source_range:
  "compile Π p c k n = (n', en, E, K) ⟹ (u, act, ce, af) ∈ K
   ⟹ ∃j. u = Statement j ∧ n ≤ j ∧ j < n'"
proof (induction c arbitrary: k n n' en E K rule: gcom.induct)
  case (Seq c1 c2)
  from Seq.prems(1) obtain n1 en1 E1 K1 n2 en2 E2 K2 where
      c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, en1, E1, K1)"
    and c2: "compile Π p c2 k (n + csize c1) = (n2, en2, E2, K2)"
    and n': "n' = n2" and K: "K = K1 ∪ K2"
    by (auto simp: Let_def split: prod.splits)
  have i1: "n1 = n + csize c1" using compile_next_id[OF c1] .
  have i2: "n2 = n + csize c1 + csize c2" using compile_next_id[OF c2] by simp
  from Seq.prems(2) K consider (L) "(u, act, ce, af) ∈ K1" | (R) "(u, act, ce, af) ∈ K2" by auto
  then show ?case
  proof cases
    case L with Seq.IH(1)[OF c1 L] i1 i2 show ?thesis unfolding n' by auto
  next
    case R with Seq.IH(2)[OF c2 R] i1 i2 show ?thesis unfolding n' by auto
  qed
next
  case (If i b c1 c2)
  from If.prems(1) obtain n1 en1 E1 K1 n2 en2 E2 K2 where
      c1: "compile Π p c1 k (Suc n) = (n1, en1, E1, K1)"
    and c2: "compile Π p c2 k n1 = (n2, en2, E2, K2)"
    and n': "n' = n2" and K: "K = K1 ∪ K2"
    by (auto split: prod.splits)
  have i1: "n1 = Suc n + csize c1" using compile_next_id[OF c1] by simp
  have i2: "n2 = n1 + csize c2" using compile_next_id[OF c2] .
  from If.prems(2) K consider (L) "(u, act, ce, af) ∈ K1" | (R) "(u, act, ce, af) ∈ K2" by auto
  then show ?case
  proof cases
    case L with If.IH(1)[OF c1 L] i1 i2 show ?thesis unfolding n' by auto
  next
    case R with If.IH(2)[OF c2 R] i1 i2 show ?thesis unfolding n' by auto
  qed
next
  case (While i b c)
  from While.prems(1) obtain n1 en1 E1 K1 where
      c1: "compile Π p c (Statement n) (Suc n) = (n1, en1, E1, K1)"
    and n': "n' = n1" and K: "K = K1"
    by (auto split: prod.splits)
  from While.prems(2) K have "(u, act, ce, af) ∈ K1" by simp
  with While.IH[OF c1] show ?case unfolding n' by auto
qed (auto split: prod.splits option.splits)
```

Two call sets whose sources lie on either side of one counter value cannot share a source, so each being source-unique makes their union source-unique. Both compound commands that split the counter, `Seq` and `gcom.If`,

close their case with it.

```
lemma calls_source_unique_Un:
  assumes u1: "(u, ca1, ce1, af1) ∈ K1 ⟹ (u, ca2, ce2, af2) ∈ K1
               ⟹ ca1 = ca2 ∧ ce1 = ce2 ∧ af1 = af2"
    and u2: "(u, ca1, ce1, af1) ∈ K2 ⟹ (u, ca2, ce2, af2) ∈ K2
               ⟹ ca1 = ca2 ∧ ce1 = ce2 ∧ af1 = af2"
    and lo: "⋀u ca ce af. (u, ca, ce, af) ∈ K1 ⟹ ∃j. u = Statement j ∧ j < m"
    and hi: "⋀u ca ce af. (u, ca, ce, af) ∈ K2 ⟹ ∃j. u = Statement j ∧ m ≤ j"
    and e1: "(u, ca1, ce1, af1) ∈ K1 ∪ K2" and e2: "(u, ca2, ce2, af2) ∈ K1 ∪ K2"
  shows "ca1 = ca2 ∧ ce1 = ce2 ∧ af1 = af2"
  using u1 u2 lo hi e1
  by (metis Un_iff cfg_node.inject(1) e2 leD)

lemma compile_calls_source_unique:
  "compile Π p c k n = (n', en, E, K) ⟹ (u, ca1, ce1, af1) ∈ K ⟹ (u, ca2, ce2, af2) ∈ K
   ⟹ ca1 = ca2 ∧ ce1 = ce2 ∧ af1 = af2"
proof (induction c arbitrary: k n n' en E K rule: gcom.induct)
  case (Seq c1 c2)
  from Seq.prems(1) obtain n1 en1 E1 K1 n2 en2 E2 K2 where
      cc1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, en1, E1, K1)"
    and cc2: "compile Π p c2 k (n + csize c1) = (n2, en2, E2, K2)"
    and K: "K = K1 ∪ K2"
    by (auto simp: Let_def split: prod.splits)
  have i1: "n1 = n + csize c1" using compile_next_id[OF cc1] .
  show ?case
    by (rule calls_source_unique_Un[OF Seq.IH(1)[OF cc1] Seq.IH(2)[OF cc2], where m = n1])
       (use compile_calls_source_range[OF cc1] compile_calls_source_range[OF cc2] i1
            Seq.prems(2,3) K in ‹fastforce+›)
next
  case (If i b c1 c2)
  from If.prems(1) obtain n1 en1 E1 K1 n2 en2 E2 K2 where
      cc1: "compile Π p c1 k (Suc n) = (n1, en1, E1, K1)"
    and cc2: "compile Π p c2 k n1 = (n2, en2, E2, K2)"
    and K: "K = K1 ∪ K2"
    by (auto split: prod.splits)
  show ?case
    by (rule calls_source_unique_Un[OF If.IH(1)[OF cc1] If.IH(2)[OF cc2], where m = n1])
       (use compile_calls_source_range[OF cc1] compile_calls_source_range[OF cc2]
            If.prems(2,3) K in ‹fastforce+›)
next
  case (While i b c)
  from While.prems(1) obtain n1 en1 E1 K1 where
      cc1: "compile Π p c (Statement n) (Suc n) = (n1, en1, E1, K1)"
    and K: "K = K1"
    by (auto split: prod.splits)
  from While.prems(2) K have "(u, ca1, ce1, af1) ∈ K1" by simp
  moreover from While.prems(3) K have "(u, ca2, ce2, af2) ∈ K1" by simp
  ultimately show ?case using While.IH[OF cc1] by blast
qed (auto split: prod.splits option.splits)

lemma compile_proc_counter_mono:
  "compile_proc Π p decl n = (n', E, K) ⟹ n ≤ n'"
  by auto

lemma compile_proc_calls_source_range:
  assumes "compile_proc Π p decl n = (n', E, K)" and "(u, act, ce, af) ∈ K"
  shows "∃j. u = Statement j ∧ n ≤ j ∧ j < n'"
proof -
  from assms(1) obtain Eb where
      body: "compile Π p (body decl) (Statement (n + csize (body decl))) n
               = (n + csize (body decl), Statement n, Eb, K)"
    and n': "n' = Suc (n + csize (body decl))"
    by (rule compile_procE)
  from compile_calls_source_range[OF body assms(2)] show ?thesis unfolding n' by auto
qed

lemma compile_proc_calls_source_unique:
  assumes "compile_proc Π p decl n = (n', E, K)"
    and "(u, ca1, ce1, af1) ∈ K" and "(u, ca2, ce2, af2) ∈ K"
  shows "ca1 = ca2 ∧ ce1 = ce2 ∧ af1 = af2"
proof -
  from assms(1) obtain Eb where
    body: "compile Π p (body decl) (Statement (n + csize (body decl))) n
             = (n + csize (body decl), Statement n, Eb, K)"
    by (rule compile_procE)
  from compile_calls_source_unique[OF body assms(2,3)] show ?thesis .
```

```
qed

lemma compile_procs_counter_mono:
  "compile_procs Π ps n = (n', E, K) ⟹ n ≤ n'"
proof (induction ps arbitrary: n n' E K)
  case Nil then show ?case by simp
next
  case (Cons p ps)
  show ?case
  proof (cases "Π p")
    case None with Cons show ?thesis by simp
  next
    case (Some decl)
    with Cons.prems obtain n1 E0 K0 n2 E' K' where
        cp: "compile_proc Π p decl n = (n1, E0, K0)"
      and rest: "compile_procs Π ps n1 = (n2, E', K')" and n': "n' = n2"
      by (auto split: prod.splits)
    from compile_proc_counter_mono[OF cp] Cons.IH[OF rest] n' show ?thesis by simp
  qed
qed
```

Every edge of a `compile_procs` pass comes from one declared member's `compile_proc` fragment, compiled at some offset inside the pass's counter range; conversely every declared member contributes such a fragment. The three facts below are what every structural property of a pass reduces to.

```
lemma compile_procs_intra_origin:
  assumes "compile_procs Π ps n = (n', E, K)" and "e ∈ E"
  shows "∃p decl m m' Ep Kp. p ∈ set ps ∧ Π p = Some decl
          ∧ compile_proc Π p decl m = (m', Ep, Kp) ∧ n ≤ m ∧ m' ≤ n' ∧ e ∈ Ep"
  using assms
proof (induction ps arbitrary: n n' E K)
  case Nil then show ?case by simp
next
  case (Cons q qs)
  show ?case
  proof (cases "Π q")
    case None
    with Cons.prems have "compile_procs Π qs n = (n', E, K)" by simp
    from Cons.IH[OF this Cons.prems(2)] show ?thesis by (blast intro: list.set_intros(2))
  next
    case (Some declq)
    from Cons.prems(1) Some obtain n1 Eq Kq n2 E' K' where
        cq: "compile_proc Π q declq n = (n1, Eq, Kq)"
      and rest: "compile_procs Π qs n1 = (n2, E', K')"
      and eq: "n' = n2" "E = Eq ∪ E'"
      by (auto split: prod.splits)
    have mono: "n ≤ n1" "n1 ≤ n2"
      using compile_proc_counter_mono[OF cq] compile_procs_counter_mono[OF rest] .
    show ?thesis
    proof (cases "e ∈ Eq")
      case True
      have "q ∈ set (q # qs) ∧ Π q = Some declq ∧ compile_proc Π q declq n = (n1, Eq, Kq)
              ∧ n ≤ n ∧ n1 ≤ n' ∧ e ∈ Eq"
        using Some cq mono(2) eq(1) True by simp
      then show ?thesis by blast
    next
      case False
      with Cons.prems(2) eq(2) have "e ∈ E'" by simp
      from Cons.IH[OF rest this] obtain p decl m m' Ep Kp where
        "p ∈ set qs" "Π p = Some decl" "compile_proc Π p decl m = (m', Ep, Kp)"
        "n1 ≤ m" "m' ≤ n2" "e ∈ Ep" by blast
      with mono(1) eq(1)
      have "p ∈ set (q # qs) ∧ Π p = Some decl ∧ compile_proc Π p decl m = (m', Ep, Kp)
              ∧ n ≤ m ∧ m' ≤ n' ∧ e ∈ Ep"
        by (auto intro: le_trans)
      then show ?thesis by blast
    qed
  qed
qed

lemma compile_procs_calls_origin:
  assumes "compile_procs Π ps n = (n', E, K)" and "e ∈ K"
  shows "∃p decl m m' Ep Kp. p ∈ set ps ∧ Π p = Some decl
          ∧ compile_proc Π p decl m = (m', Ep, Kp) ∧ n ≤ m ∧ m' ≤ n' ∧ e ∈ Kp"
  using assms
proof (induction ps arbitrary: n n' E K)
  case Nil then show ?case by simp
```

```
next
  case (Cons q qs)
  show ?case
  proof (cases "Π q")
    case None
    with Cons.prems have "compile_procs Π qs n = (n', E, K)" by simp
    from Cons.IH[OF this Cons.prems(2)] show ?thesis by (blast intro: list.set_intros(2))
  next
    case (Some declq)
    from Cons.prems(1) Some obtain n1 Eq Kq n2 E' K' where
        cq: "compile_proc Π q declq n = (n1, Eq, Kq)"
      and rest: "compile_procs Π qs n1 = (n2, E', K')"
      and eq: "n' = n2" "K = Kq ∪ K'"
      by (auto split: prod.splits)
    have mono: "n ≤ n1" "n1 ≤ n2"
      using compile_proc_counter_mono[OF cq] compile_procs_counter_mono[OF rest] .
    show ?thesis
    proof (cases "e ∈ Kq")
      case True
      have "q ∈ set (q # qs) ∧ Π q = Some declq ∧ compile_proc Π q declq n = (n1, Eq, Kq)
            ∧ n ≤ n ∧ n1 ≤ n' ∧ e ∈ Kq"
        using Some cq mono(2) eq(1) True by simp
      then show ?thesis by blast
    next
      case False
      with Cons.prems(2) eq(2) have "e ∈ K'" by simp
      from Cons.IH[OF rest this] obtain p decl m m' Ep Kp where
        "p ∈ set qs" "Π p = Some decl" "compile_proc Π p decl m = (m', Ep, Kp)"
        "n1 ≤ m" "m' ≤ n2" "e ∈ Kp" by blast
      with mono(1) eq(1)
      have "p ∈ set (q # qs) ∧ Π p = Some decl ∧ compile_proc Π p decl m = (m', Ep, Kp)
            ∧ n ≤ m ∧ m' ≤ n' ∧ e ∈ Kp"
        by (auto intro: le_trans)
      then show ?thesis by blast
    qed
  qed
qed

lemma compile_procs_member:
  assumes "compile_procs Π ps n = (n', E, K)" and "p ∈ set ps" and "Π p = Some decl"
  shows "∃m m' Ep Kp. compile_proc Π p decl m = (m', Ep, Kp)
         ∧ n ≤ m ∧ m' ≤ n' ∧ Ep ⊆ E ∧ Kp ⊆ K"
  using assms
proof (induction ps arbitrary: n n' E K)
  case Nil then show ?case by simp
next
  case (Cons q qs)
  show ?case
  proof (cases "Π q")
    case None
    with Cons.prems have "compile_procs Π qs n = (n', E, K)" "p ∈ set qs" by auto
    then show ?thesis using Cons.IH Cons.prems(3) by blast
  next
    case (Some declq)
    from Cons.prems(1) Some obtain n1 Eq Kq n2 E' K' where
        cq: "compile_proc Π q declq n = (n1, Eq, Kq)"
      and rest: "compile_procs Π qs n1 = (n2, E', K')"
      and eq: "n' = n2" "E = Eq ∪ E'" "K = Kq ∪ K'"
      by (auto split: prod.splits)
    have mono: "n ≤ n1" "n1 ≤ n2"
      using compile_proc_counter_mono[OF cq] compile_procs_counter_mono[OF rest] .
    show ?thesis
    proof (cases "p = q")
      case True
      with Cons.prems(3) Some have "decl = declq" by simp
      with True cq mono(2) eq
      have "compile_proc Π p decl n = (n1, Eq, Kq) ∧ n ≤ n ∧ n1 ≤ n' ∧ Eq ⊆ E ∧ Kq ⊆ K"
        by simp
      then show ?thesis by blast
    next
      case False
      with Cons.prems(2) have "p ∈ set qs" by simp
      from Cons.IH[OF rest this Cons.prems(3)] obtain m m' Ep Kp where
        "compile_proc Π p decl m = (m', Ep, Kp)" "n1 ≤ m" "m' ≤ n2" "Ep ⊆ E'" "Kp ⊆ K'"
        by blast
      with mono(1) eq
```

```
    have "compile_proc Π p decl m = (m', Ep, Kp) ∧ n ≤ m ∧ m' ≤ n' ∧ Ep ⊆ E ∧ Kp ⊆ K"
      by (auto intro: le_trans)
    then show ?thesis by blast
  qed
 qed
qed

lemma compile_procs_calls_source_range:
  assumes "compile_procs Π ps n = (n', E, K)" and "(u, act, ce, af) ∈ K"
  shows "∃j. u = Statement j ∧ n ≤ j ∧ j < n'"
  using compile_procs_calls_origin[OF assms] compile_proc_calls_source_range by fastforce

lemma compile_procs_calls_source_unique:
  "compile_procs Π ps n = (n', E, K) ⟹ (u, ca1, ce1, af1) ∈ K ⟹ (u, ca2, ce2, af2) ∈ K
   ⟹ ca1 = ca2 ∧ ce1 = ce2 ∧ af1 = af2"
proof (induction ps arbitrary: n n' E K)
  case Nil then show ?case by simp
next
  case (Cons p ps)
  show ?case
  proof (cases "Π p")
    case None with Cons show ?thesis by simp
  next
    case (Some decl)
    with Cons.prems(1) obtain n1 E0 K0 n2 E' K' where
        cp: "compile_proc Π p decl n = (n1, E0, K0)"
      and rest: "compile_procs Π ps n1 = (n2, E', K')" and K: "K = K0 ∪ K'"
      by (auto split: prod.splits)
    have not_both: "⋀u' act1' ce1' af1' act2' ce2' af2'.
        (u', act1', ce1', af1') ∈ K0 ⟹ (u', act2', ce2', af2') ∈ K' ⟹ False"
      using compile_proc_calls_source_range[OF cp]
            compile_procs_calls_source_range[OF rest]
      by fastforce
    from Cons.prems(2) K have m1: "(u, ca1, ce1, af1) ∈ K0 ∨ (u, ca1, ce1, af1) ∈ K'" by auto
    from Cons.prems(3) K have m2: "(u, ca2, ce2, af2) ∈ K0 ∨ (u, ca2, ce2, af2) ∈ K'" by auto
    consider (LL) "(u, ca1, ce1, af1) ∈ K0" "(u, ca2, ce2, af2) ∈ K0"
           | (RR) "(u, ca1, ce1, af1) ∈ K'" "(u, ca2, ce2, af2) ∈ K'"
           | (LR) "(u, ca1, ce1, af1) ∈ K0" "(u, ca2, ce2, af2) ∈ K'"
           | (RL) "(u, ca1, ce1, af1) ∈ K'" "(u, ca2, ce2, af2) ∈ K0"
      using m1 m2 by blast
    then show ?thesis
    proof cases
      case LL then show ?thesis using compile_proc_calls_source_unique[OF cp] by blast
    next
      case RR then show ?thesis using Cons.IH[OF rest] by blast
    next
      case LR then show ?thesis using not_both by blast
    next
      case RL then show ?thesis using not_both by blast
    qed
  qed
qed
```

Whole-program call-source uniqueness: the fact a routed-context analysis needs to resolve the formals of the one call at a node without ambiguity, for any `compile_prog` output. Combines the declared-procedures pass and the `main` pass the same way the well-formedness theorem combines its own two passes.

```
theorem compile_prog_calls_source_unique:
  assumes "(u, ca1, ce1, af1) ∈ calls (compile_prog Π ps)"
    and "(u, ca2, ce2, af2) ∈ calls (compile_prog Π ps)"
  shows "ca1 = ca2 ∧ ce1 = ce2 ∧ af1 = af2"
proof -
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    and mainc: "compile_proc Π prog_main_name (|formals = [], body = (main_body Π)|) n1
                  = (n2, Emain, Kmain)"
    and g: "calls (compile_prog Π ps) = Kprocs ∪ Kmain"
    by (rule compile_prog_intra_split)
  have not_both: "⋀u' act1' ce1' af1' act2' ce2' af2'.
      (u', act1', ce1', af1') ∈ Kprocs ⟹ (u', act2', ce2', af2') ∈ Kmain ⟹ False"
    using compile_procs_calls_source_range[OF procs] compile_proc_calls_source_range[OF mainc]
    by fastforce
  from assms show ?thesis unfolding g
    using compile_procs_calls_source_unique[OF procs] compile_proc_calls_source_unique[OF mainc]
      not_both by (elim UnE) (blast | (drule not_both; blast))+
qed
```

The `Statement` indices a compiled fragment touches. Sources are always freshly allocated, so they lie in the fragment's own counter range; a target may in addition be the continuation, which belongs to the enclosing construct. `kstmt` names that one extra index, and is empty when the continuation is not a `Statement` node.

```
definition frag_stmts ::
  "(cfg_node × edge_action × cfg_node) set
   ⇒ (cfg_node × call_action × cfg_node × cfg_node) set ⇒ nat set"
where
  "frag_stmts E K =
     {j. ∃a v. (Statement j, a, v) ∈ E} ∪ {j. ∃u a. (u, a, Statement j) ∈ E}
   ∪ {j. ∃act ce af. (Statement j, act, ce, af) ∈ K}
   ∪ {j. ∃u act ce. (u, act, ce, Statement j) ∈ K}"

fun kstmt :: "cfg_node ⇒ nat set" where
  "kstmt (Statement j) = {j}"
| "kstmt (FunctionEntry q) = {}"
| "kstmt (FunctionResult q) = {}"

lemma kstmt_iff [simp]: "j ∈ kstmt v ⟷ v = Statement j"
  by (cases v) auto

lemma frag_stmts_empty [simp]: "frag_stmts {} {} = {}"
  by (simp add: frag_stmts_def)

lemma frag_stmts_Un:
  "frag_stmts (E1 ∪ E2) (K1 ∪ K2) = frag_stmts E1 K1 ∪ frag_stmts E2 K2"
  by (auto simp: frag_stmts_def)

lemma frag_stmts_E_srcI [intro]:
  "(Statement j, a, v) ∈ E ⟹ j ∈ frag_stmts E K"
  unfolding frag_stmts_def by blast

lemma frag_stmts_E_tgtI [intro]:
  "(u, a, Statement j) ∈ E ⟹ j ∈ frag_stmts E K"
  unfolding frag_stmts_def by blast

lemma frag_stmts_K_srcI [intro]:
  "(Statement j, act, ce, af) ∈ K ⟹ j ∈ frag_stmts E K"
  unfolding frag_stmts_def by blast

lemma frag_stmts_K_tgtI [intro]:
  "(u, act, ce, Statement j) ∈ K ⟹ j ∈ frag_stmts E K"
  unfolding frag_stmts_def by blast

lemma compile_frag_stmts_range:
  "compile Π p c k n = (n', en, E, K) ⟹ frag_stmts E K ⊆ {n..<n'} ∪ kstmt k"
proof (induction c arbitrary: k n n' en E K rule: gcom.induct)
  case (Seq c1 c2)
  from Seq.prems obtain n1 en1 E1 K1 n2 en2 E2 K2 where
    c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, en1, E1, K1)"
    and c2: "compile Π p c2 k (n + csize c1) = (n2, en2, E2, K2)"
    and n': "n' = n2" and E: "E = E1 ∪ E2" and K: "K = K1 ∪ K2"
    by (auto simp: Let_def split: prod.splits)
  have i1: "n1 = n + csize c1" using compile_next_id[OF c1] .
  have i2: "n2 = n + csize c1 + csize c2" using compile_next_id[OF c2] by simp
  have r1: "frag_stmts E1 K1 ⊆ {n..<n1} ∪ {n + csize c1}" using Seq.IH(1)[OF c1] by simp
  have r2: "frag_stmts E2 K2 ⊆ {n + csize c1..<n2} ∪ kstmt k" using Seq.IH(2)[OF c2] .
  show ?case
    using r1 r2 i1 i2 csize_pos[of c2]
    unfolding E K n' by (simp only: frag_stmts_Un) auto
next
  case (If i b c1 c2)
  from If.prems obtain n1 E1 K1 n2 E2 K2 where
    c1: "compile Π p c1 k (Suc n) = (n1, Statement (Suc n), E1, K1)"
    and c2: "compile Π p c2 k (Suc n + csize c1) =
      (n2, Statement (Suc n + csize c1), E2, K2)"
    and E: "E = {(Statement n, EA_Assume b, if is_SKIP c1 then k else Statement (Suc n)),
      (Statement n, EA_AssumeNot b, if is_SKIP c2 then k else Statement (Suc n + csize c1))}
     ∪ (if is_SKIP c1 then {} else E1) ∪ (if is_SKIP c2 then {} else E2)"
    and K: "K = K1 ∪ K2"
    by (rule compile_IfE)
  have ids: "n1 = Suc n + csize c1" "n2 = Suc n + csize c1 + csize c2"
    "n' = n + csize (If i b c1 c2)"
    using compile_next_id[OF c1] compile_next_id[OF c2] compile_next_id[OF If.prems]
    by simp_all
  have sub: "frag_stmts E K ⊆ frag_stmts E1 K1 ∪ frag_stmts E2 K2
```

```
      ∪ {n, Suc n, Suc n + csize c1} ∪ kstmt k"
    unfolding E K frag_stmts_def by auto
  show ?case
    using sub If.IH(1)[OF c1] If.IH(2)[OF c2] ids csize_pos[of c1] csize_pos[of c2]
    by fastforce
next
  case (While i b c)
  from While.prems obtain n1 en1 E1 K1 where
    c1: "compile Π p c (Statement n) (Suc n) = (n1, en1, E1, K1)"
    and n': "n' = n1"
    and E: "E = {(Statement n, EA_Assume b, en1), (Statement n, EA_AssumeNot b, k)} ∪ E1"
    and K: "K = {} ∪ K1"
    by (auto split: prod.splits)
  have e1: "en1 = Statement (Suc n)" using compile_entry[OF c1] .
  have i1: "n1 = Suc n + csize c" using compile_next_id[OF c1] .
  have r1: "frag_stmts E1 K1 ⊆ {Suc n..<n1} ∪ {n}" using While.IH[OF c1] by simp
  have guard: "frag_stmts {(Statement n, EA_Assume b, en1),
                           (Statement n, EA_AssumeNot b, k)} {} ⊆ {n..<n1} ∪ kstmt k"
    using e1 i1 csize_pos[of c] by (auto simp: frag_stmts_def)
  have dec: "frag_stmts E K
              = frag_stmts {(Statement n, EA_Assume b, en1),
                            (Statement n, EA_AssumeNot b, k)} {} ∪ frag_stmts E1 K1"
    unfolding E K by (rule frag_stmts_Un)
  show ?case unfolding n' dec using guard r1 i1 csize_pos[of c] by force
qed (auto simp: frag_stmts_def split: option.splits)
```

Finiteness

Each pass emits finitely many edges. Everything downstream that folds over the graph – the join at a node, the equation enumeration, the solver's variable set – needs this and nothing else about the compiler.

```
lemma compile_finite:
  "compile Π p c k n = (n', en, E, K) ⟹ finite E ∧ finite K"
  by (induction c arbitrary: k n n' en E K rule: gcom.induct)
     (auto simp: Let_def split: prod.splits option.splits)

lemma compile_proc_finite:
  "compile_proc Π p decl n = (n', E, K) ⟹ finite E ∧ finite K"
  by (auto simp: compile_proc_def Let_def split: prod.splits dest: compile_finite)

lemma compile_procs_finite:
  "compile_procs Π ps n = (n', E, K) ⟹ finite E ∧ finite K"
proof (induction ps arbitrary: n n' E K)
  case Nil then show ?case by simp
next
  case (Cons p ps)
  show ?case
  proof (cases "Π p")
    case None with Cons show ?thesis by simp
  next
    case (Some decl)
    with Cons.prems obtain n1 E0 K0 n2 E' K' where
      cp: "compile_proc Π p decl n = (n1, E0, K0)"
      and rest: "compile_procs Π ps n1 = (n2, E', K')"
      and E: "E = E0 ∪ E'" and K: "K = K0 ∪ K'"
      by (auto split: prod.splits)
    from compile_proc_finite[OF cp] Cons.IH[OF rest] show ?thesis
      unfolding E K by simp
  qed
qed

lemma compile_prog_finite:
  "finite (intra (compile_prog Π ps))
   ∧ finite (calls (compile_prog Π ps))"
  unfolding compile_prog_def
  by (auto simp: Let_def split: prod.splits
       dest: compile_procs_finite compile_proc_finite)
```

Edge shapes for well-formedness

Every emitted call edge targets a `FunctionEntry`; a call is never an intra edge.

```
lemma compile_call_ce_entry:
  "compile Π p c k n = (n', en, E, K) ⟹ (u, act, ce, af) ∈ K
   ⟹ ∃q. ce = FunctionEntry q"
  by (induction c arbitrary: k n n' en E K)
```

```
    (auto simp: Let_def split: prod.splits if_splits option.splits)
```

Edge endpoints: a source is always an allocated `Statement`; a target is an allocated `Statement`, the own `FunctionResult p`, or the continuation. The continuation disjunct is what continuation passing adds, and it is what lets the two `wf_cfg` conditions below be discharged without carrying a side condition through the induction.

```
lemma compile_E_shape:
  "compile Π p c k n = (n', en, E, K) ⟹ (u, a, v) ∈ E
   ⟹ (∃j. u = Statement j)
      ∧ (v = k ∨ v = FunctionResult p ∨ (∃j. v = Statement j))"
proof (induction c arbitrary: k n n' en E K)
  case (Seq c1 c2)
  then obtain n1 en1 E1 K1 n2 en2 E2 K2 where
    c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, en1, E1, K1)"
    and c2: "compile Π p c2 k (n + csize c1) = (n2, en2, E2, K2)"
    and E: "E = E1 ∪ E2"
    by (auto simp: Let_def split: prod.splits)
  consider (L) "(u, a, v) ∈ E1" | (R) "(u, a, v) ∈ E2"
    using Seq.prems(2) E by auto
  then show ?case
  proof cases
    case L from Seq.IH(1)[OF c1 L] show ?thesis by auto
  next
    case R from Seq.IH(2)[OF c2 R] show ?thesis by auto
  qed
next
  case (If i b c1 c2)
  then obtain n1 en1 E1 K1 n2 en2 E2 K2 where
    c1: "compile Π p c1 k (Suc n) = (n1, en1, E1, K1)"
    and c2: "compile Π p c2 k n1 = (n2, en2, E2, K2)"
    and E: "E = {(Statement n, EA_Assume b, if is_SKIP c1 then k else en1), (Statement n, EA_AssumeNot b, if is_SKIP
c2 then k else en2)}
                ∪ (if is_SKIP c1 then {} else E1) ∪ (if is_SKIP c2 then {} else E2)"
    by (auto split: prod.splits)
  have e1: "en1 = Statement (Suc n)" using compile_entry[OF c1] .
  have e2: "en2 = Statement n1" using compile_entry[OF c2] .
  consider (Guard) "(u, a, v) ∈ {(Statement n, EA_Assume b, if is_SKIP c1 then k else en1),
                                 (Statement n, EA_AssumeNot b, if is_SKIP c2 then k else en2)}"
    | (L) "(u, a, v) ∈ E1" | (R) "(u, a, v) ∈ E2"
    using If.prems(2) E by (auto split: if_splits)
  then show ?case
  proof cases
    case Guard then show ?thesis using e1 e2 by (auto split: if_splits)
  next
    case L from If.IH(1)[OF c1 L] show ?thesis by auto
  next
    case R from If.IH(2)[OF c2 R] show ?thesis by auto
  qed
next
  case (While i b c)
  then obtain n1 en1 E1 K1 where
    c1: "compile Π p c (Statement n) (Suc n) = (n1, en1, E1, K1)"
    and E: "E = {(Statement n, EA_Assume b, en1), (Statement n, EA_AssumeNot b, k)} ∪ E1"
    by (auto split: prod.splits)
  have e1: "en1 = Statement (Suc n)" using compile_entry[OF c1] .
  consider (Guard) "(u, a, v) ∈ {(Statement n, EA_Assume b, en1),
                                 (Statement n, EA_AssumeNot b, k)}"
    | (Body) "(u, a, v) ∈ E1"
    using While.prems(2) E by auto
  then show ?case
  proof cases
    case Guard then show ?thesis using e1 by auto
  next
    case Body from While.IH[OF c1 Body] show ?thesis by auto
  qed
qed (auto split: prod.splits option.splits)
```

Every `EA_Ret` edge lands on the matching `FunctionResult` — the enclosing procedure name in the action equals the one in the target node.

```
lemma compile_ret_wf:
  "compile Π p c k n = (n', en, E, K) ⟹ (u, EA_Ret e q, v) ∈ E
   ⟹ ∀r. k ≠ FunctionResult r ⟹ v = FunctionResult q"
  by (induction c arbitrary: k n n' en E K)
     (auto simp: Let_def split: prod.splits if_splits option.splits)

lemma compile_proc_call_ce_entry:
```

```
  "compile_proc Π p decl n = (n', E, K) ⟹ (u, act, ce, af) ∈ K
   ⟹ ∃q. ce = FunctionEntry q"
  by (auto simp: compile_proc_def Let_def split: prod.splits
       dest: compile_call_ce_entry)

lemma compile_proc_intra_tgt_not_entry:
  "compile_proc Π p decl n = (n', E, K) ⟹ (u, a, v) ∈ E ⟹ v ≠ FunctionEntry q"
  by (auto simp: compile_proc_def Let_def split: prod.splits if_splits
       dest: compile_E_shape compile_entry)

lemma compile_proc_ret_wf:
  "compile_proc Π p decl n = (n', E, K) ⟹ (u, EA_Ret e q, v) ∈ E
   ⟹ v = FunctionResult q"
  by (auto simp: compile_proc_def Let_def split: prod.splits if_splits
       dest: compile_ret_wf)

lemma compile_procs_call_ce_entry:
  "compile_procs Π ps n = (n', E, K) ⟹ (u, act, ce, af) ∈ K
   ⟹ ∃q. ce = FunctionEntry q"
  using compile_procs_calls_origin compile_proc_call_ce_entry by metis

lemma compile_procs_intra_tgt_not_entry:
  "compile_procs Π ps n = (n', E, K) ⟹ (u, a, v) ∈ E ⟹ v ≠ FunctionEntry q"
  using compile_procs_intra_origin compile_proc_intra_tgt_not_entry by metis

lemma compile_procs_ret_wf:
  "compile_procs Π ps n = (n', E, K) ⟹ (u, EA_Ret e q, v) ∈ E ⟹ v = FunctionResult q"
  using compile_procs_intra_origin compile_proc_ret_wf by metis
```

The compiled program is well-formed

The per-pass shape facts above, collected into the single predicate the CFG layer states its own theorems over. Nothing after this point has to know that a graph was compiled rather than written by hand.

```
theorem compile_prog_wf: "wf_cfg (compile_prog Π ps)"
proof -
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    and mainc: "compile_proc Π prog_main_name (|formals = [], body = (main_body Π)|) n1
                = (n2, Emain, Kmain)"
    and g: "intra (compile_prog Π ps) = Eprocs ∪ Emain"
           "calls (compile_prog Π ps) = Kprocs ∪ Kmain"
    by (rule compile_prog_intra_split)
  show ?thesis
    unfolding wf_cfg_def g
    using compile_procs_call_ce_entry[OF procs] compile_proc_call_ce_entry[OF mainc]
      compile_procs_intra_tgt_not_entry[OF procs] compile_proc_intra_tgt_not_entry[OF mainc]
      compile_procs_ret_wf[OF procs] compile_proc_ret_wf[OF mainc]
    by (intro conjI allI impI; elim UnE) blast+
qed
```

Naming one compiled graph

A worked example names its graph with a definition and then wants the five facts above about that name rather than about `compile_prog`. Restating them per example costs a proof each and re-derives nothing: the facts hold of every compiled graph, with no well-formedness hypothesis. Interpreting this locale at the defining equation supplies all five at once, already phrased in the example's own name.

```
locale compiled_cfg = finite_cfg g
  for Π :: "'a ptable" and ps :: "pname list" and g :: cfg +
  assumes cfg_eq: "g = compile_prog Π ps"
begin
```

`entry` and `exit` are deliberately not `[simp]`: an interpretation would make them global rewrites that unfold `cfg_entry g` under the example's own graph name, which the placement examples reason about symbolically. The finiteness facts (`finite_intra`, `finite_calls`) come from the `finite_cfg` sublocale and carry no such risk.

```
lemma entry: "cfg_entry g = FunctionEntry prog_main_name"
  unfolding cfg_eq by (rule cfg_entry_compile_prog)

lemma exit: "cfg_exit g = FunctionResult prog_main_name"
  unfolding cfg_eq by (rule cfg_exit_compile_prog)

lemma wf: "wf_cfg g"
  unfolding cfg_eq by (rule compile_prog_wf)
```

```
end

end
```

5.3 Compile_Invariants

```
theory Compile_Invariants
  imports Compile_Wellformed "Voblint_VIMP.VIMP_Program" "Voblint_CFG.CFG_Prune"
begin
```

What a program must satisfy, and what its graph then satisfies

Two things live here. `wf_compile_input` is the contract a source program has to meet before it is worth compiling: the procedure names are distinct, the entry procedure is declared and takes no arguments, and every body is a well-formed source command. `wf_program_compile_input_exec` is the executable form of that contract, and the check the command-line tool actually runs — the contract itself quantifies over the whole procedure table, which no generated code can enumerate, so the executable form walks the declaration list instead and is proved sufficient.

The remaining results are about the graph that comes out: a source `Return` compiles to an edge into its own `FunctionResult`, every statement index is listed in source post-order (`prog_stmt_post_order`) so a report can name its source command, and the entry node reaches the exit.

Compiler input well-formedness

`prog_main_name` names the distinguished entry procedure, declared in Π with an empty formal list; `main_body` Π reads its body back out. The callee list `ps` enumerates the other declared procedures (`dom` Π minus `prog_main_name`), so the entry's nodes never collide with a callee's, yet `FunctionEntry prog_main_name` is an ordinary activation of a declared procedure like any other.

```
definition wf_compile_input ::
  "(vname => bool) => 'a ptable => pname list => bool" where
  "wf_compile_input 𝒢 Π ps ⟷
     distinct ps ∧
     set ps = {p. Π p ~= None} - {prog_main_name} ∧
     prog_main_name ∉ set ps ∧
     wf_source_program 𝒢 Π"
```

The leaf definitions an executable instance unfolds to decide the contract. No abstract proof unfolds them, so they only ever fire under `unfolding wf_compile_input_simps`.

```
declare
  reserved_ret_var_def [wf_compile_input_simps]
  source_exp_def [wf_compile_input_simps]
  valid_formal_def [wf_compile_input_simps]
  value_providing_def [wf_compile_input_simps]
  special_table_def [wf_compile_input_simps]
  special_pname_nondet_int_def [wf_compile_input_simps]
  special_pname_min_def [wf_compile_input_simps]
  special_pname_max_def [wf_compile_input_simps]
  ret_var_def [wf_compile_input_simps]
  prog_main_name_def [wf_compile_input_simps]
  main_body_def [wf_compile_input_simps]

lemma wf_compile_inputD:
  assumes "wf_compile_input 𝒢 Π ps"
  shows "reserved_ret_var 𝒢" and "Π prog_main_name = Some (|formals = [], body = (main_body Π)|)"
    and "wf_source_com Π (main_body Π)" and "no_return (main_body Π)"
    and "Π p = Some decl ⟹ wf_proc_decl 𝒢 Π decl"
    and "Π p = Some decl ⟹ special_table p = None"
    and "source_pi Π" and "source_com (main_body Π)"
    and "distinct ps" and "set ps = {p. Π p ≠ None} - {prog_main_name}"
    and "prog_main_name ∉ set ps"
  using wf_source_programD
    assms[unfolded wf_compile_input_def] by auto

lemmas wf_compile_input_reserved_ret_var [dest] = wf_compile_inputD(1)

abbreviation wf_program_compile_input :: "imp_prog ⇒ bool" where
  "wf_program_compile_input p ≡
    wf_compile_input (declared_global p) (prog_table p) (prog_procs p)"
```

An executable, sufficient reformulation

`wf_source_program`'s two $\forall$`p. Π p = ... ⟶ ...` conjuncts range over Π `:: pname => proc_decl option`'s entire (unbounded) function domain, so `wf_program_compile_input` cannot be exported/evaluated as literally stated

– Isabelle's code generator has nothing to enumerate a function's domain against. `proc_rep` is always finite in practice (`prog_table p = map_of (proc_rep p)`), so both conjuncts are provably implied by the corresponding `list_all` check over `proc_rep p` directly: every entry `proc_rep p` actually lists is checked, which is at least as strong as checking only the ones `map_of` would resolve a lookup to.

`wf_program_compile_input_exec` only needs to be *sufficient* for the CLI well-formedness gate to be sound (never silently accepting a source program `wf_program_compile_input` would reject) – not a decision procedure complete in the other direction, which `map_of`'s first-occurrence-wins semantics on a `proc_rep` with a shadowed duplicate key would make materially harder to establish for no operational benefit here.

```
definition wf_program_compile_input_exec :: "imp_prog => bool" where
  "wf_program_compile_input_exec p ⟷
     (let procs = proc_rep p; 𝒢 = declared_global p; pi = map_of procs
      in reserved_ret_var 𝒢 ∧
         distinct (prog_procs p) ∧
         set (prog_procs p) = set (map fst procs) - {prog_main_name} ∧
         prog_main_name ∉ set (prog_procs p) ∧
         pi prog_main_name = Some (⦇formals = [], body = prog_main p⦈) ∧
         wf_source_com pi (prog_main p) ∧ no_return (prog_main p) ∧
         list_all (λ(_, decl). wf_proc_decl 𝒢 pi decl) procs ∧
         list_all (λ(q, _). special_table q = None) procs)"

lemma wf_program_compile_input_exec_sound:
  assumes "wf_program_compile_input_exec p"
  shows "wf_program_compile_input p"
proof -
  define procs where "procs = proc_rep p"
  define 𝒢 where "𝒢 = declared_global p"
  define pi where "pi = map_of procs"
  have pi_eq: "pi = prog_table p"
    unfolding pi_def procs_def prog_table_def ..
  from assms have h:
    "reserved_ret_var 𝒢"
    "distinct (prog_procs p)"
    "set (prog_procs p) = set (map fst procs) - {prog_main_name}"
    "prog_main_name ∉ set (prog_procs p)"
    "pi prog_main_name = Some (⦇formals = [], body = prog_main p⦈)"
    "wf_source_com pi (prog_main p)"
    "no_return (prog_main p)"
    "list_all (λ(_, decl). wf_proc_decl 𝒢 pi decl) procs"
    "list_all (λ(q, _). special_table q = None) procs"
    unfolding wf_program_compile_input_exec_def procs_def 𝒢_def pi_def Let_def
    by simp_all
  have domain: "set (map fst procs) = {q. pi q ≠ None}"
    unfolding pi_def by (induction procs) auto
  have every_decl: "⋀q decl. pi q = Some decl ⟹ wf_proc_decl 𝒢 pi decl"
  proof -
    fix q decl assume "pi q = Some decl"
    then have "(q, decl) ∈ set procs" unfolding pi_def by (rule map_of_SomeD)
    with h(8) show "wf_proc_decl 𝒢 pi decl" by (auto simp: list_all_iff)
  qed
  have every_special: "⋀q. pi q ≠ None ⟹ special_table q = None"
  proof -
    fix q assume "pi q ≠ None"
    then    obtain decl where "(q, decl) ∈ set procs"
      using domain by force
    with h(9) show "special_table q = None" by (auto simp: list_all_iff)
  qed
  have main_eq: "main_body pi = prog_main p" using pi_eq by simp
  have wf_source_program: "wf_source_program 𝒢 pi"
    unfolding wf_source_program_def main_eq
    using h(1) h(5) h(6) h(7) every_decl every_special by blast
  have wf_compile_input: "wf_compile_input 𝒢 pi (prog_procs p)"
    unfolding wf_compile_input_def
    using h(2) h(3) h(4) wf_source_program
    using domain by blast
  show ?thesis
    unfolding 𝒢_def[symmetric] pi_eq[symmetric] using wf_compile_input .
qed
```

`compile_prog` takes a `proc_table` and a `pname list`, so a domain holding only an `imp_prog` needs the two-projection wrapper below. The entry name is `prog_main_name` and its body is looked up, so neither is threaded through as a parameter. This is also the name the OCaml export publishes: a second wrapper for the export alone would put the same function in the generated module twice.

```
definition prog_cfg :: "imp_prog => cfg" where
```

```
  "prog_cfg p = compile_prog (prog_table p) (prog_procs p)"
```

Syntactic occurrence predicates

`returns_in` asks a purely syntactic question – does this command contain a `Return` of exactly this expression – which is the shape the compiler lemmas below need. They induct over the source command, so their hypothesis has to be readable off the syntax rather than off the graph the compiler produced from it.

```
fun returns_in :: "exp option ⇒ 'a acom ⇒ bool" where
  "returns_in e (Return i e') = (e = e')"
| "returns_in e (Seq c1 c2) = (returns_in e c1 ∨ returns_in e c2)"
| "returns_in e (If i _ c1 c2) = (returns_in e c1 ∨ returns_in e c2)"
| "returns_in e (While i _ c) = returns_in e c"
| "returns_in e _ = False"
```

Return edges and call-freeness

Every source `Return e` in procedure `p` compiles to an intra edge that reaches `FunctionResult p` through `EA_Ret e p`.

```
lemma compile_return_edge:
  "compile Π p c k n = (n', en, E, K) ⟹ returns_in e c
   ⟹ ∃j. (Statement j, EA_Ret e p, FunctionResult p) ∈ E"
proof (induction c arbitrary: k n n' en E K rule: gcom.induct)
  case (Seq c1 c2)
  from Seq.prems(1) obtain n1 en1 E1 K1 n2 en2 E2 K2 where
    c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, en1, E1, K1)"
    and c2: "compile Π p c2 k (n + csize c1) = (n2, en2, E2, K2)"
    and E: "E = E1 ∪ E2"
    by (auto simp: Let_def split: prod.splits)
  from Seq.prems(2) have "returns_in e c1 ∨ returns_in e c2" by simp
  then show ?case using Seq.IH(1)[OF c1] Seq.IH(2)[OF c2] E by auto
next
  case (If i b c1 c2)
  from If.prems(1) obtain n1 en1 E1 K1 n2 en2 E2 K2 where
    c1: "compile Π p c1 k (Suc n) = (n1, en1, E1, K1)"
    and c2: "compile Π p c2 k n1 = (n2, en2, E2, K2)"
    and E: "E = {(Statement n, EA_Assume b, if is_SKIP c1 then k else en1),
                 (Statement n, EA_AssumeNot b, if is_SKIP c2 then k else en2)}
                 ∪ (if is_SKIP c1 then {} else E1) ∪ (if is_SKIP c2 then {} else E2)"
    by (auto split: prod.splits)
  from If.prems(2) have "returns_in e c1 ∨ returns_in e c2" by simp
  then show ?case using If.IH(1)[OF c1] If.IH(2)[OF c2] E
    by (auto split: if_splits)
next
  case (While i b c)
  from While.prems(1) obtain n1 en1 E1 K1 where
    c1: "compile Π p c (Statement n) (Suc n) = (n1, en1, E1, K1)"
    and E: "E = {(Statement n, EA_Assume b, en1), (Statement n, EA_AssumeNot b, k)} ∪ E1"
    by (auto split: prod.splits)
  from While.prems(2) have "returns_in e c" by simp
  then show ?case using While.IH[OF c1] E by auto
next
  case (Return i e') then show ?case by (auto split: if_splits)
qed auto
```

Public compiler invariants

A `Return` fragment ignores its continuation: it emits its result edge and nothing else, so no node exists to carry a control flow the source does not have.

```
theorem compile_Return_ignores_continuation:
  "compile Π p (Return i e) k n = compile Π p (Return i e) k' n"
  by simp
```

Where each statement index came from in the source

A source position is nowhere in the compiled graph, yet a report has to name one. What follows supplies the missing link in two steps: an enumeration of the statement indices a fragment allocates, ordered the way a front end completes its reductions, and the lemmas pinning that enumeration to the compiler's own allocation.

Statement order in the source

Which `Statement` index `compile` gives each command of a fragment, listed in the order a bottom-up parser completes the commands: a reduction finishes only once its whole right-hand side has been read, so an `if` is listed after both its branches and a loop after its body. Every clause allocates the same index its `compile` counterpart does: a leaf takes `n`; `Seq` hands `c2` the counter `c1` ends at; `gcom.If` and `While` take `n` for the guard and start the body at `Suc n`. A front end that records one position per reduction can pair its `k`th position with this list's `k`th index without reproducing `compile`'s counter arithmetic.

```
fun com_stmt_post_order :: "nat ⇒ 'a acom ⇒ cfg_node list" where
  "com_stmt_post_order n (SKIP i) = [Statement n]"
| "com_stmt_post_order n (Assign i x a) = [Statement n]"
| "com_stmt_post_order n (Check i l c) = [Statement n]"
| "com_stmt_post_order n (Seq c1 c2) =
     com_stmt_post_order n c1 @ com_stmt_post_order (n + csize c1) c2"
| "com_stmt_post_order n (If i b c1 c2) =
     com_stmt_post_order (Suc n) c1 @ com_stmt_post_order (Suc n + csize c1) c2
       @ [Statement n]"
| "com_stmt_post_order n (While i b c) = com_stmt_post_order (Suc n) c @ [Statement n]"
| "com_stmt_post_order n (Call i dst q actuals) = [Statement n]"
| "com_stmt_post_order n (Return i e) = [Statement n]"
| "com_stmt_post_order n (Restore i) = [Statement n]"
| "com_stmt_post_order n (Unwind i) = [Statement n]"
```

The enumeration has one entry per index the fragment allocates: a shorter or longer list would silently misalign every position after it.

```
lemma length_com_stmt_post_order [simp]: "length (com_stmt_post_order n c) = csize c"
  by (induction c arbitrary: n) auto
```

`com_stmt_post_order` answers this for one fragment given the counter it starts at. A whole program needs those starting counters, and they are not a fragment-local fact: `compile_procs` lays the procedures out one after another and `compile_prog` compiles `main` after all of them, whatever order the source wrote them in.

`procs_stmt_next` below is that layout and nothing else — each declared procedure advances the counter past its body and past the one index `compile_proc` reserves for the epilogue. Undeclared names advance it by nothing, matching the clause `compile_procs` skips.

```
fun procs_stmt_next :: "'a ptable ⇒ pname list ⇒ nat ⇒ nat" where
  "procs_stmt_next Π [] n = n"
| "procs_stmt_next Π (p # ps) n =
     (case Π p of
        None ⇒ procs_stmt_next Π ps n
      | Some decl ⇒ procs_stmt_next Π ps (Suc (n + csize (body decl))))"
```

Each definition paired with the statement indices its body owns, in the order a bottom-up parser finishes them. `main` comes last because that is where `compile_prog` compiles it, not because of where it appears in the source — which is exactly why a front end cannot pair a flat position list against the program and must group by definition first.

```
fun defs_stmt_post_order ::
  "'a ptable ⇒ pname list ⇒ nat ⇒ (pname × cfg_node list) list"
where
  "defs_stmt_post_order Π [] n = []"
| "defs_stmt_post_order Π (p # ps) n =
     (case Π p of
        None ⇒ defs_stmt_post_order Π ps n
      | Some decl ⇒
          (p, com_stmt_post_order n (body decl))
            # defs_stmt_post_order Π ps (Suc (n + csize (body decl))))"

definition prog_stmt_post_order :: "imp_prog ⇒ (pname × cfg_node list) list" where
  "prog_stmt_post_order p =
     defs_stmt_post_order (prog_table p) (prog_procs p) 0
       @ [(prog_main_name,
           com_stmt_post_order (procs_stmt_next (prog_table p) (prog_procs p) 0)
             (prog_main p))]"
```

Connectivity of a compiled graph

A compiled fragment's entry reaches its continuation or its procedure result along the structural successor relation. Nothing downstream consumes this: a routed D/G system's own coverage witness is `vars_cover_exec`, decided per node from whether a caller actually published a seed, not from graph reachability. This connectivity

fact is instead what pins the transparent `Restore`/`Unwind` encoding — emitting nothing for those clauses would make them dead ends and force a `source_com` hypothesis onto the whole-program statement.

The fragment-relative forms the compiler inductions use: an edge of a fragment included in the graph.

```
lemma local_reaches_intra_sub:
  "(u, a, v) ∈ E ⟹ E ⊆ intra g ⟹ local_reaches g u v"
  by (auto intro: local_reaches_intra_step)

lemma local_reaches_comb_caller_sub:
  "(c, ca, FunctionEntry q, k) ∈ K ⟹ K ⊆ calls g ⟹ local_reaches g c k"
  by (auto intro: local_reaches_comb_step)
```

Compiled entry reaches its exit or its procedure result

A fragment's entry leads to its continuation or, along an explicit `Return` edge, to the enclosing procedure's `FunctionResult`. Both stay inside the activation, so the statement is about `local_reaches`.

```
lemma compile_reaches:
  assumes "compile Π p c k n = (n', en, E, K)" and "E ⊆ intra g" and "K ⊆ calls g"
  shows "local_reaches g en k ∨ local_reaches g en (FunctionResult p)"
  using assms
proof (induction c arbitrary: k n n' en E K)
  case (Seq c1 c2)
  from Seq.prems(1) obtain n1 E1 K1 n2 E2 K2 where
      c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, Statement n, E1, K1)"
    and c2: "compile Π p c2 k (n + csize c1) = (n2, Statement (n + csize c1), E2, K2)"
    and res: "en = Statement n" "E = E1 ∪ E2" "K = K1 ∪ K2"
    by (rule compile_SeqE)
  have h1: "local_reaches g (Statement n) (Statement (n + csize c1))
          ∨ local_reaches g (Statement n) (FunctionResult p)"
    using Seq.IH(1)[OF c1] Seq.prems res by auto
  have h2: "local_reaches g (Statement (n + csize c1)) k
          ∨ local_reaches g (Statement (n + csize c1)) (FunctionResult p)"
    using Seq.IH(2)[OF c2] Seq.prems res by auto
  from h1 h2 show ?case unfolding res(1) by (blast intro: local_reaches_trans)
next
  case (If i b c1 c2)
  from If.prems(1) obtain n1 E1 K1 E2 K2 where
      c1: "compile Π p c1 k (Suc n) = (n1, Statement (Suc n), E1, K1)"
    and res: "en = Statement n"
    "E = {(Statement n, EA_Assume b, if is_SKIP c1 then k else Statement (Suc n)),
          (Statement n, EA_AssumeNot b,
            if is_SKIP c2 then k else Statement (Suc n + csize c1))}
          ∪ (if is_SKIP c1 then {} else E1) ∪ (if is_SKIP c2 then {} else E2)"
    "K = K1 ∪ K2"
    by (rule compile_IfE)
  have b1: "local_reaches g (Statement n) (if is_SKIP c1 then k else Statement (Suc n))"
    using res(2) If.prems(2) by (blast intro: local_reaches_intra_sub)
  show ?case
  proof (cases "is_SKIP c1")
    case True with b1 res(1) show ?thesis by simp
  next
    case False
    have "local_reaches g (Statement (Suc n)) k
        ∨ local_reaches g (Statement (Suc n)) (FunctionResult p)"
      using If.IH(1)[OF c1] If.prems res False by auto
    with b1 False show ?thesis unfolding res(1) by (auto intro: local_reaches_trans)
  qed
qed (auto simp: Let_def split: prod.splits option.splits
     intro: local_reaches_intra_sub local_reaches_comb_caller_sub)

lemma compile_proc_reaches_result:
  assumes "compile_proc Π p decl n = (n', E, K)"
      and "E ⊆ intra g" "K ⊆ calls g"
  shows "local_reaches g (FunctionEntry p) (FunctionResult p)"
proof -
  let ?r = "n + csize (body decl)"
  obtain Eb where
      body: "compile Π p (body decl) (Statement ?r) n = (?r, Statement n, Eb, K)"
    and E_eq: "E = insert (FunctionEntry p, EA_Body p, Statement n)
                    (insert (Statement ?r, EA_Ret None p, FunctionResult p) Eb)"
    by (rule compile_procE[OF assms(1)])
  have Ebg: "Eb ⊆ intra g" using E_eq assms(2) by auto
  have entry: "local_reaches g (FunctionEntry p) (Statement n)"
    using E_eq assms(2) by (blast intro: local_reaches_intra_sub)
```

```
  have epi: "local_reaches g (Statement ?r) (FunctionResult p)"
    using E_eq assms(2) by (blast intro: local_reaches_intra_sub)
  from compile_reaches[OF body Ebg assms(3)] entry epi show ?thesis
    by (blast intro: local_reaches_trans)
qed

theorem compile_prog_entry_cfg_reaches_exit:
  "cfg_reaches (compile_prog Π ps)
     (cfg_entry (compile_prog Π ps)) (cfg_exit (compile_prog Π ps))"
proof -
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      cmain: "compile_proc Π prog_main_name ⦇formals = [], body = (main_body Π)⦈ n1
                = (n2, Emain, Kmain)"
    and g: "intra (compile_prog Π ps) = Eprocs ∪ Emain"
           "calls (compile_prog Π ps) = Kprocs ∪ Kmain"
    by (rule compile_prog_intra_split)
  have "cfg_reaches (compile_prog Π ps)
          (FunctionEntry prog_main_name) (FunctionResult prog_main_name)"
    using g by (intro local_reaches_cfg_reaches compile_proc_reaches_result[OF cmain]) auto
  then show ?thesis by simp
qed

end
```

5.4 Residual_Location

```
theory Residual_Location
  imports VIMP_Proc_to_CFG
begin
```

Where a partly executed command sits in the graph

Running a command part way leaves a *residual*: the piece still to execute. `control_at Π p c0 k n r v` states that residual `r` of the fragment `c0` — compiled at offset `n` with continuation `k` — corresponds to CFG node `v`. It tracks control only; store agreement is the simulation relation's business.

A command that *reduces to* `SKIP` is located at the continuation `k`, because `compile` takes the continuation as an input rather than allocating an exit node of its own. A literal source `SKIP` is the exception: it keeps its own statement node until its `EA_Nop` edge is taken.

The relation also locates positions no execution reaches, such as the second half of a `Seq` whose first half always returns. That is harmless: the compiled graph has the matching edges there too, since every procedure body ends at an epilogue whose return edge `compile_proc` always emits.

Located residuals

`Restore` and `Unwind` have no clause: they are runtime-only activation markers rather than compiled commands. Once a `Return` fires the CFG is already at `FunctionResult p` through the `EA_Ret` edge, and the activation return that remains is performed by `cstep` popping a frame, not by any fragment node.

```
inductive control_at ::
  "'a ptable ⇒ pname ⇒ 'a acom ⇒ cfg_node ⇒ nat ⇒ 'a acom ⇒ cfg_node ⇒ bool"
  for Π :: "'a ptable" and p :: pname
where
  Skip [intro]:
    "control_at Π p (SKIP i) k n (SKIP i) (Statement n)"
| Assign [intro]:
    "control_at Π p (Assign i x a) k n (Assign i x a) (Statement n)"
| AssignDone [intro]:
    "control_at Π p (Assign i x a) k n (SKIP i) k"
| Check [intro]:
    "control_at Π p (VIMP_Proc.gcom.Check i l c) k n (VIMP_Proc.gcom.Check i l c) (Statement n)"
| CheckDone [intro]:
    "control_at Π p (VIMP_Proc.gcom.Check i l c) k n (SKIP i) k"
| SeqLeft [intro]:
    "control_at Π p c1 (Statement (n + csize c1)) n r v ⟹
     control_at Π p (Seq c1 c2) k n (Seq r c2) v"
| SeqRight [intro]:
    "control_at Π p c2 k (n + csize c1) r v ⟹
     control_at Π p (Seq c1 c2) k n r v"
| IfHead [intro]:
    "control_at Π p (If i b c1 c2) k n (If i b c1 c2) (Statement n)"
| IfLeft [intro]:
    "control_at Π p c1 k (Suc n) r v ⟹ ¬ is_SKIP c1 ⟹
     control_at Π p (If i b c1 c2) k n r v"
| IfRight [intro]:
    "control_at Π p c2 k (Suc n + csize c1) r v ⟹ ¬ is_SKIP c2 ⟹
     control_at Π p (If i b c1 c2) k n r v"
| IfDone [intro]:
    "c1 = SKIP j ∨ c2 = SKIP j ⟹ control_at Π p (If i b c1 c2) k n (SKIP j) k"
| WhileHead [intro]:
    "control_at Π p (While i b c) k n (While i b c) (Statement n)"
| WhileUnfolded [intro]:
    "control_at Π p (While i b c) k n
       (If i b (Seq c (While i b c)) (SKIP i)) (Statement n)"
| WhileBody [intro]:
    "control_at Π p c (Statement n) (Suc n) r v ⟹
     control_at Π p (While i b c) k n (Seq r (While i b c)) v"
| WhileDone [intro]:
    "control_at Π p (While i b c) k n (SKIP i) k"
| CallHead [intro]:
    "control_at Π p (Call i dst q actuals) k n (Call i dst q actuals) (Statement n)"
| CallDone [intro]:
    "control_at Π p (Call i dst q actuals) k n (SKIP i) k"
| ReturnHead [intro]:
    "control_at Π p (Return i e) k n (Return i e) (Statement n)"
```

Inversion by the shape of the fragment `c0`, which is what every consumer knows and cases on. `Seq`, `gcom.If` and

`While` stay plain `[elim]` because their clauses recurse into a sub-fragment, and an eager rule would chase the nesting; the rest are `[elim!]`. `Restore` and `Unwind` have no clause at all, so inverting them is outright refutation — that is what makes `control_at_not_unwind` a one-liner.

```
inductive_cases control_at_SkipE [elim!]:    "control_at Π p (SKIP i) k n r v"
inductive_cases control_at_AssignE [elim!]:  "control_at Π p (Assign i x a) k n r v"
inductive_cases control_at_CheckE [elim!]:
  "control_at Π p (VIMP_Proc.gcom.Check i l b) k n r v"
inductive_cases control_at_SeqE [elim]:      "control_at Π p (Seq c1 c2) k n r v"
inductive_cases control_at_IfE [elim]:       "control_at Π p (If i b c1 c2) k n r v"
inductive_cases control_at_WhileE [elim]:    "control_at Π p (While i b c) k n r v"
inductive_cases control_at_CallE [elim!]:
  "control_at Π p (Call i dst q actuals) k n r v"
inductive_cases control_at_ReturnE [elim!]:  "control_at Π p (Return i e) k n r v"
inductive_cases control_at_RestoreE [elim!]: "control_at Π p (Restore i) k n r v"
inductive_cases control_at_UnwindE [elim!]:  "control_at Π p (Unwind i) k n r v"
```

Initial location

A source command is initially located at its entry node `Statement n`.

```
lemma control_at_initial:
  "source_com c ⟹ control_at Π p c k n c (Statement n)"
  by (induction c arbitrary: k n) auto
```

Normal completion is located at the continuation

A completed residual is at the continuation already, except for a source `SKIP`, whose own node still carries one store-preserving `EA_Nop` edge to it: both branches of a conditional are compiled with the same continuation, so a completed branch is at `k` directly.

```
lemma compile_control_at_SKIP_exit_path:
  "control_at Π p c0 k n (SKIP i) v ⟹ compile Π p c0 k n = (n', en, E, K)
   ⟹ E ⊆ intra g ⟹ intra_path g (v, s) (k, s)"
proof (induction c0 k n "SKIP i :: 'a acom" v arbitrary: n' en E K rule: control_at.induct)
  case (Skip k n)
  then have "(Statement n, EA_Nop, k) ∈ intra g" by auto
  then show ?case by (rule intra_path_nop)
next
  case (SeqRight c2 k n c1 v)
  from SeqRight.prems(1) obtain n2 E2 K2 where
    c2: "compile Π p c2 k (n + csize c1) = (n2, Statement (n + csize c1), E2, K2)"
    and "E2 ⊆ E"
    by (rule compile_SeqE) blast
  with SeqRight.prems(2) show ?case by (intro SeqRight.hyps(2)[OF c2]) blast
next
  case (IfLeft c1 k n v i b c2)
  from IfLeft.prems(1) obtain n1 E1 K1 where
    c1: "compile Π p c1 k (Suc n) = (n1, Statement (Suc n), E1, K1)" and "E1 ⊆ E"
    by (rule compile_If_leftE[OF _ IfLeft.hyps(3)]) blast
  with IfLeft.prems(2) show ?case by (intro IfLeft.hyps(2)[OF c1]) blast
next
  case (IfRight c2 k n c1 v i b)
  from IfRight.prems(1) obtain n2 E2 K2 where
    c2: "compile Π p c2 k (Suc n + csize c1) = (n2, Statement (Suc n + csize c1), E2, K2)"
    and "E2 ⊆ E"
    by (rule compile_If_rightE[OF _ IfRight.hyps(3)]) blast
  with IfRight.prems(2) show ?case by (intro IfRight.hyps(2)[OF c2]) blast
qed simp_all
```

Located residuals remain source commands

Execution never introduces the runtime-only commands into a residual: whatever remains of a source procedure is itself source syntax. The simulation needs this to rule out the frame-popping cases when it is reasoning inside a procedure body.

```
lemma control_at_source_com:
  assumes "control_at Π p c0 k n r v" and "source_com c0"
  shows "source_com r"  using assms
  by (induction rule: control_at.induct) auto

lemma source_com_no_Restore:
  "source_com c ⟹ ¬ is_Restore c"
  by auto

end
```

5.5 Residual_Edges

```
theory Residual_Edges
  imports Residual_Location "Voblint_CFG.CFG_Exec"
begin
```

The next CFG edge of a located residual

If a residual is located at a node and the command about to run is a base one — an assignment, a check, a call, a return — then the edge the compiler emitted for it really is in the graph, and taking the source step re-locates the successor residual at that edge's target. Every simulation step reads its edge off one of these facts.

`intra_step` collects the source steps that stay inside one activation, and `intra_step_simulation` assembles the facts below into the statement that any such step is matched by a run of `cstep`.

Six proofs here induct on `control_at` and repeat the same descent: project the sub-fragment's compile tuple, apply the induction hypothesis, lift the result back through `control_at.SeqRight` / `IfLeft` / `IfRight`. It looks factorable and is not. A shared *descend to a base clause* lemma would have to carry the residual down unchanged, and `SeqLeft` and `WhileBody` rebuild it — they relate the outer residual `Seq r c2` to the inner `r`. So the descent survives only in the three clauses that pass the residual through, which is half the cases in two of the six proofs and none of the rest.

Located base residuals emit their compiled edge

A *base* residual is one the compiler turns into a single intra edge, rather than a branch, a call, or a relocation. `emitted_action` is the action on that edge. An ordinary `Call` has none, because it is compiled to a call edge instead; a `Call` that `special_table` classifies is resolved in place, and `wf_source_com` fixes its destination to `Some x`, so only that shape emits an action.

The annotation of the statement a residual runs next. A step leaves the annotation of the command it reduces on the `SKIP` it produces, so this is what the completed residual carries.

```
fun stmt_ann :: "'a acom ⇒ 'a" where
  "stmt_ann (SKIP i) = i"
| "stmt_ann (Assign i x a) = i"
| "stmt_ann (VIMP_Proc.gcom.Check i l c) = i"
| "stmt_ann (Seq c1 c2) = stmt_ann c1"
| "stmt_ann (If i b c1 c2) = i"
| "stmt_ann (While i b c) = i"
| "stmt_ann (Call i dst q actuals) = i"
| "stmt_ann (Return i e) = i"
| "stmt_ann (Restore i) = i"
| "stmt_ann (Unwind i) = i"

fun emitted_action :: "'a acom ⇒ edge_action option" where
  "emitted_action (Assign i x a) = Some (EA_Assign x a)"
| "emitted_action (VIMP_Proc.gcom.Check i l c) = Some (EA_Check l c)"
| "emitted_action (Call i (Some x) q actuals) =
     (case special_table q of
        None ⇒ None
      | Some desc ⇒ map_option (λsc. EA_Special sc x) (classify_special desc actuals))"
| "emitted_action _ = None"
```

Which action it is does not matter to the recursive `control_at` cases: they only carry an edge up through the enclosing fragment and re-locate the completed residual at the same node. So one induction serves every base shape.

```
lemma control_at_emitted_edge:
  "control_at Π p c0 k n r v ⟹ emitted_action r = Some a ⟹
   compile Π p c0 k n = (n', en, E, K) ⟹
   ∃j w. v = Statement j ∧ (Statement j, a, w) ∈ E
       ∧ control_at Π p c0 k n (SKIP (stmt_ann r)) w"
proof (induction arbitrary: n' en E K rule: control_at.induct)
  case (Assign i x' a' k n0)
  then show ?case by auto
next
  case (Check i l b' k n0)
  then show ?case by auto
next
  case (CallHead i dst' q' actuals' k n0)
  show ?case
  proof (cases dst')
    case None with CallHead.prems(1) show ?thesis by simp
```

```
  next
    case (Some x)
    with CallHead.prems(1) obtain desc sc where
      sp: "special_table q' = Some desc" and cl: "classify_special desc actuals' = Some sc"
        and act: "a = EA_Special sc x"
      by (auto split: option.splits)
    have "(Statement n0, EA_Special sc x, k) ∈ E"
      using CallHead.prems(2) sp cl Some by auto
    with act show ?thesis by auto
  qed
next
  case (SeqRight c2 k n0 c1 r v)
  from SeqRight.prems(2) obtain n2 E2 K2 where
    c2c: "compile Π p c2 k (n0 + csize c1) = (n2, Statement (n0 + csize c1), E2, K2)"
    and sub: "E2 ⊆ E"
    by (rule compile_Seq_rightE)
  from SeqRight.IH[OF SeqRight.prems(1) c2c] obtain j w where
    jw: "v = Statement j" "(Statement j, a, w) ∈ E2"
        "control_at Π p c2 k (n0 + csize c1) (SKIP (stmt_ann r)) w" by blast
  have "control_at Π p (Seq c1 c2) k n0 (SKIP (stmt_ann r)) w"
    using control_at.SeqRight[OF jw(3)] .
  then show ?case using jw sub by blast
next
  case (IfLeft c1 k n0 r v i b c2)
  from IfLeft.prems(2) obtain n1 E1 K1 where
    c1c: "compile Π p c1 k (Suc n0) = (n1, Statement (Suc n0), E1, K1)"
    and sub: "E1 ⊆ E"
    by (rule compile_If_leftE[OF _ IfLeft.hyps(2)])
  from IfLeft.IH[OF IfLeft.prems(1) c1c] obtain j w where
    jw: "v = Statement j" "(Statement j, a, w) ∈ E1"
        "control_at Π p c1 k (Suc n0) (SKIP (stmt_ann r)) w" by blast
  have "control_at Π p (If i b c1 c2) k n0 (SKIP (stmt_ann r)) w"
    using control_at.IfLeft[OF jw(3) IfLeft.hyps(2)] .
  then show ?case using jw sub by blast
next
  case (IfRight c2 k n0 c1 r v i b)
  from IfRight.prems(2) obtain n2 E2 K2 where
    c2c: "compile Π p c2 k (Suc n0 + csize c1)
            = (n2, Statement (Suc n0 + csize c1), E2, K2)"
    and sub: "E2 ⊆ E"
    by (rule compile_If_rightE[OF _ IfRight.hyps(2)])
  from IfRight.IH[OF IfRight.prems(1) c2c] obtain j w where
    jw: "v = Statement j" "(Statement j, a, w) ∈ E2"
        "control_at Π p c2 k (Suc n0 + csize c1) (SKIP (stmt_ann r)) w" by blast
  have "control_at Π p (If i b c1 c2) k n0 (SKIP (stmt_ann r)) w"
    using control_at.IfRight[OF jw(3) IfRight.hyps(2)] .
  then show ?case using jw sub by blast
qed simp_all
```

The check a residual runs next, when its leftmost command is one. A located residual about to run a check sits at that check's `EA_Check` edge, however much sequencing wraps it.

```
fun next_check :: "'a acom ⇒ (check_label × exp) option" where
  "next_check (VIMP_Proc.gcom.Check i l c) = Some (l, c)"
| "next_check (Seq c1 c2) = next_check c1"
| "next_check _ = None"

lemma control_at_next_check_edge:
  "control_at Π p c0 k n r v ⟹ next_check r = Some (l, c) ⟹
   compile Π p c0 k n = (n', en, E, K) ⟹ ∃w. (v, EA_Check l c, w) ∈ E"
proof (induction arbitrary: n' en E K rule: control_at.induct)
  case (SeqLeft c1 n0 r v c2 k)
  from SeqLeft.prems(2) obtain m F L
    where "compile Π p c1 (Statement (n0 + csize c1)) n0 = (m, Statement n0, F, L)"
      and "F ⊆ E"
    by (rule compile_Seq_leftE)
  with SeqLeft.IH SeqLeft.prems(1) show ?case by auto
next
  case (SeqRight c2 k n0 c1 r v)
  from SeqRight.prems(2) obtain m F L
    where "compile Π p c2 k (n0 + csize c1) = (m, Statement (n0 + csize c1), F, L)"
      and "F ⊆ E"
    by (rule compile_Seq_rightE)
  with SeqRight.IH SeqRight.prems(1) show ?case by blast
next
  case (IfLeft c1 k n0 r v i b c2)
  from IfLeft.prems(2) obtain m F L
```

```
    where "compile Π p c1 k (Suc n0) = (m, Statement (Suc n0), F, L)" and "F ⊆ E"
    by (rule compile_If_leftE[OF _ IfLeft.hyps(2)])
  with IfLeft.IH IfLeft.prems(1) show ?case by blast
next
  case (IfRight c2 k n0 c1 r v i b)
  from IfRight.prems(2) obtain m F L
    where "compile Π p c2 k (Suc n0 + csize c1) = (m, Statement (Suc n0 + csize c1), F, L)"
      and "F ⊆ E"
    by (rule compile_If_rightE[OF _ IfRight.hyps(2)])
  with IfRight.IH IfRight.prems(1) show ?case by blast
next
  case (WhileBody c n0 r v i b k)
  from WhileBody.prems(2) obtain m F L
    where "compile Π p c (Statement n0) (Suc n0) = (m, Statement (Suc n0), F, L)"
      and "F ⊆ E"
    by (rule compile_While_bodyE)
  with WhileBody.IH WhileBody.prems(1) show ?case by auto
qed auto
```

A located conditional also arises from a loop unfolding (`While` steps to `If b (Seq c (While b c)) SKIP`), so the `WhileUnfolded` case is real: the true branch re-locates the loop body followed by the loop, the false branch re-locates `SKIP` at the loop's continuation.

```
lemma control_at_if_edges:
  "control_at Π p c0 k n r v ⟹ r = If i b c1 c2 ⟹ source_com r ⟹
   compile Π p c0 k n = (n', en, E, K) ⟹
   ∃j en1 en2. v = Statement j
       ∧ (Statement j, EA_Assume b, en1) ∈ E ∧ (Statement j, EA_AssumeNot b, en2) ∈ E
       ∧ control_at Π p c0 k n c1 en1 ∧ control_at Π p c0 k n c2 en2"
proof (induction arbitrary: n' en E K rule: control_at.induct)
  case (IfHead i b' c1' c2' k n0)
  from IfHead.prems(1) have r: "b' = b" "c1' = c1" "c2' = c2" by auto
  note E = compile_If_assume_edges[OF IfHead.prems(3)]
  from IfHead.prems(2) r have src: "source_com c1'" "source_com c2'" by auto
  have ca1: "control_at Π p (If i b' c1' c2') k n0 c1'
      (if is_SKIP c1' then k else Statement (Suc n0))"
    using src(1) by (cases "is_SKIP c1'") (auto intro: control_at_initial)
  have ca2: "control_at Π p (If i b' c1' c2') k n0 c2'
      (if is_SKIP c2' then k else Statement (Suc n0 + csize c1'))"
  proof (cases "is_SKIP c2'")
    case True then show ?thesis by auto
  next
    case False
    have "control_at Π p (If i b' c1' c2') k n0 c2'
        (Statement (Suc n0 + csize c1'))"
      by (rule control_at.IfRight[OF control_at_initial[OF src(2)] False])
    with False show ?thesis by simp
  qed
  from E ca1 ca2 r show ?case by auto
next
  case (SeqRight c2' k n0 c1' r v)
  from SeqRight.prems(3) obtain n2 E2 K2 where
    c2c: "compile Π p c2' k (n0 + csize c1') = (n2, Statement (n0 + csize c1'), E2, K2)"
    and sub: "E2 ⊆ E"
    by (rule compile_Seq_rightE)
  from SeqRight.IH[OF SeqRight.prems(1) SeqRight.prems(2) c2c] obtain j e1 e2 where
    jw: "v = Statement j" "(Statement j, EA_Assume b, e1) ∈ E2"
        "(Statement j, EA_AssumeNot b, e2) ∈ E2"
        "control_at Π p c2' k (n0 + csize c1') c1 e1"
        "control_at Π p c2' k (n0 + csize c1') c2 e2" by blast
  have "control_at Π p (Seq c1' c2') k n0 c1 e1" "control_at Π p (Seq c1' c2') k n0 c2 e2"
    using control_at.SeqRight[OF jw(4)]
          control_at.SeqRight[OF jw(5)] .
  then show ?case using jw sub by blast
next
  case (IfLeft c1' k n0 r v i b' c2')
  from IfLeft.prems(3) obtain n1 E1 K1 where
    c1c: "compile Π p c1' k (Suc n0) = (n1, Statement (Suc n0), E1, K1)"
    and sub: "E1 ⊆ E"
    by (rule compile_If_leftE[OF _ IfLeft.hyps(2)])
  from IfLeft.IH[OF IfLeft.prems(1) IfLeft.prems(2) c1c] obtain j e1 e2 where
    jw: "v = Statement j" "(Statement j, EA_Assume b, e1) ∈ E1"
        "(Statement j, EA_AssumeNot b, e2) ∈ E1"
        "control_at Π p c1' k (Suc n0) c1 e1" "control_at Π p c1' k (Suc n0) c2 e2" by blast
  have "control_at Π p (If i b' c1' c2') k n0 c1 e1" "control_at Π p (If i b' c1' c2') k n0 c2 e2"
    using control_at.IfLeft[OF jw(4) IfLeft.hyps(2)] control_at.IfLeft[OF jw(5) IfLeft.hyps(2)] .
  then show ?case using jw sub by blast
```

```
next
  case (IfRight c2' k n0 c1' r v i b')
  from IfRight.prems(3) obtain n2 E2 K2 where
    c2c: "compile Π p c2' k (Suc n0 + csize c1')
            = (n2, Statement (Suc n0 + csize c1'), E2, K2)"
    and sub: "E2 ⊆ E"
    by (rule compile_If_rightE[OF _ IfRight.hyps(2)])
  from IfRight.IH[OF IfRight.prems(1) IfRight.prems(2) c2c] obtain j e1 e2 where
    jw: "v = Statement j" "(Statement j, EA_Assume b, e1) ∈ E2"
        "(Statement j, EA_AssumeNot b, e2) ∈ E2"
        "control_at Π p c2' k (Suc n0 + csize c1') c1 e1"
        "control_at Π p c2' k (Suc n0 + csize c1') c2 e2" by blast
  have "control_at Π p (If i b' c1' c2') k n0 c1 e1" "control_at Π p (If i b' c1' c2') k n0 c2 e2"
    using control_at.IfRight[OF jw(4) IfRight.hyps(2)] control_at.IfRight[OF jw(5) IfRight.hyps(2)]
    .
  then show ?case using jw sub by blast
next
  case (WhileUnfolded i b'' c'' k n0)
  from WhileUnfolded.prems(1)
  have r: "b'' = b" "Seq c'' (While i b'' c'') = c1" "(SKIP i) = c2" by auto
  note E = compile_While_assume_edges[OF WhileUnfolded.prems(3)]
  from WhileUnfolded.prems(2) have src: "source_com c''" by auto
  have ca1: "control_at Π p (While i b'' c'') k n0 (Seq c'' (While i b'' c'')) (Statement (Suc n0))"
    by (rule control_at.WhileBody[OF control_at_initial[OF src]])
  have ca2: "control_at Π p (While i b'' c'') k n0 (SKIP i) k"
    by (rule control_at.WhileDone)
  from E ca1 ca2 r show ?case by auto
qed simp_all
```

SKIP relocation to the continuation

A located `SKIP` — a completed sub-command — is at its fragment's continuation already, except for a source `SKIP`, which still has its own node and one `EA_Nop` edge. `compile_control_at_SKIP_exit_path` already establishes that route as a store-preserving `intra_path`; an intra path never touches the activation stack, so `intra_path_imp_cstep_star` reads it back as a `cstep` run at whatever stack is current.

```
lemma control_at_skip_to_exit:
  assumes "control_at Π p c0 k n r v" and "is_SKIP r"
      and "compile Π p c0 k n = (n', en, E, K)" and "E ⊆ intra g"
  shows "𝒢, g ⊢ (v, s, stk) →c* (k, s, stk)"
proof -
  from assms(2) obtain j where r: "r = SKIP j" by auto
  show ?thesis
    using intra_path_imp_cstep_star[
            OF compile_control_at_SKIP_exit_path[OF assms(1)[unfolded r] assms(3,4)],
            where stk = stk]
    by simp
qed
```

Completed-head sequence relocation

Under continuation passing the head's own last edge already targets the entry of `c2` — or, when the continuation is the enclosing `While`, the loop head — so relocation is exactly `control_at_skip_to_exit`.

```
lemma control_at_seq_skip_reloc:
  "control_at Π p c0 k n r v ⟹ r = Seq (SKIP i) c2 ⟹
   compile Π p c0 k n = (n', en, E, K) ⟹ E ⊆ intra g ⟹ source_com c0 ⟹
   ∃v'. control_at Π p c0 k n c2 v' ∧ 𝒢, g ⊢ (v, s, stk) →c* (v', s, stk)"
proof (induction arbitrary: n' en E K rule: control_at.induct)
  case (SeqLeft c1 n0 r_in v c2r k)
  from SeqLeft.prems(1) have ri: "is_SKIP r_in" and c2eq: "c2r = c2" by auto
  from SeqLeft.prems(2) obtain n1 E1 K1 where
    c1c: "compile Π p c1 (Statement (n0 + csize c1)) n0 = (n1, Statement n0, E1, K1)"
    and sub: "E1 ⊆ E"
    by (rule compile_Seq_leftE)
  have src2: "source_com c2r" using SeqLeft.prems(4) by simp
  from control_at_skip_to_exit[OF SeqLeft.hyps(1) ri c1c subset_trans[OF sub SeqLeft.prems(3)]]
  have sk: "𝒢, g ⊢ (v, s, stk) →c* (Statement (n0 + csize c1), s, stk)" .
  have "control_at Π p (Seq c1 c2r) k n0 c2r (Statement (n0 + csize c1))"
    by (rule control_at.SeqRight[OF control_at_initial[OF src2]])
  then have "control_at Π p (Seq c1 c2r) k n0 c2 (Statement (n0 + csize c1))" using c2eq by simp
  with sk show ?case by blast
next
  case (WhileBody c n0 r_in v i b k)
  from WhileBody.prems(1) have ri: "is_SKIP r_in" and c2eq: "c2 = While i b c" by auto
```

```
  from WhileBody.prems(2) obtain n1 E1 K1 where
    cc: "compile Π p c (Statement n0) (Suc n0) = (n1, Statement (Suc n0), E1, K1)"
    and sub: "E1 ⊆ E"
    by (rule compile_While_bodyE)
  from control_at_skip_to_exit[OF WhileBody.hyps(1) ri cc subset_trans[OF sub WhileBody.prems(3)]]
  have sk: "𝒢, g ⊢ (v, s, stk) →c* (Statement n0, s, stk)" .
  have "control_at Π p (While i b c) k n0 (While i b c) (Statement n0)"
    by (rule control_at.WhileHead)
  then have "control_at Π p (While i b c) k n0 c2 (Statement n0)" using c2eq by simp
  with sk show ?case by blast
next
  case (SeqRight c2' k n0 c1 r v)
  from SeqRight.prems(2) obtain n2 E2 K2 where
    c2c: "compile Π p c2' k (n0 + csize c1) = (n2, Statement (n0 + csize c1), E2, K2)"
    and sub: "E2 ⊆ E"
    by (rule compile_Seq_rightE)
  have src2: "source_com c2'" using SeqRight.prems(4) by simp
  from SeqRight.IH[OF SeqRight.prems(1) c2c subset_trans[OF sub SeqRight.prems(3)] src2]
  obtain v' where v': "control_at Π p c2' k (n0 + csize c1) c2 v'"
    "𝒢, g ⊢ (v, s, stk) →c* (v', s, stk)" by blast
  have "control_at Π p (Seq c1 c2') k n0 c2 v'"
    using control_at.SeqRight[OF v'(1)] .
  with v'(2) show ?case by blast
next
  case (IfLeft c1 k n0 r v i b c2')
  from IfLeft.prems(2) obtain n1 E1 K1 where
    c1c: "compile Π p c1 k (Suc n0) = (n1, Statement (Suc n0), E1, K1)"
    and sub: "E1 ⊆ E"
    by (rule compile_If_leftE[OF _ IfLeft.hyps(2)])
  have src1: "source_com c1" using IfLeft.prems(4) by simp
  from IfLeft.IH[OF IfLeft.prems(1) c1c subset_trans[OF sub IfLeft.prems(3)] src1]
  obtain v' where v': "control_at Π p c1 k (Suc n0) c2 v'"
    "𝒢, g ⊢ (v, s, stk) →c* (v', s, stk)" by blast
  have "control_at Π p (If i b c1 c2') k n0 c2 v'" using control_at.IfLeft[OF v'(1) IfLeft.hyps(2)]
    .
  with v'(2) show ?case by blast
next
  case (IfRight c2' k n0 c1 r v i b)
  from IfRight.prems(2) obtain n2 E2 K2 where
    c2c: "compile Π p c2' k (Suc n0 + csize c1)
            = (n2, Statement (Suc n0 + csize c1), E2, K2)"
    and sub: "E2 ⊆ E"
    by (rule compile_If_rightE[OF _ IfRight.hyps(2)])
  have src2: "source_com c2'" using IfRight.prems(4) by simp
  from IfRight.IH[OF IfRight.prems(1) c2c subset_trans[OF sub IfRight.prems(3)] src2]
  obtain v' where v': "control_at Π p c2' k (Suc n0 + csize c1) c2 v'"
    "𝒢, g ⊢ (v, s, stk) →c* (v', s, stk)" by blast
  have "control_at Π p (If i b c1 c2') k n0 c2 v'"
    using control_at.IfRight[OF v'(1) IfRight.hyps(2)]
    .
  with v'(2) show ?case by blast
qed simp_all
```

The caller continuation after a call

A call site's caller has, after the call, a finite left-spine of residual continuations `afters`: the source commands sequenced after the call, in execution order, collected while descending the activation's left-associated `Seq` spine. `seq_after c afters` wraps them around the active head `c`: `seq_after c [s3, s4] = Seq (Seq c s3) s4`. A tail-position call has `afters = []` and the caller resumes at bare `c`. The list holds pending commands *inside one activation*; suspended *caller activations* are the recursive nesting of `csim` instead.

```
fun seq_after :: "'a acom ⇒ 'a acom list ⇒ 'a acom" where
  "seq_after c [] = c"
| "seq_after c (a # as) = seq_after (Seq c a) as"

lemma seq_after_append: "seq_after (seq_after c xs) ys = seq_after c (xs @ ys)"
  by (induction xs arbitrary: c) auto

lemma seq_after_snoc: "seq_after c (xs @ [a]) = Seq (seq_after c xs) a"
  by (simp add: seq_after_append[symmetric])

lemma next_check_seq_after [simp]: "next_check (seq_after c afters) = next_check c"
  by (induction afters arbitrary: c) auto
```

A `seq_after` spine equals a non-`Seq` atom only when its head does and the continuation list is empty. These

discriminators close the leaf cases of the spine inductions.

```
lemma seq_after_eq_iff [simp]:
  "(seq_after c afters = SKIP i) = (c = SKIP i ∧ afters = [])"
  "(SKIP i = seq_after c afters) = (c = SKIP i ∧ afters = [])"
  "(seq_after c afters = Restore i) = (c = Restore i ∧ afters = [])"
  "(Restore i = seq_after c afters) = (c = Restore i ∧ afters = [])"
  "(seq_after c afters = Unwind i) = (c = Unwind i ∧ afters = [])"
  "(Unwind i = seq_after c afters) = (c = Unwind i ∧ afters = [])"
  "(seq_after c afters = Assign i x a) = (c = Assign i x a ∧ afters = [])"
  "(Assign i x a = seq_after c afters) = (c = Assign i x a ∧ afters = [])"
  "(seq_after c afters = VIMP_Proc.gcom.Check i l b) = (c = VIMP_Proc.gcom.Check i l b ∧ afters = [])"
  "(VIMP_Proc.gcom.Check i l b = seq_after c afters) = (c = VIMP_Proc.gcom.Check i l b ∧ afters = [])"
  "(seq_after c afters = If i b c1 c2) = (c = If i b c1 c2 ∧ afters = [])"
  "(If i b c1 c2 = seq_after c afters) = (c = If i b c1 c2 ∧ afters = [])"
  "(seq_after c afters = While i b cw) = (c = While i b cw ∧ afters = [])"
  "(While i b cw = seq_after c afters) = (c = While i b cw ∧ afters = [])"
  "(seq_after c afters = Return i e) = (c = Return i e ∧ afters = [])"
  "(Return i e = seq_after c afters) = (c = Return i e ∧ afters = [])"
  "(seq_after c afters = Call i dst q actuals) = (c = Call i dst q actuals ∧ afters = [])"
  "(Call i dst q actuals = seq_after c afters) = (c = Call i dst q actuals ∧ afters = [])"
  by (induction afters arbitrary: c; auto)+

lemma source_com_seq_afterD [dest]:
  "source_com (seq_after c afters) ⟹
   source_com c ∧ (∀a ∈ set afters. source_com a)"
proof (induction afters arbitrary: c)
  case Nil
  then show ?case by simp
next
  case (Cons a afters)
  have h: "source_com (seq_after (Seq c a) afters)"
    using Cons.prems by simp
  from Cons.IH[OF h] have h1: "source_com (Seq c a)"
    and h2: "∀b ∈ set afters. source_com b" by blast+
  from h1 h2 show ?case by auto
qed

lemma control_at_head_return_afters_no_Restore:
  assumes "control_at Π p c0 k n (seq_after (Return i e) afters) v"
      and "source_com c0"
  shows "∀a ∈ set afters. ¬ is_Restore a"
proof (rule ballI)
  fix a
  assume ain: "a ∈ set afters"
  from control_at_source_com[OF assms] source_com_seq_afterD
  have "source_com a" using ain by blast
  then show "¬ is_Restore a" by (rule source_com_no_Restore)
qed
```

A `Call` at the bottom-left of a `seq_after` spine still sits on the call edge the compiler emitted at the call node, and the source step re-locates `seq_after (SKIP i) afters` at the call's continuation node. The `SeqLeft` / `WhileBody` cases peel the outermost continuation, the passthrough cases keep it.

```
lemma control_at_seq_after_call_edge:
  "control_at Π p c0 k n r v ⟹ r = seq_after (Call i dst q actuals) afters ⟹
   compile Π p c0 k n = (n', en, E, K) ⟹ special_table q = None ⟹
   ∃j w. v = Statement j
       ∧ (Statement j, CallEdge dst (call_formals Π q) actuals,
          FunctionEntry q, w) ∈ K
       ∧ control_at Π p c0 k n (seq_after (SKIP i) afters) w"
proof (induction arbitrary: afters n' en E K rule: control_at.induct)
  case (CallHead ia dst' q' actuals' k n0 afters)
  then show ?case by (auto simp: Let_def split: prod.splits)
next
  case (SeqLeft c1 n0 r v c2 k afters)
  obtain xs where afx: "afters = xs @ [c2]" and req: "r = seq_after (Call i dst q actuals) xs"
    using SeqLeft.prems(1) by (cases afters rule: rev_cases) (auto simp: seq_after_snoc)
  from SeqLeft.prems(2) obtain n1 E1 K1 where
    c1c: "compile Π p c1 (Statement (n0 + csize c1)) n0 = (n1, Statement n0, E1, K1)"
    and sub: "K1 ⊆ K"
    by (rule compile_Seq_leftE)
  from SeqLeft.IH[OF req c1c SeqLeft.prems(3)] obtain j w where
    jw: "v = Statement j"
       "(Statement j, CallEdge dst (call_formals Π q) actuals,
         FunctionEntry q, w) ∈ K1"
       "control_at Π p c1 (Statement (n0 + csize c1)) n0 (seq_after (SKIP i) xs) w" by blast
```

```
  have "control_at Π p (Seq c1 c2) k n0 (seq_after (SKIP i) afters) w"
    using control_at.SeqLeft[OF jw(3)] by (simp add: afx seq_after_snoc)
  then show ?case using jw sub by blast
next
  case (SeqRight c2 k n0 c1 r v afters)
  from SeqRight.prems(2) obtain n2 E2 K2 where
    c2c: "compile Π p c2 k (n0 + csize c1) = (n2, Statement (n0 + csize c1), E2, K2)"
    and sub: "K2 ⊆ K"
    by (rule compile_Seq_rightE)
  from SeqRight.IH[OF SeqRight.prems(1) c2c SeqRight.prems(3)] obtain j w where
    jw: "v = Statement j"
       "(Statement j, CallEdge dst (call_formals Π q) actuals,
         FunctionEntry q, w) ∈ K2"
       "control_at Π p c2 k (n0 + csize c1) (seq_after (SKIP i) afters) w" by blast
  have "control_at Π p (Seq c1 c2) k n0 (seq_after (SKIP i) afters) w"
    using control_at.SeqRight[OF jw(3)] .
  then show ?case using jw sub by blast
next
  case (IfLeft c1 k n0 r v ia b c2 afters)
  from IfLeft.prems(2) obtain n1 E1 K1 where
    c1c: "compile Π p c1 k (Suc n0) = (n1, Statement (Suc n0), E1, K1)"
    and sub: "K1 ⊆ K"
    by (rule compile_If_leftE[OF _ IfLeft.hyps(2)])
  from IfLeft.IH[OF IfLeft.prems(1) c1c IfLeft.prems(3)] obtain j w where
    jw: "v = Statement j"
       "(Statement j, CallEdge dst (call_formals Π q) actuals,
         FunctionEntry q, w) ∈ K1"
       "control_at Π p c1 k (Suc n0) (seq_after (SKIP i) afters) w" by blast
  have "control_at Π p (If ia b c1 c2) k n0 (seq_after (SKIP i) afters) w"
    using control_at.IfLeft[OF jw(3) IfLeft.hyps(2)] .
  then show ?case using jw sub by blast
next
  case (IfRight c2 k n0 c1 r v ia b afters)
  from IfRight.prems(2) obtain n2 E2 K2 where
    c2c: "compile Π p c2 k (Suc n0 + csize c1)
            = (n2, Statement (Suc n0 + csize c1), E2, K2)"
    and sub: "K2 ⊆ K"
    by (rule compile_If_rightE[OF _ IfRight.hyps(2)])
  from IfRight.IH[OF IfRight.prems(1) c2c IfRight.prems(3)] obtain j w where
    jw: "v = Statement j"
       "(Statement j, CallEdge dst (call_formals Π q) actuals,
         FunctionEntry q, w) ∈ K2"
       "control_at Π p c2 k (Suc n0 + csize c1) (seq_after (SKIP i) afters) w" by blast
  have "control_at Π p (If ia b c1 c2) k n0 (seq_after (SKIP i) afters) w"
    using control_at.IfRight[OF jw(3) IfRight.hyps(2)] .
  then show ?case using jw sub by blast
next
  case (WhileBody c n0 r v ia b k afters)
  obtain xs where afx: "afters = xs @ [While ia b c]"
      and req: "r = seq_after (Call i dst q actuals) xs"
    using WhileBody.prems(1) by (cases afters rule: rev_cases) (auto simp: seq_after_snoc)
  from WhileBody.prems(2) obtain n1 E1 K1 where
    cc: "compile Π p c (Statement n0) (Suc n0) = (n1, Statement (Suc n0), E1, K1)"
    and sub: "K1 ⊆ K"
    by (rule compile_While_bodyE)
  from WhileBody.IH[OF req cc WhileBody.prems(3)] obtain j w where
    jw: "v = Statement j"
       "(Statement j, CallEdge dst (call_formals Π q) actuals,
         FunctionEntry q, w) ∈ K1"
       "control_at Π p c (Statement n0) (Suc n0) (seq_after (SKIP i) xs) w" by blast
  have "control_at Π p (While ia b c) k n0 (seq_after (SKIP i) afters) w"
    using control_at.WhileBody[OF jw(3)] by (simp add: afx seq_after_snoc)
  then show ?case using jw sub by blast
qed simp_all

lemma control_at_seq_after_return_edge:
  "control_at Π p c0 k n r v ⟹ r = seq_after (Return i e) afters ⟹
   compile Π p c0 k n = (n', en, E, K) ⟹
   ∃j. v = Statement j ∧ (Statement j, EA_Ret e p, FunctionResult p) ∈ E"
proof (induction arbitrary: afters n' en E K rule: control_at.induct)
  case (ReturnHead i e' k n0 afters)
  then show ?case by auto
next
  case (SeqLeft c1 n0 r v c2 k afters)
  obtain xs where afx: "afters = xs @ [c2]" and req: "r = seq_after (Return i e) xs"
    using SeqLeft.prems(1) by (cases afters rule: rev_cases) (auto simp: seq_after_snoc)
```

```
  from SeqLeft.prems(2) obtain n1 E1 K1 where
    c1c: "compile Π p c1 (Statement (n0 + csize c1)) n0 = (n1, Statement n0, E1, K1)"
    and sub: "E1 ⊆ E"
    by (rule compile_Seq_leftE)
  from SeqLeft.IH[OF req c1c] show ?case using sub by blast
next
  case (SeqRight c2 k n0 c1 r v afters)
  from SeqRight.prems(2) obtain n2 E2 K2 where
    c2c: "compile Π p c2 k (n0 + csize c1) = (n2, Statement (n0 + csize c1), E2, K2)"
    and sub: "E2 ⊆ E"
    by (rule compile_Seq_rightE)
  from SeqRight.IH[OF SeqRight.prems(1) c2c] show ?case using sub by blast
next
  case (IfLeft c1 k n0 r v i b c2 afters)
  from IfLeft.prems(2) obtain n1 E1 K1 where
    c1c: "compile Π p c1 k (Suc n0) = (n1, Statement (Suc n0), E1, K1)"
    and sub: "E1 ⊆ E"
    by (rule compile_If_leftE[OF _ IfLeft.hyps(2)])
  from IfLeft.IH[OF IfLeft.prems(1) c1c] show ?case using sub by blast
next
  case (IfRight c2 k n0 c1 r v i b afters)
  from IfRight.prems(2) obtain n2 E2 K2 where
    c2c: "compile Π p c2 k (Suc n0 + csize c1)
            = (n2, Statement (Suc n0 + csize c1), E2, K2)"
    and sub: "E2 ⊆ E"
    by (rule compile_If_rightE[OF _ IfRight.hyps(2)])
  from IfRight.IH[OF IfRight.prems(1) c2c] show ?case using sub by blast
next
  case (WhileBody c n0 r v ia b k afters)
  obtain xs where afx: "afters = xs @ [While ia b c]" and req: "r = seq_after (Return i e) xs"
    using WhileBody.prems(1) by (cases afters rule: rev_cases) (auto simp: seq_after_snoc)
  from WhileBody.prems(2) obtain n1 E1 K1 where
    cc: "compile Π p c (Statement n0) (Suc n0) = (n1, Statement (Suc n0), E1, K1)"
    and sub: "E1 ⊆ E"
    by (rule compile_While_bodyE)
  from WhileBody.IH[OF req cc] show ?case using sub by blast
qed simp_all

lemma control_at_return_edge:
  "control_at Π p c0 k n r v ⟹ r = Return i e ⟹
   compile Π p c0 k n = (n', en, E, K) ⟹
   ∃j. v = Statement j ∧ (Statement j, EA_Ret e p, FunctionResult p) ∈ E"
  by (rule control_at_seq_after_return_edge[where afters = "[]"]) simp_all
```

The intra-procedural source steps

`intra_step` is the fragment of `pstep` that stays inside one activation without initiating a return: assignment, sequencing (head execution and head completion), both conditionals, and the loop unfolding. It excludes `Call` (pushes a frame), `Return` (produces `Unwind`), and the runtime-only `Restore` / `Unwind` steps. Every intra step preserves the frame stack and keeps the residual source-shaped.

```
inductive intra_step ::
  "'a ptable ⇒ 'a acom × store × frame list ⇒ 'a acom × store × frame list ⇒ bool" for Π where
  IAssign [intro]: "intra_step Π (Assign i x a, s, frs) (SKIP i, s(x := ⟦a⟧e s), frs)"
| ISpecial [intro]: "special_table q = Some desc ⟹ classify_special desc actuals = Some sc ⟹
             special_result sc s v ⟹
             intra_step Π (Call i (Some x) q actuals, s, frs) (SKIP i, s(x := v), frs)"
| ICheck [intro]:  "intra_step Π (VIMP_Proc.gcom.Check i l b, s, frs) (SKIP i, s, frs)"
| ISeq1 [intro]:   "intra_step Π (Seq (SKIP i) c2, s, frs) (c2, s, frs)"
| ISeq2 [intro]:   "intra_step Π (c1, s, frs) (c1', s', frs) ⟹
           intra_step Π (Seq c1 c2, s, frs) (Seq c1' c2, s', frs)"
| IIfTrue [intro]: "truthy (⟦b⟧e s) ⟹ intra_step Π (If i b c1 c2, s, frs) (c1, s, frs)"
| IIfFalse [intro]:"¬ truthy (⟦b⟧e s) ⟹ intra_step Π (If i b c1 c2, s, frs) (c2, s, frs)"
| IWhile [intro]:
  "intra_step Π (While i b c, s, frs) (If i b (Seq c (While i b c)) (SKIP i), s, frs)"


inductive_cases intra_SkipE [elim!]:   "intra_step Π (SKIP i, s, frs) y"
inductive_cases intra_AssignE [elim!]: "intra_step Π (Assign i x a, s, frs) y"
inductive_cases intra_CheckE [elim!]:  "intra_step Π (VIMP_Proc.gcom.Check i l b, s, frs) y"
inductive_cases intra_SeqE [elim]:    "intra_step Π (Seq c1 c2, s, frs) y"
inductive_cases intra_IfE [elim!]:     "intra_step Π (If i b c1 c2, s, frs) y"
inductive_cases intra_WhileE [elim]:  "intra_step Π (While i b c, s, frs) y"
inductive_cases intra_CallE [elim]:   "intra_step Π (Call i dst q actuals, s, frs) y"
inductive_cases intra_ReturnE [elim!]: "intra_step Π (Return i e, s, frs) y"
```

```
lemma intra_Seq_cases:
  "intra_step Π (Seq c1 c2, s, frs) (c', s', frs') ⟹
   (is_SKIP c1 ∧ c' = c2 ∧ s' = s ∧ frs' = frs) ∨
   (∃c1'. c' = Seq c1' c2 ∧ frs' = frs ∧ intra_step Π (c1, s, frs) (c1', s', frs))"
  by auto

lemma intra_If_cases:
  "intra_step Π (If i b c1 c2, s, frs) (c', s', frs') ⟹
   (truthy (⟦b⟧e s) ∧ c' = c1 ∧ s' = s ∧ frs' = frs) ∨
   (¬ truthy (⟦b⟧e s) ∧ c' = c2 ∧ s' = s ∧ frs' = frs)"
  by auto

lemma intra_step_any_frame:
  "intra_step Π (c, s, frs) (c', s', frs') ⟹ intra_step Π (c, s, fr) (c', s', fr)"
  by (induction "(c, s, frs)" "(c', s', frs')" arbitrary: c s frs c' s' frs'
      rule: intra_step.induct) auto

lemma intra_step_frame_eq:
  "intra_step Π (c, s, frs) (c', s', frs') ⟹ frs' = frs"
  by (induction "(c, s, frs)" "(c', s', frs')" arbitrary: c s frs c' s' frs'
      rule: intra_step.induct) auto
```

Intra-step simulation

The base cases of that simulation, once: a located residual with an emitted action moves along its own edge in one `cstep` and re-locates as a completed sub-command. Which action it is only decides the store, which the caller supplies as `s' ∈ edge_step a s`.

```
lemma control_at_base_step:
  assumes loc: "control_at Π p c0 k n r v"
      and act: "emitted_action r = Some a"
      and cmp: "compile Π p c0 k n = (n', en, E, K)"
      and sub: "E ⊆ intra g"
      and stp: "s' ∈ edge_step a s"
  obtains w where
    "control_at Π p c0 k n (SKIP (stmt_ann r)) w"
    "𝒢, g ⊢ (v, s, stk) →c* (w, s', stk)"
proof -
  from control_at_emitted_edge[OF loc act cmp] obtain j w where
    jw: "v = Statement j" "(Statement j, a, w) ∈ E"
        "control_at Π p c0 k n (SKIP (stmt_ann r)) w" by blast
  have "(Statement j, a, w) ∈ intra g" using jw(2) sub by blast
  from cstep.Intra[OF this stp] have "𝒢, g ⊢ (Statement j, s, stk) →c (w, s', stk)" .
  then show ?thesis using jw(1) jw(3) that by auto
qed
```

A located conditional takes whichever assume edge the store selects and lands on the branch located at that edge's target. `gcom.If` heads and loop unfoldings both reach it.

```
lemma control_at_if_step:
  assumes loc: "control_at Π p c0 k n (If i b c1 c2) v"
      and stp: "intra_step Π (If i b c1 c2, s, frs) (c', s', frs')"
      and cmp: "compile Π p c0 k n = (n', en, E, K)"
      and sub: "E ⊆ intra g"
      and src: "source_com (If i b c1 c2)"
  shows "frs' = frs
         ∧ (∃v'. control_at Π p c0 k n c' v' ∧ 𝒢, g ⊢ (v, s, stk) →c* (v', s', stk))"
proof -
  from control_at_if_edges[OF loc refl src cmp] obtain j en1 en2 where
    jj: "v = Statement j"
       "(Statement j, EA_Assume b, en1) ∈ E" "(Statement j, EA_AssumeNot b, en2) ∈ E"
       "control_at Π p c0 k n c1 en1" "control_at Π p c0 k n c2 en2" by blast
  from intra_If_cases[OF stp] show ?thesis
  proof (elim disjE conjE)
    assume t: "truthy (⟦b⟧e s)" "c' = c1" "s' = s" "frs' = frs"
    have "(Statement j, EA_Assume b, en1) ∈ intra g" using jj(2) sub by blast
    from star_step1[of "cstep 𝒢 g", OF cstep_assume[OF this t(1)]]
    show ?thesis using t jj(1,4) by blast
  next
    assume f: "¬ truthy (⟦b⟧e s)" "c' = c2" "s' = s" "frs' = frs"
    have "(Statement j, EA_AssumeNot b, en2) ∈ intra g" using jj(3) sub by blast
    from star_step1[of "cstep 𝒢 g", OF cstep_assume_not[OF this f(1)]]
    show ?thesis using f jj(1,5) by blast
  qed
qed
```

A located `Seq r c2` either has finished its head, and relocates to `c2`, or steps inside `r`. Only the caller knows which sub-fragment `r` is located in, so it supplies that second case.

```
lemma control_at_seq_step:
  assumes loc: "control_at Π p c0 k n (Seq r c2) v"
      and stp: "intra_step Π (Seq r c2, s, frs) (c', s', frs')"
      and cmp: "compile Π p c0 k n = (n', en, E, K)"
      and sub: "E ⊆ intra g"
      and src: "source_com c0"
      and head: "⋀r'. intra_step Π (r, s, frs) (r', s', frs) ⟹
                  ∃v'. control_at Π p c0 k n (Seq r' c2) v'
                       ∧ 𝒢, g ⊢ (v, s, stk) →c* (v', s', stk)"
  shows "frs' = frs
         ∧ (∃v'. control_at Π p c0 k n c' v' ∧ 𝒢, g ⊢ (v, s, stk) →c* (v', s', stk))"
  using intra_Seq_cases[OF stp]
proof (elim disjE exE conjE)
  assume r: "is_SKIP r" and out: "c' = c2" "s' = s" "frs' = frs"
  then obtain j where rj: "r = SKIP j" by auto
  from control_at_seq_skip_reloc[OF loc[unfolded rj] refl cmp sub src] show ?thesis
    using out by blast
next
  fix r' assume "c' = Seq r' c2" "frs' = frs" "intra_step Π (r, s, frs) (r', s', frs)"
  with head show ?thesis by blast
qed
```

The theory's conclusion: an `intra_step` of a located residual leaves the frame stack alone, and the graph follows it to a node at which the successor residual is located again. This is the fact the simulation relation is preserved by.

```
theorem intra_step_simulation:
  "control_at Π p c0 k n c v ⟹
   intra_step Π (c, s, frs) (c', s', frs') ⟹
   compile Π p c0 k n = (n', en, E, K) ⟹ E ⊆ intra g ⟹ source_com c0 ⟹
   frs' = frs
   ∧ (∃v'. control_at Π p c0 k n c' v'
        ∧ 𝒢, g ⊢ (v, s, stk) →c* (v', s', stk))"
proof (induction arbitrary: c' s' frs' n' en E K rule: control_at.induct)
  case (Skip i k n0) then show ?case by blast
next
  case (Assign i x a k n0)
  from Assign.prems(1) have out: "c' = (SKIP i)" "s' = s(x := ⟦a⟧e s)" "frs' = frs"
    by auto
  have act: "emitted_action (Assign i x a) = Some (EA_Assign x a)" by simp
  have mem: "s' ∈ edge_step (EA_Assign x a) s" using out(2) by simp
  obtain w where "control_at Π p (Assign i x a) k n0 (SKIP i) w"
      and "𝒢, g ⊢ (Statement n0, s, stk) →c* (w, s', stk)"
    by (rule control_at_base_step[OF control_at.Assign act Assign.prems(2)
                                     Assign.prems(3) mem]) auto
  then show ?case using out(1,3) by auto
next
  case (AssignDone i x a k n0) then show ?case by blast
next
  case (Check i l b k n0)
  from Check.prems(1) have out: "c' = (SKIP i)" "s' = s" "frs' = frs"
    by auto
  have act: "emitted_action (VIMP_Proc.gcom.Check i l b) = Some (EA_Check l b)" by simp
  have mem: "s' ∈ edge_step (EA_Check l b) s" using out(2) by simp
  obtain w where "control_at Π p (VIMP_Proc.gcom.Check i l b) k n0 (SKIP i) w"
      and "𝒢, g ⊢ (Statement n0, s, stk) →c* (w, s', stk)"
    by (rule control_at_base_step[OF control_at.Check act Check.prems(2)
                                     Check.prems(3) mem]) auto
  then show ?case using out(1,3) by auto
next
  case (CheckDone i l b k n0) then show ?case by blast
next
  case (SeqLeft c1 n0 r v c2 k)
  from SeqLeft.prems(2) obtain n1 E1 K1 where
    c1c: "compile Π p c1 (Statement (n0 + csize c1)) n0 = (n1, Statement n0, E1, K1)"
    and sub: "E1 ⊆ E"
    by (rule compile_Seq_leftE)
  have src1: "source_com c1" using SeqLeft.prems(4) by simp
  show ?case
    by (rule control_at_seq_step[OF control_at.SeqLeft[OF SeqLeft.hyps] SeqLeft.prems])
       (use SeqLeft.IH[OF _ c1c subset_trans[OF sub SeqLeft.prems(3)] src1] in blast)
next
  case (SeqRight c2 k n0 c1 r v)
  from SeqRight.prems(2) obtain n2 E2 K2 where
```

```
    c2c: "compile Π p c2 k (n0 + csize c1) = (n2, Statement (n0 + csize c1), E2, K2)"
    and sub: "E2 ⊆ E"
    by (rule compile_Seq_rightE)
  have src2: "source_com c2" using SeqRight.prems(4) by simp
  from SeqRight.IH[OF SeqRight.prems(1) c2c subset_trans[OF sub SeqRight.prems(3)] src2]
  have fr: "frs' = frs" and
    "∃v'. control_at Π p c2 k (n0 + csize c1) c' v'
          ∧ 𝒢, g ⊢ (v, s, stk) →c* (v', s', stk)" by auto
  then obtain v' where v': "control_at Π p c2 k (n0 + csize c1) c' v'"
    "𝒢, g ⊢ (v, s, stk) →c* (v', s', stk)" by blast
  have "control_at Π p (Seq c1 c2) k n0 c' v'"
    using control_at.SeqRight[OF v'(1)] .
  with fr v'(2) show ?case by blast
next
  case (IfHead i b c1 c2 k n0)
  show ?case by (rule control_at_if_step[OF control_at.IfHead IfHead.prems])
next
  case (IfLeft c1 k n0 r v i b c2)
  from IfLeft.prems(2) obtain n1 E1 K1 where
    c1c: "compile Π p c1 k (Suc n0) = (n1, Statement (Suc n0), E1, K1)"
    and sub1: "E1 ⊆ E"
    by (rule compile_If_leftE[OF _ IfLeft.hyps(2)])
  have src1: "source_com c1" using IfLeft.prems(4) by simp
  from IfLeft.IH[OF IfLeft.prems(1) c1c subset_trans[OF sub1 IfLeft.prems(3)] src1]
  have fr: "frs' = frs" and
    "∃v'. control_at Π p c1 k (Suc n0) c' v'
          ∧ 𝒢, g ⊢ (v, s, stk) →c* (v', s', stk)" by auto
  then obtain v' where v': "control_at Π p c1 k (Suc n0) c' v'"
    "𝒢, g ⊢ (v, s, stk) →c* (v', s', stk)" by blast
  have "control_at Π p (If i b c1 c2) k n0 c' v'" using control_at.IfLeft[OF v'(1) IfLeft.hyps(2)] .
  with fr v'(2) show ?case by blast
next
  case (IfRight c2 k n0 c1 r v i b)
  from IfRight.prems(2) obtain n2 E2 K2 where
    c2c: "compile Π p c2 k (Suc n0 + csize c1)
            = (n2, Statement (Suc n0 + csize c1), E2, K2)"
    and sub2: "E2 ⊆ E"
    by (rule compile_If_rightE[OF _ IfRight.hyps(2)])
  have src2: "source_com c2" using IfRight.prems(4) by simp
  from IfRight.IH[OF IfRight.prems(1) c2c subset_trans[OF sub2 IfRight.prems(3)] src2]
  have fr: "frs' = frs" and
    "∃v'. control_at Π p c2 k (Suc n0 + csize c1) c' v'
          ∧ 𝒢, g ⊢ (v, s, stk) →c* (v', s', stk)" by auto
  then obtain v' where v': "control_at Π p c2 k (Suc n0 + csize c1) c' v'"
    "𝒢, g ⊢ (v, s, stk) →c* (v', s', stk)" by blast
  have "control_at Π p (If i b c1 c2) k n0 c' v'" using control_at.IfRight[OF v'(1) IfRight.hyps(2)]
    .
  with fr v'(2) show ?case by blast
next
  case (IfDone i b c1 j c2 k n0) then show ?case by blast
next
  case (WhileHead i b cW k n0)
  from WhileHead.prems(1)
  have out: "c' = If i b (Seq cW (While i b cW)) (SKIP i)" "s' = s" "frs' = frs"
    by auto
  have "control_at Π p (While i b cW) k n0 (If i b (Seq cW (While i b cW)) (SKIP i)) (Statement n0)"
    by (rule control_at.WhileUnfolded)
  then show ?case using out by auto
next
  case (WhileUnfolded i b cW k n0)
  have srcif: "source_com (If i b (Seq cW (While i b cW)) (SKIP i))" using WhileUnfolded.prems(4)
    by simp
  show ?case
    by (rule control_at_if_step[OF control_at.WhileUnfolded WhileUnfolded.prems(1-3) srcif])
next
  case (WhileBody cW n0 r v i b k)
  from WhileBody.prems(2) obtain n1 E1 K1 where
    cc: "compile Π p cW (Statement n0) (Suc n0) = (n1, Statement (Suc n0), E1, K1)"
    and sub: "E1 ⊆ E"
    by (rule compile_While_bodyE)
  have srcW: "source_com cW" using WhileBody.prems(4) by simp
  show ?case
    by (rule control_at_seq_step[OF control_at.WhileBody[OF WhileBody.hyps] WhileBody.prems])
       (use WhileBody.IH[OF _ cc subset_trans[OF sub WhileBody.prems(3)] srcW] in blast)
next
  case (WhileDone i b cW k n0) then show ?case by blast
```

```
next
  case (CallHead i dst q actuals k n0)
  from CallHead.prems(1) obtain x desc sc v where
    out: "dst = Some x" "special_table q = Some desc" "classify_special desc actuals = Some sc"
         "special_result sc s v" "c' = (SKIP i)" "s' = s(x := v)" "frs' = frs"
    by auto
  have act: "emitted_action (Call i (Some x) q actuals) = Some (EA_Special sc x)"
    using out(2,3) by simp
  have mem: "s' ∈ edge_step (EA_Special sc x) s" using out(4,6) by auto
  obtain w where "control_at Π p (Call i (Some x) q actuals) k n0 (SKIP i) w"
      and "𝒢, g ⊢ (Statement n0, s, stk) →c* (w, s', stk)"
    by (rule control_at_base_step[OF control_at.CallHead act
              CallHead.prems(2)[unfolded out(1)] CallHead.prems(3) mem]) auto
  then show ?case using out(1,5,7) by auto
next
  case (CallDone i dst q actuals k n0) then show ?case by blast
next
  case (ReturnHead i e k n0) then show ?case by blast
qed

end
```

5.6 Simulation_Relation

```
theory Simulation_Relation
  imports Residual_Edges
begin
```

What it means for a source state and a graph state to agree

`csim` relates a source configuration `(c, s, frs)` to a CFG configuration `(v, s, stk)` holding the very same store. The two represent suspended callers differently: the source nests each caller's continuation inside the single command `c`, as `Seq (Seq inner (Restore i)) after` wrappings, while the CFG keeps them as continuation nodes on its frame stack. `csim` pairs the two off, one layer per suspended caller.

Both nestings run outermost-first, so the top command layer pairs with the last frame. After `main` calls `f` calls `g` the source command is `Seq (Seq (Seq (Seq C Restore) B) Restore) A` — `A` is main's continuation and `C` the running residual of `g` — while the CFG sits at `g`'s node with continuation frames `[B's node, A's node]`.

Three constructors, one per phase an activation can be in. `Base` relates a single activation with nothing beneath it. `Nested` peels one suspended caller. `Returning` covers the skew just after a callee finishes: the CFG has already reached `FunctionResult p`, while the source is still propagating `Restore` or `Unwind` towards its frame-pop rule. Proving that execution preserves `csim` is the next theory's business; this one defines the relation and the vocabulary it is stated in.

Return-store update

The store published by a return, common to the source `Return` step and the compiled `EA_Ret` edge.

```
definition ret_store :: "exp option ⇒ store ⇒ store" where
  "ret_store e s = s(ret_var := (case e of None ⇒ s ret_var | Some a ⇒ ⟦a⟧e s))"

lemma ret_store_None [simp]: "ret_store None s = s"
  by (simp add: ret_store_def)

lemma ret_store_Some [simp]: "ret_store (Some e) s = s(ret_var := ⟦e⟧e s)"
  by (simp add: ret_store_def)

lemma edge_step_EA_Ret_ret_store_mem [simp]: "ret_store e s ∈ edge_step (EA_Ret e p) s"
  by (simp add: ret_store_def)
```

Bare Unwind is stuck

A lone `Unwind` has no source step: only the `Seq` rules propagate it, and the frame pop it is heading for fires on `Seq (Unwind j) (Restore i)`.

```
lemma pstep_Unwind_stuck: "¬ 𝒢, Π ⊢ (Unwind i, s, frs) →p x"
  by auto
```

Why a top-level `Return` must not reach the source: an empty frame stack has nowhere to unwind to, so a run that starts at `Unwind` or `Return` never completes.

```
lemma Unwind_not_pcompletes: "¬ pcompletes 𝒢 Π (Unwind ()) s t"
proof (rule notI)
  assume "𝒢, Π ⊢ (Unwind (), s, []) →p* (SKIP (), t, [])"
  then show False
    by (cases rule: star.cases) (auto simp: pstep_Unwind_stuck)
qed

lemma Return_empty_not_pcompletes: "¬ pcompletes 𝒢 Π (Return () e) s t"
proof (rule notI)
  assume "𝒢, Π ⊢ (Return () e, s, []) →p* (SKIP (), t, [])"
  then obtain y where step: "𝒢, Π ⊢ (Return () e, s, []) →p y"
      and rest: "𝒢, Π ⊢ y →p* (SKIP (), t, [])"
    by (cases rule: star.cases) auto
  from step have "y = (Unwind (), ret_store e s, [])"
    by (cases e) (auto simp: ret_store_def)
  with rest have "𝒢, Π ⊢ (Unwind (), ret_store e s, []) →p* (SKIP (), t, [])" by simp
  then show False
    by (cases rule: star.cases) (auto simp: pstep_Unwind_stuck)
qed
```

The returning (frame-pop) source shapes

Once a callee finishes, the CFG is already at `FunctionResult p` while the source propagates toward the frame-pop rule. `unwinding` describes an `Unwind` travelling up through the callee's residual, dropping the dead code

it skips (never the activation's own `Restore`); `pop_ready` describes the whole activation body in that phase — either the normal fall-through `Restore` or an `Unwind` spine `Seq u Restore` from an explicit `Return`. Both pop the single caller frame with one `cstep` return; `unwinding` propagation is matched by *no* CFG step, because the CFG waits at `FunctionResult p`.

```
fun unwinding :: "'a acom ⇒ bool" where
  "unwinding (Unwind i) = True"
| "unwinding (Seq u c2) = (unwinding u ∧ ¬ is_Restore c2)"
| "unwinding _ = False"

fun pop_ready :: "'a acom ⇒ bool" where
  "pop_ready (Restore i) = True"
| "pop_ready (Seq u (Restore i)) = unwinding u"
| "pop_ready _ = False"
```

The annotation of the `Restore` a returning activation pops: the call it came from, which is also the annotation of the `SKIP` the pop leaves behind.

```
fun pop_ann :: "'a acom ⇒ 'a option" where
  "pop_ann (Restore i) = Some i"
| "pop_ann (Seq u (Restore i)) = Some i"
| "pop_ann _ = None"

lemma unwinding_not_SKIP: "unwinding u ⟹ ¬ is_SKIP u"
  by (cases u) auto

lemma pop_ready_not_SKIP: "pop_ready w ⟹ ¬ is_SKIP w"
  by (cases w rule: pop_ready.cases) auto

lemma pop_ready_not_Unwind: "pop_ready w ⟹ w ≠ Unwind i"
  by (cases w rule: pop_ready.cases) auto
```

Compile-into-g evidence for an active fragment

`compiled_at Π g p body_p k n` is an activation's certificate. It says two things at once, because every activation needs both: `body_p` is genuinely procedure `p`'s body, and that body, compiled at the exact offset `n` with continuation `k`, is embedded in the target graph `g` — its intra edges lie in `intra g` and its call edges in `calls g`.

The offset matters because compilation is shift-parametric: neither `n` nor `k` is fixed by the syntax, so graph membership *at the chosen offset* is the missing fact and this predicate supplies it, at the same `k` and `n` that `control_at` uses. The procedure identity matters because `control_at`'s fragment parameter is otherwise free, and fall-through preservation is false for a fragment that is not a whole body.

A procedure body's continuation is its epilogue node, which is why the fall-through return edge is stated from `k`. `compile_proc` emits that edge for every body.

The offset itself is anchored to the graph: the body enters at the target of `p`'s `EA_Body` edge, and `k` is the epilogue right after the body. Without the anchor the body's edges could be matched at a shifted offset, and the relation would locate a residual at a node no execution of `p` uses.

```
definition compiled_at ::
  "'a ptable ⇒ cfg ⇒ pname ⇒ 'a acom ⇒ cfg_node ⇒ nat ⇒ bool" where
  "compiled_at Π g p body_p k n ⟷
     (∃decl n' en E K. Π p = Some decl ∧ body_p = body decl
        ∧ compile Π p body_p k n = (n', en, E, K)
        ∧ E ⊆ intra g ∧ K ⊆ calls g
        ∧ (k, EA_Ret None p, FunctionResult p) ∈ intra g
        ∧ (FunctionEntry p, EA_Body p, Statement n) ∈ intra g
        ∧ k = Statement (n + csize body_p))"

lemma compiled_atI [intro]:
  "Π p = Some decl ⟹ body_p = body decl ⟹
   compile Π p body_p k n = (n', en, E, K) ⟹ E ⊆ intra g ⟹ K ⊆ calls g ⟹
   (k, EA_Ret None p, FunctionResult p) ∈ intra g ⟹
   (FunctionEntry p, EA_Body p, Statement n) ∈ intra g ⟹ k = Statement (n + csize body_p) ⟹
   compiled_at Π g p body_p k n"
  unfolding compiled_at_def by blast

lemma compiled_atE [elim]:
  assumes "compiled_at Π g p body_p k n"
  obtains decl n' en E K where
    "Π p = Some decl" "body_p = body decl"
    "compile Π p body_p k n = (n', en, E, K)" "E ⊆ intra g" "K ⊆ calls g"
    "(k, EA_Ret None p, FunctionResult p) ∈ intra g"
  using assms unfolding compiled_at_def by blast
```

The procedure-identity half on its own, for the many proofs that only need the declaration behind an activation.

```
lemma compiled_at_decl:
  assumes "compiled_at Π g p body_p k n"
  obtains decl where "Π p = Some decl" "body_p = body decl"
  using assms unfolding compiled_at_def by blast

lemma compiled_at_exit:
  "compiled_at Π g p body_p k n ⟹ (k, EA_Ret None p, FunctionResult p) ∈ intra g"
  unfolding compiled_at_def by auto
```

The simulation relation

What it means for a source configuration and a graph configuration to be at the same place. Three shapes, one per stack situation: a single activation, a call in progress with the caller recorded on both stacks, and the moment between a return and the frame pop. All three carry the same store, so the relation constrains position and stack only.

The index `p` in `Π, g, p ⊢ cfg ≈ dg` is the procedure of the outermost activation of `cfg`, the one with no caller below it in the relation. `Nested` and `Returning` record the call that suspended that activation, `Call i dst q args`: the activation inside `Seq inner (Restore i)` runs `q`, and a returning activation is at `FunctionResult q` of that same `q`. Recursion is harmless because the index belongs to an activation, not to a body.

```
inductive csim :: "'a ptable ⇒ cfg ⇒ pname ⇒ 'a acom × store × frame list
                   ⇒ cfg_node × store × cframe list ⇒ bool"
    ("(_,_,_ ⊢/ _ ≈/ _)" [51, 51, 51, 51, 51] 50) for Π g where
  Base:
    "control_at Π p body_p k n c v ⟹ compiled_at Π g p body_p k n ⟹
     Π, g, p ⊢ (c, s, []) ≈ (v, s, [])"
| Nested:
    "Π, g, q ⊢ (inner, s, frs) ≈ (v, s, stk) ⟹
     control_at Π pc body_pc kc nc (seq_after (SKIP i) afters) cont ⟹
     control_at Π pc body_pc kc nc (seq_after (Call i dst q args) afters) site ⟹
     compiled_at Π g pc body_pc kc nc ⟹
     Π, g, pc ⊢ (seq_after (Seq inner (Restore i)) afters, s, frs @ [Frame caller dst])
       ≈ (v, s, stk @ [(cont, dst, caller)])"
| Returning:
    "pop_ready w ⟹ pop_ann w = Some i ⟹
     control_at Π pc body_pc kc nc (seq_after (SKIP i) afters) cont ⟹
     control_at Π pc body_pc kc nc (seq_after (Call i dst q args) afters) site ⟹
     compiled_at Π g pc body_pc kc nc ⟹
     Π, g, pc ⊢ (seq_after w afters, callee, [Frame caller dst])
       ≈ (FunctionResult q, callee, [(cont, dst, caller)])"
```

On its own `csim` is a *structural* correspondence, not a reachability one. For an unannotated program it is not functional either: a residual carries no record of which occurrence it came from, so two textually equal statements share every location. A caller that needs the semantic node pairs it with membership in `node_collect` at the same node. Annotated programs whose annotations are the compiler's own locations are the case where the node is determined.

Inversion at an empty source frame stack. Only `Base` can produce one — `Nested` and `Returning` both append a caller frame — so the CFG stack is empty too, the stores agree, and the whole activation certificate comes back. `csim`'s introduction rules stay undeclared: choosing between `Base`, `Nested` and `Returning` is the step each preservation proof exists to make, and it belongs in the proof text rather than in a search.

```
inductive_cases csim_empty_stackE [elim]: "Π, g, p ⊢ (c, s, []) ≈ (v, t, stk)"
```

`Nested` and `Returning` both locate the *resumed caller* — the ordinary residual `seq_after (SKIP i) afters` that the return will land in — at the continuation node `cont`. `Returning` carries exactly the one frame the return pops; outer callers are consumed by the `Nested` layers wrapping it, and after the pop the caller resumes as a `Base` activation with that wrapping untouched.

```
lemma csim_store_eq:
  "Π, g, ϱ ⊢ (c, s, frs) ≈ (v, t, stk) ⟹ s = t"
  by (induction "(c, s, frs)" "(v, t, stk)" arbitrary: c frs v stk rule: csim.induct) auto
```

Returning commands and the frame-pop phase

`is_returning c` holds when the leftmost-innermost of `c` is `Restore` or `Unwind`: the shape of every returning-phase command, ordinary or `Nested`-wrapped.

```
fun is_returning :: "'a acom ⇒ bool" where
  "is_returning (Restore i) = True"
```

```
| "is_returning (Unwind i) = True"
| "is_returning (Seq c1 c2) = is_returning c1"
| "is_returning _ = False"

lemma is_returning_seq_after [simp]:
  "is_returning (seq_after c afters) = is_returning c"
  by (induction afters arbitrary: c) auto
```

The call-redex source shape

`head_call c` holds when the leftmost-innermost of `c` is a `Call` to a procedure `special_table` does not classify: the shape of an active residual whose next `pstep` pushes a frame. A call `special_table` classifies resolves in place instead (`intra_step`'s `ISpecial`), so it is deliberately excluded here, mirroring `intra_step`'s own exclusion of `Call`.

```
fun head_call :: "'a acom ⇒ bool" where
  "head_call (Call i dst q actuals) = (special_table q = None)"
| "head_call (Seq c1 c2) = head_call c1"
| "head_call _ = False"

lemma head_call_seq_after [simp]:
  "head_call (seq_after c afters) = head_call c"
  by (induction afters arbitrary: c) auto

lemma head_call_not_SKIP: "head_call c ⟹ ¬ is_SKIP c"
  by (cases c) auto

lemma head_call_not_Unwind: "head_call c ⟹ c ≠ Unwind i"
  by (cases c) auto

lemma head_call_seq_after_form:
  "head_call c ⟹ ∃i dst q actuals afters.
     c = seq_after (Call i dst q actuals) afters ∧ special_table q = None"
proof (induction c)
  case (Seq c1 c2)
  from Seq.prems have "head_call c1" by simp
  from Seq.IH(1)[OF this] obtain i dst q actuals afters where
    "c1 = seq_after (Call i dst q actuals) afters" "special_table q = None" by blast
  then have "Seq c1 c2 = seq_after (Call i dst q actuals) (afters @ [c2])"
            "special_table q = None"
    by (simp_all add: seq_after_snoc)
  then show ?case by blast
qed auto

lemma unwinding_not_head_call: "unwinding u ⟹ ¬ head_call u"
  by (induction u rule: unwinding.induct) auto

lemma pop_ready_not_head_call: "pop_ready w ⟹ ¬ head_call w"
  by (cases w rule: pop_ready.cases) (auto simp: unwinding_not_head_call)
```

The return-initiation source shape

`head_return c` holds when the leftmost-innermost of `c` is a source `Return`: the shape whose next `pstep` initiates a return.

```
fun head_return :: "'a acom ⇒ bool" where
  "head_return (Return i e) = True"
| "head_return (Seq c1 c2) = head_return c1"
| "head_return _ = False"

lemma head_return_seq_after [simp]:
  "head_return (seq_after c afters) = head_return c"
  by (induction afters arbitrary: c) auto

lemma head_return_not_SKIP: "head_return c ⟹ ¬ is_SKIP c"
  by (cases c) auto

lemma head_return_seq_after_form:
  "head_return c ⟹ ∃i e afters. c = seq_after (Return i e) afters"
proof (induction c)
  case (Seq c1 c2)
  from Seq.prems have "head_return c1" by simp
  from Seq.IH(1)[OF this] obtain i e afters where "c1 = seq_after (Return i e) afters" by blast
  then have "Seq c1 c2 = seq_after (Return i e) (afters @ [c2])" by (simp add: seq_after_snoc)
  then show ?case by blast
```

```
qed auto

lemma unwinding_not_head_return: "unwinding u ⟹ ¬ head_return u"
  by (induction u rule: unwinding.induct) auto

lemma pop_ready_not_head_return: "pop_ready w ⟹ ¬ head_return w"
  by (cases w rule: pop_ready.cases) (auto simp: unwinding_not_head_return)
```

Source well-formedness: the base-activation return guard

`ret_guarded allow_return c` is the deep source-only well-formedness check: an explicit `Return` (or an in-flight `Unwind`) is admissible only where `allow_return` holds — inside a called activation. Permission is *granted by the activation tail*: the right child `Restore` of a `Seq` marks a callee body, so its left child is checked with `allow_return = True`, whereas ordinary continuations inherit the ambient permission. A bare `Restore` (the transient tail after the callee body reduces to `SKIP`) is always admissible. Crucially this looks *behind* `Restore` boundaries: a `Return` hidden in a base-activation continuation is rejected, which a head-only check misses — exactly the configuration a frame pop would later surface at the base.

```
fun ret_guarded :: "bool ⇒ 'a acom ⇒ bool" where
  "ret_guarded allow_return (SKIP i) = True"
| "ret_guarded allow_return (Assign i x a) = True"
| "ret_guarded allow_return (VIMP_Proc.gcom.Check i l b) = True"
| "ret_guarded allow_return (Seq c1 c2) =
     (if is_Restore c2 then ret_guarded True c1
      else ret_guarded allow_return c1 ∧ ret_guarded allow_return c2)"
| "ret_guarded allow_return (If i b c1 c2) =
     (ret_guarded allow_return c1 ∧ ret_guarded allow_return c2)"
| "ret_guarded allow_return (While i b c) = ret_guarded allow_return c"
| "ret_guarded allow_return (Call i dst p actuals) = True"
| "ret_guarded allow_return (Return i e) = allow_return"
| "ret_guarded allow_return (Restore i) = True"
| "ret_guarded allow_return (Unwind i) = allow_return"

lemma ret_guarded_True_source: "source_com c ⟹ ret_guarded True c"
  by (induction c) (auto split: if_splits)

lemma ret_guarded_False_source_not_head_return:
  "source_com c ⟹ ret_guarded False c ⟹ ¬ head_return c"
  by (induction c) (auto split: if_splits)

lemma ret_guarded_pstep:
  assumes bodies: "⋀p decl. Π p = Some decl ⟹ source_com (body decl)"
      and step: "𝒢, Π ⊢ (c, s, frs) →p (c', s', frs')"
      and "ret_guarded allow_return c"
  shows "ret_guarded allow_return c'"
  using step assms(3)
proof (induction "(c, s, frs)" "(c', s', frs')"
       arbitrary: c s frs c' s' frs' allow_return rule: pstep.induct)
  case Call
  then show ?case using bodies ret_guarded_True_source by auto
next
  case (Seq2 c1 s1 f1 c1' s1' f1' c2)
  then show ?case by (auto split: if_splits)
qed (auto split: if_splits)
```

The runtime invariant `csim_step` asks for: the active command may not return until a call has opened an activation for it. Only the command matters — neither the store nor the frame stack — and unlike a head-only guard this survives a `pstep`, because it already forbids the base-activation returns that a frame pop would expose.

```
definition return_safe :: "'a acom ⇒ bool" where
  "return_safe c ⟷ ret_guarded False c"

lemma return_safe_if_no_return:
  assumes "source_com c" and "no_return c"
  shows "return_safe c"
  using assms
  unfolding return_safe_def
  by (induction c) (auto split: if_splits)

lemma return_safe_not_head_return:
  "return_safe c ⟹ source_com c ⟹ ¬ head_return c"
  by (auto simp: return_safe_def ret_guarded_False_source_not_head_return)

lemma return_safe_pstep:
```

```
  assumes bodies: "⋀p decl. Π p = Some decl ⟹ source_com (body decl)"
      and step: "𝒢, Π ⊢ (c, s, frs) →p (c', s', frs')"
      and wf: "return_safe c"
  shows "return_safe c'"
  using ret_guarded_pstep[OF bodies step] wf by (simp add: return_safe_def)

lemma unwinding_seq_after_Unwind:
  "(∀a ∈ set afters. ¬ is_Restore a) ⟹ unwinding (seq_after (Unwind i) afters)"
  by (induction afters rule: rev_induct) (auto simp: seq_after_snoc)
```

Source-step classification

Sorting a source step into the three cases the simulation treats separately: an ordinary intra step, a call, or a return in progress. The classification is what lets each preservation proof assume it is in exactly one case rather than re-deriving the exclusions from `pstep` every time.

```
lemma intra_step_not_returning:
  "intra_step Π (c, s, frs) (c', s', frs') ⟹ ¬ is_returning c"
  by (induction "(c, s, frs)" "(c', s', frs')" arbitrary: c s frs c' s' frs'
      rule: intra_step.induct) auto
```

A declared procedure's name is never one `special_table` also classifies: the same disjointness `procs_embedded` bundles in the next theory, supplied directly since this lemma sits upstream of that definition. Without it the `pstep.Call` case cannot rule out `c` also matching `pstep.Special`, which `head_call` excludes.

```
lemma pstep_intra_classify:
  assumes disj: "⋀p decl. Π p = Some decl ⟹ special_table p = None"
  shows "𝒢, Π ⊢ (c, s, frs) →p (c', s', frs') ⟹ ¬ head_call c ⟹ ¬ head_return c ⟹
   ¬ is_returning c ⟹ intra_step Π (c, s, frs) (c', s', frs')"
proof (induction "(c, s, frs)" "(c', s', frs')"
       arbitrary: c s frs c' s' frs' rule: pstep.induct)
  case (Seq2 c1 s1 f1 c1' s1' f1' c2)
  from Seq2.prems have "¬ head_call c1" "¬ head_return c1" "¬ is_returning c1" by auto
  from Seq2.hyps(2)[OF this] have ih: "intra_step Π (c1, s1, f1) (c1', s1', f1')" .
  from intra_step_frame_eq[OF ih] ih show ?case by auto
next
  case (Call i p decl actuals dst vals callee s1 frs0)
  then show ?case using disj by simp
qed auto

lemma control_at_not_returning:
  "control_at Π p c0 k n r v ⟹ ¬ is_returning r"
  by (induction rule: control_at.induct) auto

lemma control_at_not_unwind:
  "control_at Π p c0 k n r v ⟹ r ≠ Unwind i"
  using control_at_not_returning by fastforce
```

`is_returning` classifies the active head. The `unwinding` and `pop_ready` predicates refine that class with the nested residual shape needed by frame-pop proofs; they deliberately overlap instead of forming a phase partition.

```
lemma unwinding_is_returning: "unwinding u ⟹ is_returning u"
  by (induction u rule: unwinding.induct) auto

lemma pop_ready_is_returning: "pop_ready w ⟹ is_returning w"
  by (cases w rule: pop_ready.cases) (auto simp: unwinding_is_returning)

lemma is_returning_next_check: "is_returning c ⟹ next_check c = None"
  by (induction c) auto
```

A source configuration about to run a check is, under `csim`, at a node whose `EA_Check` edge carries that check. Suspended callers only wrap the running residual, and the frame-pop phase runs no check.

```
lemma csim_next_check_edge:
  assumes "Π, g, ϱ ⊢ (c, s, frs) ≈ (v, t, stk)" and "next_check c = Some (l, e)"
  shows "∃w. (v, EA_Check l e, w) ∈ intra g"
  using assms
proof (induction "(c, s, frs)" "(v, t, stk)" arbitrary: c s frs v t stk rule: csim.induct)
  case (Base p body_p k n c v s)
  from Base.hyps(2) obtain n' en E K
    where "compile Π p body_p k n = (n', en, E, K)" and "E ⊆ intra g"
    by blast
  with control_at_next_check_edge [OF Base.hyps(1) Base.prems] show ?case by blast
next
  case (Returning w i pc body_pc kc nc afters cont callee caller dst p)
  then show ?case by (simp add: is_returning_next_check pop_ready_is_returning)
qed simp
```

Stepping the frame-pop phase

How the phase between a return and its frame pop behaves: it neither changes the store nor touches the frames, and it stays in the phase until the pop. That is what lets the simulation hold the graph side still while the source side unwinds.

```
lemma pstep_unwinding:
  assumes "unwinding u" and "𝒢, Π ⊢ (u, s, frs) →p (u', s', frs')"
  shows "s' = s ∧ frs' = frs ∧ unwinding u'"
  using assms
proof (induction u arbitrary: u' s' frs')
  case (Seq u1 c2)
  from Seq.prems(1) have u1w: "unwinding u1" and c2R: "¬ is_Restore c2" by auto
  from Seq.prems(2) consider
      (dead) j where "u' = Unwind j" "s' = s" "frs' = frs"
    | (step) c1' where "u' = Seq c1' c2" "𝒢, Π ⊢ (u1, s, frs) →p (c1', s', frs')"
    using c2R unwinding_not_SKIP[OF u1w] by auto
  then show ?case
  proof cases
    case dead then show ?thesis by simp
  next
    case (step c1')
    from Seq.IH(1)[OF u1w step(2)] have "s' = s" "frs' = frs" "unwinding c1'" by simp_all
    then show ?thesis using step(1) c2R by simp
  qed
next
  case SKIP    from SKIP.prems(1)    show ?case by simp
next
  case Assign  from Assign.prems(1)  show ?case by simp
next
  case Check   from Check.prems(1)   show ?case by simp
next
  case If      from If.prems(1)      show ?case by simp
next
  case While   from While.prems(1)   show ?case by simp
next
  case Call    from Call.prems(1)    show ?case by simp
next
  case Return  from Return.prems(1)  show ?case by simp
next
  case Restore from Restore.prems(1) show ?case by simp
next
  case (Unwind i)
  from Unwind.prems(2) show ?case using pstep_Unwind_stuck by blast
qed

lemma pstep_pop_ready_head:
  assumes "pop_ready w" and "pop_ann w = Some i"
      and "𝒢, Π ⊢ (w, s, Frame fr dst # frs) →p x"
  shows "x = (SKIP i, combine_assign dst (s ret_var) (combine_env 𝒢 fr s), frs)
       ∨ (∃w'. x = (w', s, Frame fr dst # frs) ∧ pop_ready w' ∧ pop_ann w' = Some i)"
  using assms
proof (cases w rule: pop_ready.cases)
  case 1
  with assms(2,3) show ?thesis by auto
next
  case (2 u)
  hence uw: "unwinding u" using assms(1) by simp
  from assms(3) show ?thesis
    unfolding 2 using assms(2) unfolding 2
    by (elim SeqSE) (auto dest: pstep_unwinding[OF uw] simp: unwinding_not_SKIP[OF uw])
qed (use assms(1) in auto)
```

Stepping through a `seq_after` spine

A step of a `seq_after` spine is a step of its head, with the pending continuations carried along untouched. `SKIP` and `Unwind` heads are excluded because those relocate into the spine instead of stepping in place.

```
lemma pstep_seq_after_headE [elim]:
  assumes step: "𝒢, Π ⊢ (seq_after w afters, s, frs) →p src'"
      and wsk: "¬ is_SKIP w" and wunw: "⋀i. w ≠ Unwind i"
  obtains h' s' fz where
    "src' = (seq_after h' afters, s', fz)"
    "𝒢, Π ⊢ (w, s, frs) →p (h', s', fz)"
proof -
  have "∃h' s' fz. src' = (seq_after h' afters, s', fz)
```

```
        ∧ 𝒢, Π ⊢ (w, s, frs) →p (h', s', fz)"
    using step
  proof (induction afters arbitrary: src' rule: rev_induct)
    case Nil
    then obtain h' s' fz where "src' = (h', s', fz)"
      "𝒢, Π ⊢ (w, s, frs) →p (h', s', fz)"
      by (cases src') auto
    then show ?case by auto
  next
    case (snoc a xs)
    from snoc.prems have "𝒢, Π ⊢ (Seq (seq_after w xs) a, s, frs) →p src'"
      by (simp add: seq_after_snoc)
    then obtain B' s' fz where
      A: "src' = (Seq B' a, s', fz)"
        and B: "𝒢, Π ⊢ (seq_after w xs, s, frs) →p (B', s', fz)"
      using wsk wunw by auto
    from snoc.IH[OF B] obtain h' where
      "B' = seq_after h' xs" "𝒢, Π ⊢ (w, s, frs) →p (h', s', fz)" by auto
    then show ?case using A by (auto simp: seq_after_snoc)
  qed
  then show ?thesis using that by blast
qed

end
```

5.7 Simulation_Preservation

```
theory Simulation_Preservation
  imports Simulation_Relation Compile_Invariants
begin
```

Every source step is matched by the graph

The result the session exists for: if `csim` holds and the source takes one step, the graph can follow and `csim` holds again (`csim_step`); hence the same for a whole run (`csim_star`). A source redex is one of four things — a call, a return initiation, a return already in progress, or an ordinary step inside the activation — and there is one completion theorem per case, which `csim_step` dispatches to.

`procs_embedded` is the static side condition every case but the returning phase needs: every declared procedure's body has been compiled into this graph, with its entry and exit wiring present. It is deliberately not part of `csim`, which says only how two configurations correspond, so that the returning phase can proceed without it. `procs_embedded_compile_prog` discharges it for any graph `compile_prog` actually produced.

```
definition procs_embedded :: "'a ptable ⇒ cfg ⇒ bool" where
  "procs_embedded Π g ⟷
     (∀p decl. Π p = Some decl ⟶
        (∃k n n' en E K.
           compile Π p (body decl) k n = (n', en, E, K)
         ∧ E ⊆ intra g ∧ K ⊆ calls g
         ∧ (FunctionEntry p, EA_Body p, en) ∈ intra g
         ∧ (k, EA_Ret None p, FunctionResult p) ∈ intra g
         ∧ k = Statement (n + csize (body decl))
         ∧ source_com (body decl) ∧ special_table p = None))"

lemma procs_embedded_proc:
  assumes "procs_embedded Π g" and "Π p = Some decl"
  obtains k n n' en E K where
    "compile Π p (body decl) k n = (n', en, E, K)"
    "E ⊆ intra g" "K ⊆ calls g"
    "(FunctionEntry p, EA_Body p, en) ∈ intra g"
    "(k, EA_Ret None p, FunctionResult p) ∈ intra g"
    "k = Statement (n + csize (body decl))"
    "source_com (body decl)" "special_table p = None"
  using assms unfolding procs_embedded_def by blast

lemma procs_embedded_source_com:
  assumes "procs_embedded Π g" and "Π p = Some decl"
  shows "source_com (body decl)"
  using assms by (blast elim: procs_embedded_proc)
```

A declared procedure's name is never one `special_table` also classifies: `procs_embedded` bundles this disjointness alongside compiler-correctness so a declared call and a special call are always distinguishable without a separate well-formedness premise.

```
lemma procs_embedded_special_table_none:
  assumes "procs_embedded Π g" and "Π p = Some decl"
  shows "special_table p = None"
  using assms by (blast elim: procs_embedded_proc)
```

Everything needed to start a fresh activation of `p`, in one step: the certificate its `csim` layer will carry, the initial location of its body, and the `EA_Body` edge from `FunctionEntry p` that reaches that location. Call preservation is otherwise the same destructuring written out by hand.

```
lemma procs_embedded_activation:
  assumes pc: "procs_embedded Π g" and decl: "Π p = Some decl"
  obtains k n en where
    "compiled_at Π g p (body decl) k n"
    "control_at Π p (body decl) k n (body decl) en"
    "(FunctionEntry p, EA_Body p, en) ∈ intra g"
    "source_com (body decl)"
proof -
  obtain k n n' en E K where
    cb: "compile Π p (body decl) k n = (n', en, E, K)"
      and Esub: "E ⊆ intra g" and Ksub: "K ⊆ calls g"
      and entry: "(FunctionEntry p, EA_Body p, en) ∈ intra g"
      and ex: "(k, EA_Ret None p, FunctionResult p) ∈ intra g"
      and kk: "k = Statement (n + csize (body decl))"
      and src: "source_com (body decl)"
      and "special_table p = None"
    by (rule procs_embedded_proc[OF pc decl])
```

```
  have entry': "(FunctionEntry p, EA_Body p, Statement n) ∈ intra g"
    using entry compile_entry[OF cb] by simp
  have cacc: "compiled_at Π g p (body decl) k n"
    by (rule compiled_atI[OF decl refl cb Esub Ksub ex entry' kk])
  have ctrl: "control_at Π p (body decl) k n (body decl) en"
    using control_at_initial[OF src, of Π p k n, folded compile_entry[OF cb]] .
  from cacc ctrl entry src show ?thesis by (rule that)
qed
```

Frame-stack plumbing

A single `pstep` touches only the head region of the frame stack, so a bottom segment `extra` rides along unchanged when the active part `frs` is non-empty. This bridges the `Nested` snoc (the inner step threads all frames) to the induction hypothesis (stated on the inner frames alone).

```
lemma pstep_frame_restrict:
  "𝒢, Π ⊢ (c, s, fr) →p (c', s', frs') ⟹ fr = frs @ extra ⟹ frs ≠ [] ⟹
   ∃frs''. frs' = frs'' @ extra ∧ 𝒢, Π ⊢ (c, s, frs) →p (c', s', frs'')"
proof (induction "(c, s, fr)" "(c', s', frs')"
       arbitrary: c s fr frs c' s' frs' rule: pstep.induct)
  case (Seq2 c1 s1 f1 c1' s1' f1' c2 frs)
  from Seq2.hyps(2)[OF Seq2.prems] obtain frs'' where
    ih: "f1' = frs'' @ extra" "𝒢, Π ⊢ (c1, s1, frs) →p (c1', s1', frs'')" by blast
  show ?case
    by (rule exI[of _ frs'']) (use ih in auto)
next
  case (Call i p decl actuals dst vals callee s1 frs0 frs)
  then show ?case by auto
next
  case (RestoreStep i s1 fr0 dst frs0 frs)
  from RestoreStep.prems obtain frs1 where "frs = Frame fr0 dst # frs1" "frs0 = frs1 @ extra"
    by (auto simp: Cons_eq_append_conv)
  then show ?case by auto
next
  case (UnwindAct i j s1 fr0 dst frs0 frs)
  from UnwindAct.prems obtain frs1 where "frs = Frame fr0 dst # frs1" "frs0 = frs1 @ extra"
    by (auto simp: Cons_eq_append_conv)
  then show ?case by auto
qed auto
```

Unlike `pstep_frame_restrict` this needs no non-empty active stack: a call-headed command never heads with `SKIP` / `Restore` / `Unwind`, so no pop can fire.

```
lemma pstep_call_frame_restrict:
  "𝒢, Π ⊢ (c, s, fr) →p (c', s', frs') ⟹ head_call c ⟹ fr = frs @ extra ⟹
   ∃frs''. frs' = frs'' @ extra ∧ 𝒢, Π ⊢ (c, s, frs) →p (c', s', frs'')"
proof (induction "(c, s, fr)" "(c', s', frs')"
       arbitrary: c s fr frs c' s' frs' rule: pstep.induct)
  case (Seq2 c1 s1 f1 c1' s1' f1' c2 frs)
  from Seq2.prems(1) have "head_call c1" by simp
  from Seq2.hyps(2)[OF this Seq2.prems(2)] obtain frs'' where
    ih: "f1' = frs'' @ extra" "𝒢, Π ⊢ (c1, s1, frs) →p (c1', s1', frs'')" by blast
  show ?case by (rule exI[of _ frs'']) (use ih in auto)
next
  case (Call i p decl actuals dst vals callee s1 frs0 frs)
  then show ?case by auto
qed auto
```

The CFG dual: a `cstep` also touches only the head of the stack, so an extra bottom segment rides along — with no non-emptiness needed, since the return step already requires a non-empty stack.

```
lemma cstep_frame_extend:
  "𝒢, g ⊢ (u, s, stk) →c (u', s', stk') ⟹
   𝒢, g ⊢ (u, s, stk @ E) →c (u', s', stk' @ E)"
  by (erule cstep.cases) auto

lemma cstep_star_frame_extend:
  assumes "𝒢, g ⊢ c →c* c'"
  shows "𝒢, g ⊢ (fst c, fst (snd c), snd (snd c) @ E)
                      →c* (fst c', fst (snd c'), snd (snd c') @ E)"
  using assms
proof (induction rule: star.induct)
  case (refl a) show ?case by simp
next
  case (step a b c)
  obtain ua sa stka where a: "a = (ua, sa, stka)" by (cases a)
```

```
  obtain ub sb stkb where b: "b = (ub, sb, stkb)" by (cases b)
  from step.hyps(1) a b have "𝒢, g ⊢ (ua, sa, stka) →c (ub, sb, stkb)" by simp
  hence "𝒢, g ⊢ (ua, sa, stka @ E) →c (ub, sb, stkb @ E)" by (rule cstep_frame_extend)
  with step.IH a b show ?case by (auto intro: star.step)
qed
```

The `Nested` step-inversion rule: stepping a wrapped activation `Seq inner (Restore i)` is stepping `inner`, provided `inner` has neither completed nor begun to unwind (either of which would relocate rather than step).

```
lemma pstep_seq_after_seq_restore:
  assumes step: "𝒢, Π ⊢ (seq_after (Seq inner (Restore i)) afters, s, frs) →p src'"
      and nsk: "¬ is_SKIP inner" and nunw: "⋀j. inner ≠ Unwind j"
  obtains inner' s' fz where
    "src' = (seq_after (Seq inner' (Restore i)) afters, s', fz)"
    "𝒢, Π ⊢ (inner, s, frs) →p (inner', s', fz)"
proof -
  obtain h' s' fz where
    src': "src' = (seq_after h' afters, s', fz)"
      and hstep: "𝒢, Π ⊢ (Seq inner (Restore i), s, frs) →p (h', s', fz)"
    by (rule pstep_seq_after_headE[OF step]) auto
  from hstep nsk obtain inner' where
    "h' = Seq inner' (Restore i)" "𝒢, Π ⊢ (inner, s, frs) →p (inner', s', fz)"
    by (auto simp: nunw)
  with src' show ?thesis using that by blast
qed
```

The `Nested` step shared by every completion theorem: the inner activation's run lifts under the caller frame and its relation re-wraps as `Nested`. `csim` pins the graph store to the source store, so the lifted run ends at the source store.

```
lemma csim_Nested_lift:
  assumes run: "∃cfg'. 𝒢, g ⊢ (v0, s0, stk0) →c* cfg' ∧ Π, g, q ⊢ (inner', s', fz') ≈ cfg'"
      and loc: "control_at Π pc body_pc kc nc (seq_after (SKIP i) afters) cont"
      and cloc: "control_at Π pc body_pc kc nc (seq_after (Call i dst q args) afters) site"
      and cacc: "compiled_at Π g pc body_pc kc nc"
  shows "∃cfg'. 𝒢, g ⊢ (v0, s0, stk0 @ [(cont, dst, caller)]) →c* cfg'
             ∧ Π, g, pc ⊢ (seq_after (Seq inner' (Restore i)) afters, s', fz' @ [Frame caller dst]) ≈ cfg'"
proof -
  from run obtain v' t' stk' where
    inner: "𝒢, g ⊢ (v0, s0, stk0) →c* (v', t', stk')"
      and sim: "Π, g, q ⊢ (inner', s', fz') ≈ (v', t', stk')" by auto
  have teq: "t' = s'" using csim_store_eq[OF sim] by simp
  have "𝒢, g ⊢ (v0, s0, stk0 @ [(cont, dst, caller)])
                        →c* (v', s', stk' @ [(cont, dst, caller)])"
    using cstep_star_frame_extend[OF inner, of "[(cont, dst, caller)]"] teq by simp
  moreover have "Π, g, pc ⊢ (seq_after (Seq inner' (Restore i)) afters, s', fz' @ [Frame caller dst]) ≈ (v', s',
stk' @ [(cont, dst, caller)])"
    by (rule csim.Nested[OF sim[unfolded teq] loc cloc cacc])
  ultimately show ?thesis by blast
qed
```

Call preservation

A source `Call` at the head of the active residual is matched by two `csteps` — the call edge to `FunctionEntry q` and the `EA_Body` edge to the callee body — and adds exactly one `Nested` layer: the callee body becomes a fresh `Base` activation of `q`, and the caller resumes at `seq_after (SKIP i) afters`. Everything the callee needs comes from the single `procs_embedded` certificate.

```
lemma csim_call_base:
  assumes pc: "procs_embedded Π g"
      and loc: "control_at Π p body_p kk n (seq_after (Call i dst q actuals) afters) v"
      and cacc: "compiled_at Π g p body_p kk n"
      and decl: "Π q = Some decl"
  shows "∃cfg'. 𝒢, g ⊢ (v, s, []) →c* cfg'
             ∧ Π, g, p ⊢ (seq_after (Seq (body decl) (Restore i)) afters,
                          bind_formals (formals decl) (map (λe. ⟦e⟧e s) actuals)
                            (enter_state 𝒢 s),
                          [Frame s dst]) ≈ cfg'"
proof -
  let ?callee =
    "bind_formals (formals decl) (map (λe. ⟦e⟧e s) actuals) (enter_state 𝒢 s)"
  have spNone: "special_table q = None" by (rule procs_embedded_special_table_none[OF pc decl])
  from cacc obtain n' en E K where comp: "compile Π p body_p kk n = (n', en, E, K)"
      and Ksub: "K ⊆ calls g" by blast
  from control_at_seq_after_call_edge[OF loc refl comp spNone] obtain j w where
```

```
    vk: "v = Statement j"
    and edgeK: "(Statement j, CallEdge dst (call_formals Π q) actuals,
                 FunctionEntry q, w) ∈ K"
    and callerSKIP: "control_at Π p body_p kk n (seq_after (SKIP i) afters) w" by blast
  have edge: "(Statement j, CallEdge dst (formals decl) actuals, FunctionEntry q, w)
               ∈ calls g" using edgeK Ksub by (auto simp: decl)
  have cstep1: "𝒢, g ⊢ (Statement j, s, [])
          →c (FunctionEntry q, call_enter 𝒢 (CallEdge dst (formals decl) actuals) s,
           [(w, dst, s)])" by (rule cstep.Call[OF edge])
  have ce: "call_enter 𝒢 (CallEdge dst (formals decl) actuals) s = ?callee"
    by (rule call_enter_CallEdge)
  obtain kq m en_q where
    caccq: "compiled_at Π g q (body decl) kq m"
      and ctrlq: "control_at Π q (body decl) kq m (body decl) en_q"
      and entry: "(FunctionEntry q, EA_Body q, en_q) ∈ intra g"
    by (rule procs_embedded_activation[OF pc decl])
  have cstep2: "𝒢, g ⊢ (FunctionEntry q, ?callee, [(w, dst, s)])
                       →c (en_q, ?callee, [(w, dst, s)])"
    using cstep_body[OF entry] .
  have star: "𝒢, g ⊢ (v, s, []) →c* (en_q, ?callee, [(w, dst, s)])"
    using cstep1[unfolded ce] cstep2 vk by (simp add: star.step)
  have baseCallee: "Π, g, q ⊢ (body decl, ?callee, []) ≈ (en_q, ?callee, [])"
    by (rule csim.Base[OF ctrlq caccq])
  have "Π, g, p ⊢ (seq_after (Seq (body decl) (Restore i)) afters, ?callee, [] @ [Frame s dst]) ≈ (en_q, ?callee,
[] @ [(w, dst, s)])"
    by (rule csim.Nested[OF baseCallee callerSKIP loc cacc])
  then have "Π, g, p ⊢ (seq_after (Seq (body decl) (Restore i)) afters, ?callee, [Frame s dst]) ≈ (en_q, ?callee,
[(w, dst, s)])"
    by simp
  with star show ?thesis by blast
qed

theorem csim_call_completion:
  "Π, g, ϱ ⊢ (c, s, frs) ≈ (v, t, stk) ⟹ procs_embedded Π g ⟹ head_call c ⟹
   𝒢, Π ⊢ (c, s, frs) →p src' ⟹
   ∃cfg'. 𝒢, g ⊢ (v, t, stk) →c* cfg' ∧ Π, g, ϱ ⊢ src' ≈ cfg'"
proof (induction "(c, s, frs)" "(v, t, stk)" arbitrary: c s frs v t stk src' rule: csim.induct)
  case (Base p body_p kk n cc vv ss)
  from head_call_seq_after_form[OF Base.prems(2)] obtain i dst q actuals afters where
    ceq: "cc = seq_after (Call i dst q actuals) afters" and spNone: "special_table q = None"
    by blast
  have step: "𝒢, Π ⊢ (seq_after (Call i dst q actuals) afters, ss, []) →p src'"
    using Base.prems(3) ceq by simp
  have w1: "¬ is_SKIP (Call i dst q actuals)" by simp
  have w2: "⋀j. Call i dst q actuals ≠ Unwind j" by simp
  obtain h' s' fz where
    src': "src' = (seq_after h' afters, s', fz)"
      and pcall: "𝒢, Π ⊢ (Call i dst q actuals, ss, []) →p (h', s', fz)"
    by (rule pstep_seq_after_headE[OF step w1 w2])
  from pcall spNone obtain decl where
    qdecl: "Π q = Some decl"
      and heq: "h' = Seq (body decl) (Restore i)"
      and seq: "s' = bind_formals (formals decl) (map (λe. ⟦e⟧e ss) actuals)
                       (enter_state 𝒢 ss)"
      and fzeq: "fz = [Frame ss dst]"
    by auto
  have loc': "control_at Π p body_p kk n (seq_after (Call i dst q actuals) afters) vv"
    using Base.hyps(1) ceq by simp
  from csim_call_base[OF Base.prems(1) loc' Base.hyps(2) qdecl, where s = ss]
  obtain cfg' where
    cstar: "𝒢, g ⊢ (vv, ss, []) →c* cfg'"
      and csimr: "Π, g, p ⊢ (seq_after (Seq (body decl) (Restore i)) afters,
                bind_formals (formals decl) (map (λe. ⟦e⟧e ss) actuals)
                  (enter_state 𝒢 ss),
                [Frame ss dst]) ≈ cfg'" by blast
  from csimr have "Π, g, p ⊢ src' ≈ cfg'" by (simp add: src' heq seq fzeq)
  with cstar show ?case by (intro exI conjI)
next
  case (Returning w i pc body_pc kc nc afters cont dst p args site callee caller)
  from Returning.prems(2) have "head_call w" by simp
  with pop_ready_not_head_call[OF Returning.hyps(1)] show ?case by simp
next
  case (Nested q inner s0 frs0 v0 stk0 pc body_pc kc nc i afters cont dst args site caller)
  have headinner: "head_call inner" using Nested.prems(2) by simp
  have nsk: "¬ is_SKIP inner" using headinner by (rule head_call_not_SKIP)
  have nunw: "⋀j. inner ≠ Unwind j" using headinner by (rule head_call_not_Unwind)
```

```
  obtain inner' s' fz where
    src': "src' = (seq_after (Seq inner' (Restore i)) afters, s', fz)"
      and stepin: "𝒢, Π ⊢ (inner, s0, frs0 @ [Frame caller dst]) →p (inner', s', fz)"
    by (rule pstep_seq_after_seq_restore[OF Nested.prems(3) nsk nunw])
  from pstep_call_frame_restrict[OF stepin headinner refl] obtain fz' where
    fz: "fz = fz' @ [Frame caller dst]"
      and stepin': "𝒢, Π ⊢ (inner, s0, frs0) →p (inner', s', fz')" by blast
  from csim_Nested_lift[OF Nested.hyps(2)[OF Nested.prems(1) headinner stepin'] Nested.hyps(3,4,5)]
  show ?case by (simp add: src' fz)
qed
```

Intra-procedural preservation and callee fall-through

The bulk of the simulation: a step that stays inside one activation. Either the callee body still has work, in which case the step happens under the enclosing `Restore` and the stack is untouched, or the body is finished and control falls through to that `Restore` – the only place a callee reaches its own exit without a return statement.

```
lemma intra_step_seq_after_seq_restore:
  "intra_step Π (seq_after (Seq inner (Restore i)) afters, s, frs) src' ⟹
   (is_SKIP inner ∧ src' = (seq_after (Restore i) afters, s, frs))
   ∨ (∃inner' s'. src' = (seq_after (Seq inner' (Restore i)) afters, s', frs)
        ∧ intra_step Π (inner, s, frs) (inner', s', frs))"
proof (induction afters arbitrary: src' rule: rev_induct)
  case Nil
  obtain c' s' frs' where sc: "src' = (c', s', frs')" by (cases src')
  from Nil.prems sc have "intra_step Π (Seq inner (Restore i), s, frs) (c', s', frs')" by simp
  from intra_Seq_cases[OF this] show ?case by (auto simp: sc)
next
  case (snoc a xs)
  obtain c' s' frs' where sc: "src' = (c', s', frs')" by (cases src')
  from snoc.prems sc
  have "intra_step Π (Seq (seq_after (Seq inner (Restore i)) xs) a, s, frs) (c', s', frs')"
    by (simp add: seq_after_snoc)
  from intra_Seq_cases[OF this] obtain B' where
    A: "c' = Seq B' a" "frs' = frs"
      and B: "intra_step Π (seq_after (Seq inner (Restore i)) xs, s, frs) (B', s', frs)"
    by auto
  from snoc.IH[OF B] show ?case
  proof (rule disjE)
    assume "is_SKIP inner ∧ (B', s', frs) = (seq_after (Restore i) xs, s, frs)"
    then show ?case using A sc by (auto simp: seq_after_snoc)
  next
    assume "∃inner' s''. (B', s', frs) = (seq_after (Seq inner' (Restore i)) xs, s'', frs)
              ∧ intra_step Π (inner, s, frs) (inner', s'', frs)"
    then show ?case using A sc by (auto simp: seq_after_snoc)
  qed
qed
```

A `Nested` callee that *completes* (`inner = SKIP`) exposes `Restore` through the `ISeq1` fall-through; the CFG runs the completed callee to its exit (`control_at_skip_to_exit`) and takes the `EA_Ret None p` edge into `FunctionResult p` (`compiled_at_exit`), landing in a `Returning` activation.

```
theorem csim_intra_completion:
  "Π, g, ϱ ⊢ (c, s, frs) ≈ (v, t, stk) ⟹ procs_embedded Π g ⟹
   intra_step Π (c, s, frs) src' ⟹
   ∃cfg'. 𝒢, g ⊢ (v, t, stk) →c* cfg' ∧ Π, g, ϱ ⊢ src' ≈ cfg'"
proof (induction "(c, s, frs)" "(v, t, stk)" arbitrary: c s frs v t stk src' rule: csim.induct)
  case (Base p body_p kk n cc vv ss)
  obtain c' s' frs' where sc: "src' = (c', s', frs')" by (cases src')
  from Base.prems(2) sc have istep: "intra_step Π (cc, ss, []) (c', s', frs')" by simp
  from Base.hyps(2) obtain decl where "Π p = Some decl" "body_p = body decl"
    by (rule compiled_at_decl)
  have srcbody: "source_com body_p"
    using procs_embedded_source_com[OF Base.prems(1) ‹Π p = Some decl›] ‹body_p = body decl›
    by simp
  from Base.hyps(2) obtain n' en E K where comp: "compile Π p body_p kk n = (n', en, E, K)"
      and Esub: "E ⊆ intra g" by blast
  from intra_step_simulation[OF Base.hyps(1) istep comp Esub srcbody, where stk = "[]"]
  obtain v' where feq: "frs' = []" and loc': "control_at Π p body_p kk n c' v'"
      and cstar: "𝒢, g ⊢ (vv, ss, []) →c* (v', s', [])" by blast
  have "Π, g, p ⊢ (c', s', []) ≈ (v', s', [])" by (rule csim.Base[OF loc' Base.hyps(2)])
  with cstar show ?case using sc feq by auto
next
  case (Returning w i pc body_pc kc nc afters cont dst p args site callee caller)
  from Returning.prems(2) have "¬ is_returning (seq_after w afters)"
    by (cases src') (auto dest: intra_step_not_returning)
```

```
  with Returning.hyps(1) show ?case by (simp add: pop_ready_is_returning)
next
  case (Nested q inner s0 frs0 v0 stk0 pc body_pc kc nc i afters cont dst args site caller)
  from Nested.prems(2)
  have "intra_step Π (seq_after (Seq inner (Restore i)) afters, s0, frs0 @ [Frame caller dst]) src'"
    .
  from intra_step_seq_after_seq_restore[OF this] show ?case
  proof (rule disjE)
    assume fall:
      "is_SKIP inner ∧ src' = (seq_after (Restore i) afters, s0, frs0 @ [Frame caller dst])"
    then have innerSKIP: "is_SKIP inner"
      and src': "src' = (seq_after (Restore i) afters, s0, frs0 @ [Frame caller dst])" by auto
    from innerSKIP obtain j where ij: "inner = SKIP j" by auto
    have baseInner: "Π, g, q ⊢ (SKIP j, s0, frs0) ≈ (v0, s0, stk0)"
      using Nested.hyps(1) ij by simp
    from baseInner obtain c0in kin nin where
      ctrl: "control_at Π q c0in kin nin (SKIP j) v0" and cacc: "compiled_at Π g q c0in kin nin"
        and frs0nil: "frs0 = []" and stk0nil: "stk0 = []"
    proof (cases rule: csim.cases)
      case (Base body_p2 k2 n2) with that show ?thesis by auto
    qed simp_all
    from cacc obtain n' en E K where comp: "compile Π q c0in kin nin = (n', en, E, K)"
      and Esub: "E ⊆ intra g"
      and exitedge: "(kin, EA_Ret None q, FunctionResult q) ∈ intra g"
      by blast
    have star1: "𝒢, g ⊢ (v0, s0, [(cont, dst, caller)])
                  →c* (kin, s0, [(cont, dst, caller)])"
      by (rule control_at_skip_to_exit[OF ctrl _ comp Esub]) simp
    have "𝒢, g ⊢ (kin, s0, [(cont, dst, caller)])
           →c (FunctionResult q, s0, [(cont, dst, caller)])"
      using cstep.Intra[OF exitedge edge_step_EA_Ret_ret_store_mem] by simp
    with star1 have star: "𝒢, g ⊢ (v0, s0, [(cont, dst, caller)])
                            →c* (FunctionResult q, s0, [(cont, dst, caller)])"
      by (rule star_trans[OF _ star_step1])
    have "Π, g, pc ⊢ (seq_after (Restore i) afters, s0, [Frame caller dst]) ≈ (FunctionResult q, s0, [(cont, dst,
caller)])"
      by (rule csim.Returning[OF _ _ Nested.hyps(3) Nested.hyps(4) Nested.hyps(5)]) simp_all
    then have "Π, g, pc ⊢ src' ≈ (FunctionResult q, s0, [(cont, dst, caller)])"
      using src' frs0nil by simp
    with star show ?case using stk0nil by auto
  next
    assume "∃inner' s'.
              src' = (seq_after (Seq inner' (Restore i)) afters, s', frs0 @ [Frame caller dst])
              ∧ intra_step Π (inner, s0, frs0 @ [Frame caller dst])
                   (inner', s', frs0 @ [Frame caller dst])"
    then obtain inner' s' where
      src': "src' = (seq_after (Seq inner' (Restore i)) afters, s', frs0 @ [Frame caller dst])"
        and stepin: "intra_step Π (inner, s0, frs0 @ [Frame caller dst])
                       (inner', s', frs0 @ [Frame caller dst])"
      by blast
    from intra_step_any_frame[OF stepin]
    have stepin': "intra_step Π (inner, s0, frs0) (inner', s', frs0)" .
    from csim_Nested_lift[OF Nested.hyps(2)[OF Nested.prems(1) stepin'] Nested.hyps(3,4,5)]
    show ?case by (simp add: src')
  qed
qed
```

Return initiation

A source `Return` in head position fires `Return e --> Unwind`, turning the innermost callee activation into its returning phase, while the CFG takes the `EA_Ret e p` edge to `FunctionResult p`. The enclosing `Nested` wrapper becomes `Returning`: its body `Seq (seq_after (Unwind j) cafters) (Restore i)` is `pop_ready` because the callee's pending continuations `cafters` — source residuals of a `control_at` location — carry no `Restore`. The `frs ≠ []` premise excludes the ill-formed base `Return` (a stuck top-level return); `csim_step` discharges it from `return_safe`.

```
theorem csim_return_init_completion:
  "Π, g, ϱ ⊢ (c, s, frs) ≈ (v, t, stk) ⟹ procs_embedded Π g ⟹ frs ≠ [] ⟹
   head_return c ⟹ 𝒢, Π ⊢ (c, s, frs) →p src' ⟹
   ∃cfg'. 𝒢, g ⊢ (v, t, stk) →c* cfg' ∧ Π, g, ϱ ⊢ src' ≈ cfg'"
proof (induction "(c, s, frs)" "(v, t, stk)" arbitrary: c s frs v t stk src' rule: csim.induct)
  case (Base p body_p kk n cc vv ss)
  from Base.prems(2) show ?case by simp
next
  case (Returning w i pc body_pc kc nc afters cont dst p args site callee caller)
  from Returning.prems(3) pop_ready_not_head_return[OF Returning.hyps(1)]
```

```
  show ?case by simp
next
  case (Nested q inner s0 frs0 v0 stk0 pc body_pc kc nc i afters cont dst args site caller)
  have hr_inner: "head_return inner" using Nested.prems(3) by simp
  have nsk: "¬ is_SKIP inner" using hr_inner by (rule head_return_not_SKIP)
  have nunw: "inner ≠ Unwind j" for j using hr_inner by auto
  obtain inner' s' fz where
    src': "src' = (seq_after (Seq inner' (Restore i)) afters, s', fz)"
      and stepin: "𝒢, Π ⊢ (inner, s0, frs0 @ [Frame caller dst]) →p (inner', s', fz)"
    by (rule pstep_seq_after_seq_restore[OF Nested.prems(4) nsk nunw])
  show ?case
  proof (cases "frs0 = []")
    case True
    have baseInner: "Π, g, q ⊢ (inner, s0, []) ≈ (v0, s0, stk0)" using Nested.hyps(1) True by simp
    obtain c0in kin nin where
      stk0nil: "stk0 = []"
        and ctrl: "control_at Π q c0in kin nin inner v0"
        and cacc: "compiled_at Π g q c0in kin nin"
      using baseInner by blast
    from cacc obtain decl where "Π q = Some decl" "c0in = body decl"
      by (rule compiled_at_decl)
    have srcc0: "source_com c0in"
      using procs_embedded_source_com[OF Nested.prems(1) ‹Π q = Some decl›]
            ‹c0in = body decl›
      by simp
    obtain ir e cafters where innerform: "inner = seq_after (Return ir e) cafters"
      using head_return_seq_after_form[OF hr_inner] by blast
    — the head step is Return e --> Unwind, leaving the continuations cafters
    have stepin1: "𝒢, Π ⊢ (seq_after (Return ir e) cafters, s0, [Frame caller dst])
                    →p (inner', s', fz)"
      using stepin True innerform by simp
    obtain h' where inner'form: "inner' = seq_after h' cafters"
        and hstep: "𝒢, Π ⊢ (Return ir e, s0, [Frame caller dst]) →p (h', s', fz)"
      by (rule pstep_seq_after_headE[OF stepin1]) auto
    have hUnw: "h' = Unwind ir" and s'eq: "s' = ret_store e s0" and fzeq: "fz = [Frame caller dst]"
      using hstep by (cases e; auto simp: ret_store_def)+
    — the callee continuations carry no Restore, so the Unwind spine is pop_ready
    have noRest: "∀a ∈ set cafters. ¬ is_Restore a"
      by (rule control_at_head_return_afters_no_Restore[OF ctrl[unfolded innerform] srcc0])
    have popready: "pop_ready (Seq (seq_after (Unwind ir) cafters) (Restore i))"
      using unwinding_seq_after_Unwind[OF noRest] by simp
    — CFG: one Intra step along the return edge to FunctionResult q
    from cacc obtain n' en E K where comp: "compile Π q c0in kin nin = (n', en, E, K)"
        and Esub: "E ⊆ intra g" by blast
    obtain j where vk: "v0 = Statement j"
        and edge: "(Statement j, EA_Ret e q, FunctionResult q) ∈ E"
      using control_at_seq_after_return_edge[OF ctrl[unfolded innerform] refl comp] by blast
    have edgeg: "(Statement j, EA_Ret e q, FunctionResult q) ∈ intra g"
      using edge Esub by blast
    have cstep1: "𝒢, g ⊢ (Statement j, s0, [(cont, dst, caller)])
                         →c (FunctionResult q, ret_store e s0, [(cont, dst, caller)])"
      using cstep.Intra[OF edgeg edge_step_EA_Ret_ret_store_mem] .
    — the enclosing wrapper becomes Returning
    have rel: "Π, g, pc ⊢ (seq_after (Seq (seq_after (Unwind ir) cafters) (Restore i)) afters, ret_store e s0,
        [Frame caller dst]) ≈ (FunctionResult q, ret_store e s0, [(cont, dst, caller)])"
      by (rule csim.Returning[OF popready _ Nested.hyps(3) Nested.hyps(4) Nested.hyps(5)]) simp
    have srcshape: "src' = (seq_after (Seq (seq_after (Unwind ir) cafters) (Restore i)) afters,
                            ret_store e s0, [Frame caller dst])"
      using src' inner'form hUnw s'eq fzeq by simp
    have "𝒢, g ⊢ (v0, s0, stk0 @ [(cont, dst, caller)])
                        →c* (FunctionResult q, ret_store e s0, [(cont, dst, caller)])"
      using cstep1 vk stk0nil by auto
    with rel srcshape show ?thesis by auto
  next
    case False
    from pstep_frame_restrict[OF stepin refl False] obtain fz' where
      fz: "fz = fz' @ [Frame caller dst]"
        and stepin': "𝒢, Π ⊢ (inner, s0, frs0) →p (inner', s', fz')" by blast
    from csim_Nested_lift[OF Nested.hyps(2)[OF Nested.prems(1) False hr_inner stepin']
                             Nested.hyps(3,4,5)]
    show ?thesis by (simp add: src' fz)
  qed
qed
```

Return propagation and the frame pop

The half of the simulation where the two sides advance at different rates. A source return walks out of the nested residual one dead layer at a time while the graph stays at the callee result node, and only the final pop takes a graph step – so the correspondence is a star, not a step-for-step match.

```
lemma csim_returning_frames_nonempty:
  "Π, g, ϱ ⊢ (c, s, frs) ≈ (v, t, stk) ⟹ is_returning c ⟹ frs ≠ [] ∧ stk ≠ []"
  by (erule csim.cases) (auto dest: control_at_not_returning)

lemma csim_not_unwind:
  "Π, g, ϱ ⊢ (Unwind i, s, frs) ≈ cfg' ⟹ False"
  by (erule csim.cases) (auto dest: control_at_not_unwind pop_ready_not_Unwind)
```

One `pstep` of a base returning activation `seq_after w afters` (`pop_ready w`, framed by the single caller frame, CFG at `FunctionResult p`) either *pops*: the source resumes at `seq_after (SKIP i) afters` with the combined store and the CFG performs exactly one return `cstep` to `cont`, rebuilt as a `Base` activation; or *propagates*: an `Unwind` advances one dead-code layer and the CFG stays at `FunctionResult p` with *zero* `cstep`, rebuilt as a `Returning` activation.

```
lemma csim_returning_base_completion:
  assumes pr: "pop_ready w" and ann: "pop_ann w = Some i"
      and loc: "control_at Π pc body_pc kc nc (seq_after (SKIP i) afters) cont"
      and cloc: "control_at Π pc body_pc kc nc (seq_after (Call i dst q args) afters) site"
      and cacc: "compiled_at Π g pc body_pc kc nc"
      and step: "𝒢, Π ⊢ (seq_after w afters, callee, [Frame caller dst]) →p src'"
  shows "∃cfg'. 𝒢, g ⊢ (FunctionResult q, callee, [(cont, dst, caller)]) →c* cfg'
             ∧ Π, g, pc ⊢ src' ≈ cfg'"
proof -
  have wsk: "¬ is_SKIP w" using pr by (rule pop_ready_not_SKIP)
  have wunw: "w ≠ Unwind j" for j using pr by (rule pop_ready_not_Unwind)
  obtain h' s' frs' where
    src': "src' = (seq_after h' afters, s', frs')"
      and hstep: "𝒢, Π ⊢ (w, callee, [Frame caller dst]) →p (h', s', frs')"
    by (rule pstep_seq_after_headE[OF step wsk wunw])
  let ?rs = "combine_collect 𝒢 dst caller callee"
  from pstep_pop_ready_head[OF pr ann hstep] show ?thesis
  proof (rule disjE)
    assume "(h', s', frs') =
              (SKIP i, combine_assign dst (callee ret_var) (combine_env 𝒢 caller callee), [])"
    hence h': "h' = SKIP i" "s' = ?rs" "frs' = []" by (auto simp: combine_collect_def)
    have "𝒢, g ⊢ (FunctionResult q, callee, [(cont, dst, caller)]) →c (cont, ?rs, [])"
      by (rule cstep.Return)
    moreover have "Π, g, pc ⊢ (seq_after (SKIP i) afters, ?rs, []) ≈ (cont, ?rs, [])"
      by (rule csim.Base[OF loc cacc])
    ultimately show ?thesis using src' h' by auto
  next
    assume "∃w'. (h', s', frs') = (w', callee, [Frame caller dst])
                ∧ pop_ready w' ∧ pop_ann w' = Some i"
    then obtain w' where h': "h' = w'" "s' = callee" "frs' = [Frame caller dst]"
        and pr': "pop_ready w'" and ann': "pop_ann w' = Some i" by auto
    have "Π, g, pc ⊢ (seq_after w' afters, callee, [Frame caller dst]) ≈ (FunctionResult q, callee, [(cont, dst,
caller)])"
      by (rule csim.Returning[OF pr' ann' loc cloc cacc])
    with src' h' show ?thesis by auto
  qed
qed
```

Deep return completion: the base case above, threaded out through any number of `Nested` wrappers. No `procs_embedded` is needed — the returning phase runs on the certificates `csim` already carries.

```
theorem csim_returning_completion:
  "Π, g, ϱ ⊢ (c, s, frs) ≈ (v, t, stk) ⟹ is_returning c ⟹
   𝒢, Π ⊢ (c, s, frs) →p src' ⟹
   ∃cfg'. 𝒢, g ⊢ (v, t, stk) →c* cfg' ∧ Π, g, ϱ ⊢ src' ≈ cfg'"
proof (induction "(c, s, frs)" "(v, t, stk)" arbitrary: c s frs v t stk src' rule: csim.induct)
  case (Base p body_p kk n cc vv ss)
  from control_at_not_returning[OF Base.hyps(1)] Base.prems(1) show ?case by simp
next
  case (Returning w i pc body_pc kc nc ao cont dst p args site callee caller)
  from csim_returning_base_completion[OF Returning.hyps(1) Returning.hyps(2) Returning.hyps(3)
                                         Returning.hyps(4) Returning.hyps(5) Returning.prems(2)]
  show ?case .
next
  case (Nested q inner s0 frs0 v0 stk0 pc body_pc kc nc i afters cont dst args site caller)
  have retinner: "is_returning inner" using Nested.prems(1) by simp
```

```
  have nsk: "¬ is_SKIP inner" using retinner by auto
  have nunw: "inner ≠ Unwind j" for j
  proof
    assume "inner = Unwind j"
    with Nested.hyps(1) show False by (blast dest: csim_not_unwind)
  qed
  have frsne: "frs0 ≠ []"
    using csim_returning_frames_nonempty[OF Nested.hyps(1) retinner] by simp
  obtain inner' s' fz where
    src': "src' = (seq_after (Seq inner' (Restore i)) afters, s', fz)"
      and stepin: "𝒢, Π ⊢ (inner, s0, frs0 @ [Frame caller dst]) →p (inner', s', fz)"
    by (rule pstep_seq_after_seq_restore[OF Nested.prems(2) nsk nunw])
  from pstep_frame_restrict[OF stepin refl frsne] obtain fz' where
    fz: "fz = fz' @ [Frame caller dst]"
      and stepin': "𝒢, Π ⊢ (inner, s0, frs0) →p (inner', s', fz')" by blast
  from csim_Nested_lift[OF Nested.hyps(2)[OF retinner stepin'] Nested.hyps(3,4,5)]
  show ?case by (simp add: src' fz)
qed
```

Single-step and finite-execution forward simulation

Only a `Base` activation has an empty stack, and its located residual is a source command, so `return_safe` forbids it from heading with `Return`. This is where `return_safe` discharges the nonempty-frame premise of `csim_return_init_completion`.

```
lemma csim_head_return_frames:
  assumes SIM: "Π, g, ϱ ⊢ (c, s, frs) ≈ (v, t, stk)"
      and PC: "procs_embedded Π g"
      and WF: "return_safe c"
      and RET: "head_return c"
  shows "frs ≠ []"
proof (rule ccontr)
  assume "¬ frs ≠ []"
  hence base: "Π, g, ϱ ⊢ (c, s, []) ≈ (v, t, stk)" using SIM by simp
  obtain p body_p kk n where
    ctrl: "control_at Π p body_p kk n c v" and cacc: "compiled_at Π g p body_p kk n"
    using base by blast
  from cacc obtain decl where "Π p = Some decl" "body_p = body decl"
    by (rule compiled_at_decl)
  have "source_com body_p"
    using procs_embedded_source_com[OF PC ‹Π p = Some decl›] ‹body_p = body decl› by simp
  hence "source_com c" using control_at_source_com[OF ctrl] by simp
  from return_safe_not_head_return[OF WF this] RET show False by simp
qed
```

One-step forward simulation: the four-way redex classification (`head_call` / `head_return` / `is_returning` / otherwise-intra) dispatched to the four completion theorems above. `return_safe` is used only to supply the nonempty frame for the return-initiation branch; the statement exposes none of the internal classifiers, compile tuples, offsets, or frame conditions.

```
theorem csim_step:
  assumes SIM: "Π, g, ϱ ⊢ (c, s, frs) ≈ (v, t, stk)"
      and PC: "procs_embedded Π g"
      and WF: "return_safe c"
      and STEP: "𝒢, Π ⊢ (c, s, frs) →p src'"
  shows "∃cfg'. 𝒢, g ⊢ (v, t, stk) →c* cfg' ∧ Π, g, ϱ ⊢ src' ≈ cfg'"
proof -
  consider (call) "head_call c" | (ret) "head_return c" | (returning) "is_returning c"
    | (intra) "¬ head_call c" "¬ head_return c" "¬ is_returning c" by blast
  then show ?thesis
  proof cases
    case call
    from csim_call_completion[OF SIM PC call STEP] show ?thesis .
  next
    case ret
    from csim_head_return_frames[OF SIM PC WF ret]
    have "frs ≠ []" .
    from csim_return_init_completion[OF SIM PC this ret STEP] show ?thesis .
  next
    case returning
    from csim_returning_completion[OF SIM returning STEP] show ?thesis .
  next
    case intra
    obtain c' s' frs' where sc: "src' = (c', s', frs')" by (cases src')
    have "intra_step Π (c, s, frs) (c', s', frs')"
```

```
      by (rule pstep_intra_classify[OF procs_embedded_special_table_none[OF PC]
            STEP[unfolded sc] intra(1) intra(2) intra(3)])
    from csim_intra_completion[OF SIM PC this] show ?thesis by (simp add: sc)
  qed
qed

lemma csim_run:
  assumes PC: "procs_embedded Π g"
      and RUN: "𝒢, Π ⊢ sc →ₚ* sc'"
      and SIM: "Π, g, ϱ ⊢ sc ≈ dg"
      and WF: "return_safe (fst sc)"
  shows "∃dg'. 𝒢, g ⊢ dg →꜀* dg' ∧ Π, g, ϱ ⊢ sc' ≈ dg'"
  using RUN SIM WF
proof (induction arbitrary: dg rule: star.induct)
  case (refl a)
  obtain v0 t0 k0 where "dg = (v0, t0, k0)" by (cases dg)
  with refl show ?case by auto
next
  case (step a b cc)
  obtain c0 s0 f0 where a: "a = (c0, s0, f0)" by (cases a)
  obtain c1 s1 f1 where b: "b = (c1, s1, f1)" by (cases b)
  obtain v0 t0 k0 where dgd: "dg = (v0, t0, k0)" by (cases dg)
  have SIMa: "Π, g, ϱ ⊢ (c0, s0, f0) ≈ (v0, t0, k0)" using step.prems(1) a dgd by simp
  have WFa: "return_safe c0" using step.prems(2) a by simp
  have STEP: "𝒢, Π ⊢ (c0, s0, f0) →ₚ (c1, s1, f1)" using step.hyps(1) a b by simp
  from csim_step[OF SIMa PC WFa STEP] obtain dg1 where
    both: "𝒢, g ⊢ (v0, t0, k0) →꜀* dg1 ∧ Π, g, ϱ ⊢ (c1, s1, f1) ≈ dg1" ..
  from both have run1: "𝒢, g ⊢ (v0, t0, k0) →꜀* dg1" by (rule conjunct1)
  from both have sim1: "Π, g, ϱ ⊢ (c1, s1, f1) ≈ dg1" by (rule conjunct2)
  have bodies: "⋀p decl. Π p = Some decl ⟹ source_com (body decl)"
    using procs_embedded_source_com[OF PC] .
  have WFb: "return_safe (fst b)" using return_safe_pstep[OF bodies STEP WFa] b by simp
  have sim1b: "Π, g, ϱ ⊢ b ≈ dg1" using sim1 b by simp
  from step.IH[OF sim1b WFb] obtain dg' where
    run2: "𝒢, g ⊢ dg1 →꜀* dg'" and sim2: "Π, g, ϱ ⊢ cc ≈ dg'" by blast
  from run1 run2 have "𝒢, g ⊢ (v0, t0, k0) →꜀* dg'" by (rule star_trans)
  with sim2 show ?case using dgd by blast
qed

theorem csim_star:
  assumes SIM: "Π, g, ϱ ⊢ (c, s, frs) ≈ (v, t, stk)"
      and PC: "procs_embedded Π g"
      and WF: "return_safe c"
      and RUN: "𝒢, Π ⊢ (c, s, frs) →ₚ* src'"
  shows "∃cfg'. 𝒢, g ⊢ (v, t, stk) →꜀* cfg' ∧ Π, g, ϱ ⊢ src' ≈ cfg'"
  using csim_run[OF PC RUN SIM] WF by simp
```

The static certificate for a whole compiled program

The static source contract establishes the runtime return guard for the root activation.

```
lemma wf_compile_input_return_safe:
  assumes "wf_compile_input 𝒢 Π ps"
  shows "return_safe (main_body Π)"
  using wf_compile_inputD(8)[OF assms] wf_compile_inputD(4)[OF assms]
  by (rule return_safe_if_no_return)
```

`procs_embedded` is the static certificate `csim` reads: every procedure declared in Π has its body fragment, entry `EA_Body` wiring and `EA_Ret None` exit wiring in the target graph. It covers the entry procedure too, since Π `prog_main_name = Some (⦇formals = [], body = main_body Π⦈)` makes `FunctionEntry prog_main_name` an ordinary activation. Discharging it from actual `compile_prog` output, rather than assuming it forever, is what closes the static side of the simulation.

```
theorem procs_embedded_compile_prog:
  assumes wf: "wf_compile_input 𝒢 Π ps"
  shows "procs_embedded Π (compile_prog Π ps)"
  unfolding procs_embedded_def
proof (intro allI impI)
  fix p decl assume pd: "Π p = Some decl"
  let ?g = "compile_prog Π ps"
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    and mainc: "compile_proc Π prog_main_name (⦇formals = [], body = main_body Π⦈) n1
                = (n2, Emain, Kmain)"
    and intra_g: "intra ?g = Eprocs ∪ Emain" and calls_g: "calls ?g = Kprocs ∪ Kmain"
    by (rule compile_prog_intra_split)
```

```
  have srccom: "source_com (body decl)"
    using wf_compile_inputD(7)[OF wf] pd unfolding source_pi_def by blast
  have "∃m m' Ef Kf. compile_proc Π p decl m = (m', Ef, Kf)
        ∧ Ef ⊆ intra ?g ∧ Kf ⊆ calls ?g"
  proof (cases "p = prog_main_name")
    case True
    with pd wf_compile_inputD(2)[OF wf]
    have "decl = (|formals = [], body = main_body Π|)" by simp
    then show ?thesis using mainc True intra_g calls_g by blast
  next
    case False
    with pd wf_compile_inputD(10)[OF wf] have "p ∈ set ps" by auto
    from compile_procs_member[OF procs this pd] obtain m m' Ef Kf where
      cp: "compile_proc Π p decl m = (m', Ef, Kf)" and "Ef ⊆ Eprocs" "Kf ⊆ Kprocs"
      by blast
    then have "Ef ⊆ intra ?g" "Kf ⊆ calls ?g" using intra_g calls_g by auto
    with cp show ?thesis by blast
  qed
  then obtain m m' Ef Kf where cp: "compile_proc Π p decl m = (m', Ef, Kf)"
    and Esub: "Ef ⊆ intra ?g" and Ksub: "Kf ⊆ calls ?g" by blast
  from cp obtain Eb where
    cb: "compile Π p (body decl) (Statement (m + csize (body decl))) m
         = (m + csize (body decl), Statement m, Eb, Kf)"
    and Edef: "Ef = insert (FunctionEntry p, EA_Body p, Statement m)
                (insert (Statement (m + csize (body decl)), EA_Ret None p, FunctionResult p) Eb)"
    by (rule compile_procE)
  have Ebsub: "Eb ⊆ intra ?g" using Edef Esub by auto
  have ent: "(FunctionEntry p, EA_Body p, Statement m) ∈ intra ?g" using Edef Esub by auto
  have ext: "(Statement (m + csize (body decl)), EA_Ret None p, FunctionResult p) ∈ intra ?g"
    using Edef Esub by auto
  show "∃k n n' en E K. compile Π p (body decl) k n = (n', en, E, K)
        ∧ E ⊆ intra ?g ∧ K ⊆ calls ?g
        ∧ (FunctionEntry p, EA_Body p, en) ∈ intra ?g
        ∧ (k, EA_Ret None p, FunctionResult p) ∈ intra ?g
        ∧ k = Statement (n + csize (body decl))
        ∧ source_com (body decl) ∧ special_table p = None"
    using cb Ebsub Ksub ent ext srccom wf_compile_inputD(6)[OF wf pd] by blast
qed
```

The certificate is inhabited

Everything above is stated under `procs_embedded`, and `csim` relates configurations only where that certificate holds; a reader is owed one graph that satisfies it, or the theorems could all be vacuous. The smallest program — `main` with an empty body and no callees — is that witness, and it carries through to a `csim` relating a real source configuration to a real node of a real compiled graph.

```
definition witness_pi :: proc_table where
  "witness_pi p = (if p = STR ''main'' then Some (|formals = [], body = SKIP ()|) else None)"

abbreviation witness_cfg :: cfg where
  "witness_cfg ≡ compile_prog witness_pi []"

lemma main_body_witness [simp]: "main_body witness_pi = SKIP ()"
  by (simp add: main_body_def witness_pi_def prog_main_name_def)

lemma witness_wf: "wf_compile_input (λ_. False) witness_pi []"
  by (auto simp: wf_compile_input_def wf_source_program_def witness_pi_def
        main_body_def prog_main_name_def
        wf_proc_decl_def reserved_ret_var_def ret_var_def
        special_table_def special_pname_nondet_int_def
        special_pname_min_def special_pname_max_def)

theorem procs_embedded_witness: "procs_embedded witness_pi witness_cfg"
  by (rule procs_embedded_compile_prog[OF witness_wf])

theorem csim_witness:
  obtains v where "witness_pi, witness_cfg, STR ''main'' ⊢ (SKIP (), s, []) ≈ (v, s, [])"
proof -
  have decl: "witness_pi (STR ''main'') = Some (|formals = [], body = SKIP ()|)"
    by (simp add: witness_pi_def)
  obtain k n en where
    cacc: "compiled_at witness_pi witness_cfg (STR ''main'')
           (body (|formals = [], body = SKIP ()|)) k n"
      and ctrl: "control_at witness_pi (STR ''main'')
           (body (|formals = [], body = SKIP ()|)) k n (body (|formals = [], body = SKIP ()|)) en"
    by (rule procs_embedded_activation[OF procs_embedded_witness decl])
```

```
  have "witness_pi, witness_cfg, STR ''main'' ⊢ (SKIP (), s, []) ≈ (en, s, [])"
    by (rule csim.Base[OF ctrl[simplified] cacc[simplified]])
  then show ?thesis ..
qed

end
```

5.8 Source_Progress

```
theory Source_Progress
  imports Simulation_Preservation
begin
```

Reachable source configurations are never stuck

A stuck source configuration would silently cut executions short: every later program point would be unreachable under `pstep`, so an unreachability verdict could hold for the wrong reason. Progress rules this out for accepted programs: a reachable configuration either has finished — `SKIP` with no caller frame left — or can take a step.

The shapes that could get stuck are the runtime markers: a bare `Unwind`, and a `Restore` or `Seq (Unwind j) (Restore i)` with no frame to pop. `csim` already pairs every `Restore` with its frame and confines `Unwind` to the returning phase, so it is reused here as the invariant; the only additional fact is that call sites are well formed.

Source commands

```
lemma pstep_seq_after_lift:
  "𝒢, Π ⊢ (w, s, frs) →p (w', s', frs')
   ⟹ 𝒢, Π ⊢ (seq_after w afters, s, frs) →p (seq_after w' afters, s', frs')"
  by (induction afters arbitrary: w w') auto
```

A well-formed source command is `SKIP` or steps, at any frame stack. A `Return` steps too; whether its `Unwind` can later pop a frame is the simulation's concern, not this lemma's.

```
lemma wf_source_com_progress:
  assumes distinct_formals: "⋀p decl. Π p = Some decl ⟹ distinct (formals decl)"
      and "wf_source_com Π c"
  shows "is_SKIP c ∨ (∃cfg'. 𝒢, Π ⊢ (c, s, frs) →p cfg')"
  using assms(2)
proof (induction c)
  case (Seq c1 c2)
  from Seq.prems have "wf_source_com Π c1" by simp
  from Seq.IH(1)[OF this] show ?case
  proof
    assume "is_SKIP c1"
    then obtain j where "c1 = SKIP j" by auto
    then show ?thesis by blast
  next
    assume "∃cfg'. 𝒢, Π ⊢ (c1, s, frs) →p cfg'"
    then obtain c1' s' frs' where "𝒢, Π ⊢ (c1, s, frs) →p (c1', s', frs')" by auto
    then show ?thesis by blast
  qed
next
  case (If i b c1 c2)
  show ?case by (cases "truthy (⟦b⟧e s)") blast+
next
  case (Return i e)
  show ?case by (cases e) blast+
next
  case (Call i dst p actuals)
  show ?case
  proof (cases "special_table p")
    case None
    with Call.prems obtain decl where
      decl: "Π p = Some decl" and arity: "length actuals = length (formals decl)"
      by (auto split: option.splits)
    show ?thesis
      using pstep_Call[where Π = Π and p = p, OF decl arity distinct_formals[OF decl]]
      by blast
  next
    case (Some desc)
    with Call.prems obtain sc x where
      sc: "classify_special desc actuals = Some sc" and dst: "dst = Some x"
      by (auto split: option.splits)
    obtain v where "special_result sc s v" using special_result_ex by blast
    with Some sc show ?thesis unfolding dst by blast
  qed
qed auto
```

```
lemma control_at_wf_source_com:
  "control_at Π p c0 k n r v ⟹ wf_source_com Π c0 ⟹ wf_source_com Π r"
  by (induction rule: control_at.induct) auto
```

The frame-pop phase

A configuration unwinding a finished frame keeps stepping until it is `Unwind` (`unwinding_progress`); a `Returning` one then pops its frame.

```
lemma unwinding_progress:
  "unwinding u ⟹ (⋀j. u ≠ Unwind j) ⟹ ∃u'. 𝒢, Π ⊢ (u, s, frs) →p (u', s, frs)"
proof (induction u)
  case (Seq u1 c2)
  then have u1: "unwinding u1" and c2: "¬ is_Restore c2" by simp_all
  show ?case
  proof (cases "∃j. u1 = Unwind j")
    case True
    then obtain j where "u1 = Unwind j" by blast
    then show ?thesis using c2 by blast
  next
    case False
    then have "u1 ≠ Unwind j" for j by blast
    from Seq.IH(1)[OF u1 this] obtain u1' where "𝒢, Π ⊢ (u1, s, frs) →p (u1', s, frs)" ..
    then show ?thesis by blast
  qed
next
  case (Unwind x)
  then show ?case by blast
qed simp_all
```

The frame a `Returning` configuration carries is what lets it pop.

```
lemma pop_ready_progress:
  assumes "pop_ready w"
  shows "∃cfg'. 𝒢, Π ⊢ (w, s, Frame fr dst # frs) →p cfg'"
  using assms
proof (cases w rule: pop_ready.cases)
  case 1
  then show ?thesis by blast
next
  case (2 u)
  with assms have uw: "unwinding u" by simp
  show ?thesis
  proof (cases "∃j. u = Unwind j")
    case True
    then obtain j where "u = Unwind j" by blast
    then show ?thesis using 2 by blast
  next
    case False
    then have "u ≠ Unwind j" for j by blast
    from unwinding_progress[OF uw this] obtain u' where
      "𝒢, Π ⊢ (u, s, Frame fr dst # frs) →p (u', s, Frame fr dst # frs)" ..
    then show ?thesis using 2 by blast
  qed
qed (use assms in simp_all)
```

Progress under the simulation relation

Only a `Base` activation can have finished: `Nested` and `Returning` both carry a caller frame and a pending `Restore` that will pop it. No graph fact is used, so the relation serves here purely as a shape invariant.

```
lemma csim_progress_cfg:
  assumes "Π, g, ϱ ⊢ cfg ≈ dg"
      and wf_decls: "⋀p decl. Π p = Some decl ⟹ wf_proc_decl 𝒢 Π decl"
  shows "(is_SKIP (fst cfg) ∧ snd (snd cfg) = []) ∨ (∃cfg'. 𝒢, Π ⊢ cfg →p cfg')"
  using assms(1)
proof (induction rule: csim.induct)
  case (Base p body_p k n c v s)
  from Base.hyps(2) obtain decl where decl: "Π p = Some decl" and body_p: "body_p = body decl"
    by (rule compiled_at_decl)
  have "wf_source_com Π body_p"
    using wf_decls[OF decl] body_p by (simp add: wf_proc_decl_def)
  from control_at_wf_source_com[OF Base.hyps(1) this]
  have "wf_source_com Π c" .
  from wf_source_com_progress[OF _ this, of 𝒢 s "[]"] wf_decls
  show ?case by (auto simp: wf_proc_decl_def)
next
  case (Nested q inner s frs v stk pc body_pc kc nc i afters cont dst args site caller)
  let ?F = "[Frame caller dst]"
  have "∃w'. 𝒢, Π ⊢ (Seq inner (Restore i), s, frs @ ?F) →p w'"
    using Nested.IH
```

```
  proof
    assume "is_SKIP (fst (inner, s, frs)) ∧ snd (snd (inner, s, frs)) = []"
    then obtain j where "inner = SKIP j" by auto
    then show ?thesis by blast
  next
    assume "∃cfg'. 𝒢, Π ⊢ (inner, s, frs) →p cfg'"
    then obtain i' s' f' where "𝒢, Π ⊢ (inner, s, frs) →p (i', s', f')" by auto
    from pstep_frame_extend[OF this, of ?F]
    show ?thesis by blast
  qed
  then obtain w' s' f' where "𝒢, Π ⊢ (Seq inner (Restore i), s, frs @ ?F) →p (w', s', f')"
    by auto
  from pstep_seq_after_lift[OF this, of afters] show ?case by auto
next
  case (Returning w i pc body_pc kc nc afters cont dst p args site callee caller)
  from pop_ready_progress[OF Returning.hyps(1), of 𝒢 Π callee caller dst "[]"]
  obtain w' s' f' where "𝒢, Π ⊢ (w, callee, [Frame caller dst]) →p (w', s', f')"
    by auto
  from pstep_seq_after_lift[OF this, of afters] show ?case by auto
qed

theorem csim_progress:
  assumes "Π, g, ϱ ⊢ (c, s, frs) ≈ dg"
      and "⋀p decl. Π p = Some decl ⟹ wf_proc_decl 𝒢 Π decl"
  shows "(is_SKIP c ∧ frs = []) ∨ (∃cfg'. 𝒢, Π ⊢ (c, s, frs) →p cfg')"
  using csim_progress_cfg[OF assms] by simp
```

Progress for accepted programs

The premise is source well-formedness alone; no compiled graph is involved. `csim` still needs some graph with every body embedded, and the complete graph — every edge present — embeds anything, so it stands in for the compiler's output. The finished shape is exactly `(SKIP, s, [])`: `main` contains no `Return` (`no_return`), so a top-level `Unwind` never arises.

```
theorem source_progress:
  assumes wf: "wf_source_program 𝒢 Π"
      and run: "𝒢, Π ⊢ (main_body Π, s0, []) →p* (c, s, frs)"
  shows "(is_SKIP c ∧ frs = []) ∨ (∃cfg'. 𝒢, Π ⊢ (c, s, frs) →p cfg')"
proof -
  define g :: cfg where
    "g = (|intra = UNIV, calls = UNIV, cfg_entry = FunctionEntry prog_main_name|)"
  have pc: "procs_embedded Π g"
    unfolding procs_embedded_def
  proof (intro allI impI)
    fix p decl assume pd: "Π p = Some decl"
    obtain n' en E K where
      cb: "compile Π p (body decl) (Statement (0 + csize (body decl))) 0 = (n', en, E, K)"
      by (metis prod_cases4)
    have "source_com (body decl)" "special_table p = None"
      using wf_source_programD(6,7)[OF wf] pd by (auto simp: source_pi_def)
    with cb show "∃k n n' en E K. compile Π p (body decl) k n = (n', en, E, K)
          ∧ E ⊆ intra g ∧ K ⊆ calls g
          ∧ (FunctionEntry p, EA_Body p, en) ∈ intra g
          ∧ (k, EA_Ret None p, FunctionResult p) ∈ intra g
          ∧ k = Statement (n + csize (body decl))
          ∧ source_com (body decl) ∧ special_table p = None"
      by (intro exI[of _ "Statement (0 + csize (body decl))"] exI[of _ "0 :: nat"] exI[of _ n']
            exI[of _ en] exI[of _ E] exI[of _ K])
         (simp add: g_def)
  qed
  have main_decl: "Π prog_main_name = Some (|formals = [], body = main_body Π|)"
    by (rule wf_source_programD(2)[OF wf])
  obtain k n en where
    cacc: "compiled_at Π g prog_main_name (body (|formals = [], body = main_body Π|)) k n"
    and ctrl: "control_at Π prog_main_name (body (|formals = [], body = main_body Π|)) k n
                (body (|formals = [], body = main_body Π|)) en"
    by (rule procs_embedded_activation[OF pc main_decl])
  have base: "Π, g, prog_main_name ⊢ (main_body Π, s0, []) ≈ (en, s0, [])"
    using csim.Base[OF ctrl cacc] by simp
  have safe: "return_safe (main_body Π)"
    using wf_source_programD(8,4)[OF wf] by (rule return_safe_if_no_return)
  from csim_star[OF base pc safe run] obtain dg where "Π, g, prog_main_name ⊢ (c, s, frs) ≈ dg"
    by blast
  then show ?thesis
    by (rule csim_progress) (rule wf_source_programD(5)[OF wf])
qed
```

```
end
```

5.9 Procedure_Ownership

```
theory Procedure_Ownership
  imports Compile_Invariants "Voblint_CFG.Activation_Trace_Def"
begin
```

Each procedure owns a disjoint block of nodes

An activation's own local `path_of` stays inside one compiled procedure fragment. A *fragment* of procedure `r` compiled over `[m, m')` is `FunctionEntry r`, `FunctionResult r`, and the `Statement` nodes in that counter range. Every intra edge keeps both endpoints in one fragment, every call edge keeps its source and continuation there (the callee `FunctionEntry` is the only cross-fragment endpoint), and distinct fragments are node-disjoint, so a single activation entered at `FunctionEntry p` reaches `FunctionResult q` only for `p = q`.

```
definition pfn :: "pname ⇒ nat ⇒ nat ⇒ cfg_node set" where
  "pfn r m m' = insert (FunctionEntry r) (insert (FunctionResult r) (Statement ` {m..<m'}))"

lemma pfn_disjoint:
  "p ≠ r ⟹ mp' ≤ m ∨ m' ≤ mp ⟹ x ∈ pfn p mp mp' ⟹ x ∉ pfn r m m'"
  by (auto simp: pfn_def)
```

Edges keep endpoints in one fragment

At command level both endpoints lie in the fragment's own node set or are its continuation (`compile_E_shape`). At procedure level the continuation is the epilogue node, which is inside the fragment's range, so the extra alternative collapses.

```
lemma compile_K_shape:
  "compile Π p c k n = (n', en, E, K) ⟹ (u, ce, tgt, af) ∈ K
   ⟹ (∃j. u = Statement j) ∧ (∃q. tgt = FunctionEntry q)
       ∧ (af = k ∨ (∃j. af = Statement j))"
proof (induction c arbitrary: k n n' en E K)
  case (Seq c1 c2)
  from Seq.prems(1) obtain n1 E1 K1 n2 E2 K2 where
    c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, Statement n, E1, K1)"
    and c2: "compile Π p c2 k (n + csize c1) = (n2, Statement (n + csize c1), E2, K2)"
    and K: "K = K1 ∪ K2"
    by (rule compile_SeqE)
  from Seq.prems(2) K show ?case using Seq.IH(1)[OF c1] Seq.IH(2)[OF c2] by auto
next
  case (If i b c1 c2)
  from If.prems(1) obtain n1 E1 K1 n2 E2 K2 where
    c1: "compile Π p c1 k (Suc n) = (n1, Statement (Suc n), E1, K1)"
    and c2: "compile Π p c2 k (Suc n + csize c1)
               = (n2, Statement (Suc n + csize c1), E2, K2)"
    and K: "K = K1 ∪ K2"
    by (rule compile_IfE)
  from If.prems(2) K show ?case using If.IH(1)[OF c1] If.IH(2)[OF c2] by auto
next
  case (While i b c)
  from While.prems(1) obtain n1 E1 K1 where
    c1: "compile Π p c (Statement n) (Suc n) = (n1, Statement (Suc n), E1, K1)"
    and K: "K = K1"
    by (rule compile_WhileE)
  from While.prems(2) K show ?case using While.IH[OF c1] by auto
qed (auto split: if_splits option.splits)

lemma compile_intra_pfn:
  assumes cp: "compile Π p c k n = (n', en, E, K)" and e: "(u, a, v) ∈ E"
  shows "u ∈ insert k (pfn p n n') ∧ v ∈ insert k (pfn p n n')"
proof -
  have rng: "frag_stmts E K ⊆ {n..<n'} ∪ kstmt k" using compile_frag_stmts_range[OF cp] .
  from compile_E_shape[OF cp e] obtain ku where u: "u = Statement ku"
    and vshape: "v = k ∨ v = FunctionResult p ∨ (∃kv. v = Statement kv)" by blast
  have "ku ∈ frag_stmts E K" using frag_stmts_E_srcI[of ku a v E] e u by simp
  hence uin: "u ∈ insert k (pfn p n n')" using rng u by (auto simp: pfn_def)
  have vin: "v ∈ insert k (pfn p n n')"
  proof (cases "v = k ∨ v = FunctionResult p")
    case True then show ?thesis by (auto simp: pfn_def)
  next
    case False
    then obtain kv where v: "v = Statement kv" using vshape by blast
    have "kv ∈ frag_stmts E K" using frag_stmts_E_tgtI[of u a kv E] e v by simp
    then show ?thesis using rng v False by (auto simp: pfn_def)
  qed
```

```
  from uin vin show ?thesis ..
qed

lemma compile_calls_pfn:
  assumes cp: "compile Π p c k n = (n', en, E, K)" and e: "(u, ce, tgt, af) ∈ K"
  shows "u ∈ insert k (pfn p n n') ∧ af ∈ insert k (pfn p n n')"
proof -
  have rng: "frag_stmts E K ⊆ {n..<n'} ∪ kstmt k" using compile_frag_stmts_range[OF cp] .
  from compile_K_shape[OF cp e] obtain ku where u: "u = Statement ku"
    and afshape: "af = k ∨ (∃kaf. af = Statement kaf)" by blast
  have "ku ∈ frag_stmts E K" using frag_stmts_K_srcI[of ku ce tgt af K E] e u by simp
  hence uin: "u ∈ insert k (pfn p n n')" using rng u by (auto simp: pfn_def)
  have "af ∈ insert k (pfn p n n')"
  proof (cases "af = k")
    case True then show ?thesis by simp
  next
    case False
    then obtain kaf where af: "af = Statement kaf" using afshape by blast
    have "kaf ∈ frag_stmts E K" using frag_stmts_K_tgtI[of u ce tgt kaf K E] e af by simp
    then show ?thesis using rng af False by (auto simp: pfn_def)
  qed
  with uin show ?thesis ..
qed
```

The entry wiring edge targets the first statement index of the fragment, which is what identifies a fragment by its start counter alone.

```
lemma compile_proc_entry_edge:
  assumes cp: "compile_proc Π p decl n = (n', E, K)"
  shows "(FunctionEntry p, EA_Body p, Statement n) ∈ E"
  using cp by auto

lemma compile_proc_intra_pfn:
  assumes cp: "compile_proc Π r decl n = (n', E, K)" and e: "(u, a, v) ∈ E"
  shows "u ∈ pfn r n n' ∧ v ∈ pfn r n n'"
proof -
  from cp obtain Eb where
    cb: "compile Π r (body decl) (Statement (n + csize (body decl))) n
           = (n + csize (body decl), Statement n, Eb, K)"
    and E: "E = insert (FunctionEntry r, EA_Body r, Statement n)
              (insert (Statement (n + csize (body decl)), EA_Ret None r, FunctionResult r) Eb)"
    and n': "n' = Suc (n + csize (body decl))"
    by (rule compile_procE)
  have kin: "Statement (n + csize (body decl)) ∈ pfn r n n'"
    using n' by (auto simp: pfn_def)
  have entryin: "Statement n ∈ pfn r n n'"
    using n' csize_pos[of "body decl"] by (auto simp: pfn_def)
  have inner: "insert (Statement (n + csize (body decl)))
                 (pfn r n (n + csize (body decl))) ⊆ pfn r n n'"
    using n' by (auto simp: pfn_def)
  consider "(u, a, v) = (FunctionEntry r, EA_Body r, Statement n)"
    | "(u, a, v) = (Statement (n + csize (body decl)), EA_Ret None r, FunctionResult r)"
    | "(u, a, v) ∈ Eb"
    using e E by (auto split: if_splits)
  then show ?thesis
  proof cases
    case 1 then show ?thesis using entryin by (auto simp: pfn_def)
  next
    case 2 then show ?thesis using kin by (auto simp: pfn_def)
  next
    case 3 then show ?thesis using compile_intra_pfn[OF cb] inner by blast
  qed
qed

lemma compile_proc_calls_pfn:
  assumes cp: "compile_proc Π r decl n = (n', E, K)" and e: "(u, ce, tgt, af) ∈ K"
  shows "u ∈ pfn r n n' ∧ af ∈ pfn r n n'"
proof -
  from cp obtain Eb where
    cb: "compile Π r (body decl) (Statement (n + csize (body decl))) n
           = (n + csize (body decl), Statement n, Eb, K)"
    and n': "n' = Suc (n + csize (body decl))"
    by (rule compile_procE)
  have inner: "insert (Statement (n + csize (body decl)))
                 (pfn r n (n + csize (body decl))) ⊆ pfn r n n'"
    using n' by (auto simp: pfn_def)
  show ?thesis using compile_calls_pfn[OF cb e] inner by blast
```

qed

A fragment of another procedure over a disjoint counter range sources none of its edges in `pfn r m m'`.

```
lemma compile_proc_src_notin_pfn:
  assumes cp: "compile_proc Π p decl mp = (mp', Ep, Kp)"
    and neq: "p ≠ r" and disjoint: "mp' ≤ m ∨ m' ≤ mp"
  shows "(u, a, v) ∈ Ep ⟹ u ∉ pfn r m m'"
    and "(u, ce, tgt, af) ∈ Kp ⟹ u ∉ pfn r m m'"
  using compile_proc_intra_pfn[OF cp] compile_proc_calls_pfn[OF cp] pfn_disjoint[OF neq disjoint]
  by blast+

lemma compile_procs_intra_src_notin_pfn:
  assumes cps: "compile_procs Π ps n = (n', E, K)" and e: "(u, a, v) ∈ E"
    and absent: "r ∉ set ps" and disjoint: "n' ≤ m ∨ m' ≤ n"
  shows "u ∉ pfn r m m'"
proof -
  from compile_procs_intra_origin[OF cps e] obtain p decl mp mp' Ep Kp
    where "p ∈ set ps" and cp: "compile_proc Π p decl mp = (mp', Ep, Kp)"
      and "n ≤ mp" "mp' ≤ n'" and ep: "(u, a, v) ∈ Ep" by blast
  have "p ≠ r" using ‹p ∈ set ps› absent by blast
  moreover have "mp' ≤ m ∨ m' ≤ mp" using ‹n ≤ mp› ‹mp' ≤ n'› disjoint by auto
  ultimately show ?thesis using compile_proc_src_notin_pfn(1)[OF cp _ _ ep] by blast
qed

lemma compile_procs_calls_src_notin_pfn:
  assumes cps: "compile_procs Π ps n = (n', E, K)" and e: "(u, ce, tgt, af) ∈ K"
    and absent: "r ∉ set ps" and disjoint: "n' ≤ m ∨ m' ≤ n"
  shows "u ∉ pfn r m m'"
proof -
  from compile_procs_calls_origin[OF cps e] obtain p decl mp mp' Ep Kp
    where "p ∈ set ps" and cp: "compile_proc Π p decl mp = (mp', Ep, Kp)"
      and "n ≤ mp" "mp' ≤ n'" and ep: "(u, ce, tgt, af) ∈ Kp" by blast
  have "p ≠ r" using ‹p ∈ set ps› absent by blast
  moreover have "mp' ≤ m ∨ m' ≤ mp" using ‹n ≤ mp› ‹mp' ≤ n'› disjoint by auto
  ultimately show ?thesis using compile_proc_src_notin_pfn(2)[OF cp _ _ ep] by blast
qed
```

A procedure entry has a unique out-edge

Within a compiled procedure only the entry wiring edge sources from `FunctionEntry` (bodies source from `Statement` nodes by `compile_E_shape`, the exit wiring from the exit `Statement`), so the source name is the procedure and the target is unique.

```
lemma compile_proc_entry_mem:
  assumes cp: "compile_proc Π q decl n = (n', E, K)" and e: "(FunctionEntry r, a, v) ∈ E"
  shows "r = q"
proof -
  from cp obtain Eb where
    cb: "compile Π q (body decl) (Statement (n + csize (body decl))) n
          = (n + csize (body decl), Statement n, Eb, K)"
    and E: "E = insert (FunctionEntry q, EA_Body q, Statement n)
              (insert (Statement (n + csize (body decl)), EA_Ret None q, FunctionResult q) Eb)"
    by (rule compile_procE)
  have "⋀aa vv. (FunctionEntry r, aa, vv) ∉ Eb" using compile_E_shape[OF cb] by blast
  then show ?thesis using e E by (auto split: if_splits)
qed

lemma compile_proc_entry_unique:
  assumes cp: "compile_proc Π q decl n = (n', E, K)"
    and e1: "(FunctionEntry r, a1, v1) ∈ E" and e2: "(FunctionEntry r, a2, v2) ∈ E"
  shows "v1 = v2"
proof -
  from cp obtain Eb where
    cb: "compile Π q (body decl) (Statement (n + csize (body decl))) n
          = (n + csize (body decl), Statement n, Eb, K)"
    and E: "E = insert (FunctionEntry q, EA_Body q, Statement n)
              (insert (Statement (n + csize (body decl)), EA_Ret None q, FunctionResult q) Eb)"
    by (rule compile_procE)
  have nb: "⋀aa vv. (FunctionEntry r, aa, vv) ∉ Eb" using compile_E_shape[OF cb] by blast
  from e1 E nb have "v1 = Statement n" by (auto split: if_splits)
  moreover from e2 E nb have "v2 = Statement n" by (auto split: if_splits)
  ultimately show ?thesis by simp
qed

lemma compile_procs_entry_mem:
```

```
  "compile_procs Π ps n = (n', E, K) ⟹ (FunctionEntry r, a, v) ∈ E ⟹ r ∈ set ps"
  using compile_procs_intra_origin compile_proc_entry_mem by blast
```

Under `distinct ps` a member's fragment owns every edge of the pass that sources in the member's nodes: the other members are compiled over disjoint counter ranges, and their edges stay in their own fragments.

```
lemma compile_procs_member_frag:
  assumes "compile_procs Π ps n = (n', E, K)" and "distinct ps"
    and "r ∈ set ps" and "Π r = Some decl"
  shows "∃m m' Er Kr. compile_proc Π r decl m = (m', Er, Kr) ∧ n ≤ m ∧ m' ≤ n'
           ∧ Er ⊆ E ∧ Kr ⊆ K
           ∧ (∀u a v. (u, a, v) ∈ E ⟶ u ∈ pfn r m m' ⟶ (u, a, v) ∈ Er)
           ∧ (∀u ce tgt af. (u, ce, tgt, af) ∈ K ⟶ u ∈ pfn r m m'
                ⟶ (u, ce, tgt, af) ∈ Kr)"
  using assms
proof (induction ps arbitrary: n n' E K)
  case Nil then show ?case by simp
next
  case (Cons q qs)
  show ?case
  proof (cases "Π q")
    case None
    with Cons.prems have "compile_procs Π qs n = (n', E, K)" "distinct qs" "r ∈ set qs"
      by auto
    from Cons.IH[OF this Cons.prems(4)] show ?thesis .
  next
    case (Some dq)
    with Cons.prems(1) obtain n1 E0 K0 n2 E' K' where
       cp0: "compile_proc Π q dq n = (n1, E0, K0)"
      and rest: "compile_procs Π qs n1 = (n2, E', K')"
      and n': "n' = n2" and E: "E = E0 ∪ E'" and K: "K = K0 ∪ K'"
      by (auto split: prod.splits)
    have qnotin: "q ∉ set qs" and dqs: "distinct qs" using Cons.prems(2) by simp_all
    show ?thesis
    proof (cases "r = q")
      case True
      with Cons.prems(4) Some cp0 have cpr: "compile_proc Π r decl n = (n1, E0, K0)" by simp
      have rnotin: "r ∉ set qs" using qnotin True by simp
      have tailE: "⋀u a v. (u, a, v) ∈ E' ⟹ u ∉ pfn r n n1"
        using compile_procs_intra_src_notin_pfn[OF rest _ rnotin] by blast
      have tailK: "⋀u ce tgt af. (u, ce, tgt, af) ∈ K' ⟹ u ∉ pfn r n n1"
        using compile_procs_calls_src_notin_pfn[OF rest _ rnotin] by blast
      have "∀u a v. (u, a, v) ∈ E ⟶ u ∈ pfn r n n1 ⟶ (u, a, v) ∈ E0"
        using E tailE by blast
      moreover have "∀u ce tgt af. (u, ce, tgt, af) ∈ K ⟶ u ∈ pfn r n n1
                       ⟶ (u, ce, tgt, af) ∈ K0"
        using K tailK by blast
      moreover have "n1 ≤ n'" using compile_procs_counter_mono[OF rest] n' by simp
      moreover have "E0 ⊆ E" "K0 ⊆ K" using E K by auto
      ultimately show ?thesis using cpr by blast
    next
      case False
      have rqs: "r ∈ set qs" using Cons.prems(3) False by simp
      from Cons.IH[OF rest dqs rqs Cons.prems(4)] obtain m m' Er Kr where
         cpr: "compile_proc Π r decl m = (m', Er, Kr)" and bnds: "n1 ≤ m" "m' ≤ n2"
        and sub: "Er ⊆ E'" "Kr ⊆ K'"
        and ownE: "∀u a v. (u, a, v) ∈ E' ⟶ u ∈ pfn r m m' ⟶ (u, a, v) ∈ Er"
        and ownK: "∀u ce tgt af. (u, ce, tgt, af) ∈ K' ⟶ u ∈ pfn r m m'
                     ⟶ (u, ce, tgt, af) ∈ Kr"
        by blast
      have qr: "q ≠ r" using False by simp
      have headE: "⋀u a v. (u, a, v) ∈ E0 ⟹ u ∉ pfn r m m'"
        and headK: "⋀u ce tgt af. (u, ce, tgt, af) ∈ K0 ⟹ u ∉ pfn r m m'"
        using compile_proc_src_notin_pfn[OF cp0 qr] bnds(1) by blast+
      have "∀u a v. (u, a, v) ∈ E ⟶ u ∈ pfn r m m' ⟶ (u, a, v) ∈ Er"
        using E ownE headE by blast
      moreover have "∀u ce tgt af. (u, ce, tgt, af) ∈ K ⟶ u ∈ pfn r m m'
                       ⟶ (u, ce, tgt, af) ∈ Kr"
        using K ownK headK by blast
      moreover have "n ≤ m" using compile_proc_counter_mono[OF cp0] bnds(1) by simp
      moreover have "m' ≤ n'" using bnds(2) n' by simp
      moreover have "Er ⊆ E" "Kr ⊆ K" using sub E K by auto
      ultimately show ?thesis using cpr by blast
    qed
  qed
qed
```

```
lemma compile_procs_entry_unique:
  assumes cps: "compile_procs Π ps n = (n', E, K)" and distinct: "distinct ps"
    and e1: "(FunctionEntry r, a1, v1) ∈ E" and e2: "(FunctionEntry r, a2, v2) ∈ E"
  shows "v1 = v2"
proof -
  from compile_procs_intra_origin[OF cps e1] obtain decl
    where rin: "r ∈ set ps" and decl: "Π r = Some decl"
    using compile_proc_entry_mem by metis
  from compile_procs_member_frag[OF cps distinct rin decl] obtain m m' Er Kr where
      cp: "compile_proc Π r decl m = (m', Er, Kr)"
    and own: "∀u a v. (u, a, v) ∈ E ⟶ u ∈ pfn r m m' ⟶ (u, a, v) ∈ Er" by blast
  have "(FunctionEntry r, a1, v1) ∈ Er" "(FunctionEntry r, a2, v2) ∈ Er"
    using own e1 e2 by (auto simp: pfn_def)
  then show ?thesis by (rule compile_proc_entry_unique[OF cp])
qed
```

In a whole compiled program `FunctionEntry r` has a unique out-target: `main` and the callees are node-disjoint (`prog_main_name ∉ set ps`), and each contributes a single entry edge.

```
lemma compile_prog_entry_out_unique:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and e1: "(FunctionEntry r, a1, v1) ∈ intra (compile_prog Π ps)"
    and e2: "(FunctionEntry r, a2, v2) ∈ intra (compile_prog Π ps)"
  shows "v1 = v2"
proof -
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    and mainc: "compile_proc Π prog_main_name (|formals = [], body = (main_body Π)|) n1
                = (n2, Emain, Kmain)"
    and EI: "intra (compile_prog Π ps) = Eprocs ∪ Emain"
    by (rule compile_prog_intra_split)
  have main_mem: "⋀b w. (FunctionEntry r, b, w) ∈ Emain ⟹ r = prog_main_name"
    using compile_proc_entry_mem[OF mainc] by simp
  have procs_mem: "⋀b w. (FunctionEntry r, b, w) ∈ Eprocs ⟹ r ∈ set ps"
    using compile_procs_entry_mem[OF procs] by simp
  show ?thesis
  proof (cases "r = prog_main_name")
    case True
    have "(FunctionEntry r, a1, v1) ∈ Emain" "(FunctionEntry r, a2, v2) ∈ Emain"
      using e1 e2 EI procs_mem True wf_compile_inputD(11)[OF wf] by auto
    then show ?thesis using compile_proc_entry_unique[OF mainc] by blast
  next
    case False
    have "(FunctionEntry r, a1, v1) ∈ Eprocs" "(FunctionEntry r, a2, v2) ∈ Eprocs"
      using e1 e2 EI main_mem False by auto
    then show ?thesis using compile_procs_entry_unique[OF procs wf_compile_inputD(9)[OF wf]]
      by blast
  qed
qed
```

Whole-program edge shapes

Two shape facts a consumer of a compiled graph may use without redoing the callee/`main` split: a call edge always leaves a `Statement` node, and an entry edge names a procedure the table declares. The first is a range property each pass has on its own; the second needs well-formed input, which is what ties the compiled procedure list back to the table.

```
lemma compile_prog_calls_source_stmt:
  assumes "(u, ce, tgt, af) ∈ calls (compile_prog Π ps)"
  shows "∃k. u = Statement k"
proof -
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    and mainc: "compile_proc Π prog_main_name (|formals = [], body = (main_body Π)|) n1
                = (n2, Emain, Kmain)"
    and KC: "calls (compile_prog Π ps) = Kprocs ∪ Kmain"
    by (rule compile_prog_intra_split)
  from assms KC show ?thesis
    using compile_procs_calls_source_range[OF procs] compile_proc_calls_source_range[OF mainc]
    by blast
qed

lemma compile_prog_entry_declared:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and e: "(FunctionEntry p, a, v) ∈ intra (compile_prog Π ps)"
  shows "∃d. Π p = Some d"
```

```
proof -
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    and mainc: "compile_proc Π prog_main_name (|formals = [], body = (main_body Π)|) n1
                = (n2, Emain, Kmain)"
    and EI: "intra (compile_prog Π ps) = Eprocs ∪ Emain"
    by (rule compile_prog_intra_split)
  from e EI consider "(FunctionEntry p, a, v) ∈ Eprocs" | "(FunctionEntry p, a, v) ∈ Emain"
    by auto
  then show ?thesis
  proof cases
    case 1
    then have "p ∈ set ps" using compile_procs_entry_mem[OF procs] by blast
    with wf_compile_inputD(10)[OF wf] show ?thesis by auto
  next
    case 2
    then have "p = prog_main_name" by (rule compile_proc_entry_mem[OF mainc])
    with wf_compile_inputD(2)[OF wf] show ?thesis by blast
  qed
qed
```

Edges stay in the activation fragment

The fragment a procedure's entry wiring identifies owns every edge of the program that sources in its nodes. The callee pass and `main` occupy disjoint counter ranges, and inside the pass `compile_procs_member_frag` settles ownership; the entry edge pins the start counter through `compile_prog_entry_out_unique`.

```
lemma compile_prog_entry_frag:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and decl: "Π r = Some d"
    and cb: "compile_proc Π r d m = (m', Ep, Kp)"
    and ent: "(FunctionEntry r, EA_Body r, Statement m) ∈ intra (compile_prog Π ps)"
  shows "(∀u a v. (u, a, v) ∈ intra (compile_prog Π ps) ⟶ u ∈ pfn r m m'
            ⟶ (u, a, v) ∈ Ep)
       ∧ (∀u ce tgt af. (u, ce, tgt, af) ∈ calls (compile_prog Π ps)
            ⟶ u ∈ pfn r m m' ⟶ (u, ce, tgt, af) ∈ Kp)"
proof -
  let ?g = "compile_prog Π ps"
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    and mainc: "compile_proc Π prog_main_name (|formals = [], body = (main_body Π)|) n1
                = (n2, Emain, Kmain)"
    and EI: "intra ?g = Eprocs ∪ Emain" and KC: "calls ?g = Kprocs ∪ Kmain"
    by (rule compile_prog_intra_split)
  have distinct: "distinct ps" and setps: "set ps = {p. Π p ≠ None} - {prog_main_name}"
    and main_notin: "prog_main_name ∉ set ps"
    using wf_compile_inputD(9,10,11)[OF wf] by blast+
  show ?thesis
  proof (cases "r = prog_main_name")
    case True
    have dd: "d = (|formals = [], body = (main_body Π)|)"
      using decl True wf_compile_inputD(2)[OF wf] by simp
    have entmain: "(FunctionEntry prog_main_name, EA_Body prog_main_name, Statement n1) ∈ intra ?g"
      using EI compile_proc_entry_edge[OF mainc] by simp
    have entr: "(FunctionEntry prog_main_name, EA_Body prog_main_name, Statement m) ∈ intra ?g"
      using ent True by simp
    have mn: "m = n1" using compile_prog_entry_out_unique[OF wf entr entmain] by simp
    have mout: "m' = n2" "Ep = Emain" "Kp = Kmain" using cb mainc True dd mn by simp_all
    show ?thesis unfolding True mn mout
      using EI KC compile_procs_intra_src_notin_pfn[OF procs _ main_notin]
        compile_procs_calls_src_notin_pfn[OF procs _ main_notin] by auto
  next
    case False
    have rin: "r ∈ set ps" using decl setps False by auto
    from compile_procs_member_frag[OF procs distinct rin decl] obtain m0 m0' Er Kr where
        cpr: "compile_proc Π r d m0 = (m0', Er, Kr)" and upper: "m0' ≤ n1"
      and Ersub: "Er ⊆ Eprocs"
      and ownE: "∀u a v. (u, a, v) ∈ Eprocs ⟶ u ∈ pfn r m0 m0' ⟶ (u, a, v) ∈ Er"
      and ownK: "∀u ce tgt af. (u, ce, tgt, af) ∈ Kprocs ⟶ u ∈ pfn r m0 m0'
                  ⟶ (u, ce, tgt, af) ∈ Kr"
      by blast
    have ent0: "(FunctionEntry r, EA_Body r, Statement m0) ∈ intra ?g"
      using EI Ersub compile_proc_entry_edge[OF cpr] by auto
    have mm: "m = m0" using compile_prog_entry_out_unique[OF wf ent ent0] by simp
    have mout: "m' = m0'" "Ep = Er" "Kp = Kr" using cb cpr mm by simp_all
    have neq: "prog_main_name ≠ r" using False by simp
```

```
    show ?thesis unfolding mm mout
      using EI KC ownE ownK compile_proc_src_notin_pfn[OF mainc neq] upper by blast
  qed
qed
```

Given a genuine procedure fragment (declared body, entry wiring) and a node in it, every intra edge leaving that node, and every call leaving it, lands its same-activation successor back in the fragment.

```
lemma frag_edge_intra:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and decl: "Π r = Some d"
    and cb: "compile_proc Π r d m = (m', Ep, Kp)"
    and ent: "(FunctionEntry r, EA_Body r, Statement m) ∈ intra (compile_prog Π ps)"
    and uin: "u ∈ pfn r m m'"
    and e: "(u, a, v) ∈ intra (compile_prog Π ps)"
  shows "v ∈ pfn r m m'"
  using compile_prog_entry_frag[OF wf decl cb ent] compile_proc_intra_pfn[OF cb] uin e by blast

lemma frag_edge_calls:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and decl: "Π r = Some d"
    and cb: "compile_proc Π r d m = (m', Ep, Kp)"
    and ent: "(FunctionEntry r, EA_Body r, Statement m) ∈ intra (compile_prog Π ps)"
    and uin: "u ∈ pfn r m m'"
    and e: "(u, ce, tgt, af) ∈ calls (compile_prog Π ps)"
  shows "af ∈ pfn r m m'"
  using compile_prog_entry_frag[OF wf decl cb ent] compile_proc_calls_pfn[OF cb] uin e by blast
```

Procedure locality of a valid activation

Every declared procedure of a compiled program owns a fragment, identified by the start counter its entry wiring edge points at.

```
lemma compile_prog_proc_frag:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and decl: "Π q = Some d"
  obtains m m' Ep Kp where
    "compile_proc Π q d m = (m', Ep, Kp)"
    "(FunctionEntry q, EA_Body q, Statement m) ∈ intra (compile_prog Π ps)"
proof -
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    and mainc: "compile_proc Π prog_main_name (⦇formals = [], body = (main_body Π)⦈) n1
                = (n2, Emain, Kmain)"
    and EI: "intra (compile_prog Π ps) = Eprocs ∪ Emain"
    by (rule compile_prog_intra_split)
  show thesis
  proof (cases "q = prog_main_name")
    case True
    have dd: "d = (⦇formals = [], body = (main_body Π)⦈)"
      using decl True wf_compile_inputD(2)[OF wf] by simp
    have "(FunctionEntry q, EA_Body q, Statement n1) ∈ intra (compile_prog Π ps)"
      using EI compile_proc_entry_edge[OF mainc] True by auto
    then show ?thesis using that[of n1 n2 Emain Kmain] mainc True dd by simp
  next
    case False
    have rin: "q ∈ set ps" using decl wf_compile_inputD(10)[OF wf] False by auto
    obtain m m' Ep Kp where cp0: "compile_proc Π q d m = (m', Ep, Kp)" and sub: "Ep ⊆ Eprocs"
      using compile_procs_member[OF procs rin decl] by blast
    have "(FunctionEntry q, EA_Body q, Statement m) ∈ intra (compile_prog Π ps)"
      using EI sub compile_proc_entry_edge[OF cp0] by auto
    then show ?thesis using that cp0 by blast
  qed
qed
```

`frag_ok u`: the activation `u` either lies wholly inside a genuine procedure fragment (declared body, entry wiring, all path nodes in the fragment) or is a single-node activation stuck at an undeclared `FunctionEntry` — which can never advance and never reaches a `FunctionResult`.

```
definition frag_ok :: "'a ptable ⇒ pname list ⇒ activation_trace ⇒ bool" where
  "frag_ok Π ps u ⟷
     (∃r d m m' Ep Kp. Π r = Some d
        ∧ compile_proc Π r d m = (m', Ep, Kp)
        ∧ (FunctionEntry r, EA_Body r, Statement m) ∈ intra (compile_prog Π ps)
        ∧ fst (hd (path_of u)) = FunctionEntry r
        ∧ (∀nd ∈ fst ‘ set (path_of u). nd ∈ pfn r m m'))
```

```
  ∨ (∃p s. path_of u = [(FunctionEntry p, s)] ∧ Π p = None)"

lemma frag_okI_frag:
  assumes "Π r = Some d" "compile_proc Π r d m = (m', Ep, Kp)"
    "(FunctionEntry r, EA_Body r, Statement m) ∈ intra (compile_prog Π ps)"
    "fst (hd (path_of u)) = FunctionEntry r"
    "⋀nd. nd ∈ fst ‘ set (path_of u) ⟹ nd ∈ pfn r m m'"
  shows "frag_ok Π ps u"
  unfolding frag_ok_def using assms by blast

lemma frag_okE [elim]:
  assumes "frag_ok Π ps u"
  obtains (frag) r d m m' Ep Kp where "Π r = Some d" "compile_proc Π r d m = (m', Ep, Kp)"
    "(FunctionEntry r, EA_Body r, Statement m) ∈ intra (compile_prog Π ps)"
    "fst (hd (path_of u)) = FunctionEntry r" "∀nd ∈ fst ‘ set (path_of u). nd ∈ pfn r m m'"
  | (stub) p s where "path_of u = [(FunctionEntry p, s)]" "Π p = None"
  using assms unfolding frag_ok_def by meson

lemma sink_in_path_nodes:
  "t ∈ 𝒯_{𝒢,g,S} ⟹ sink_node t ∈ fst ‘ set (path_of t)"
  using valid_activation_trace_path_nonempty by (auto simp: sink_node_def)
```

Extending a fragment-local activation by one node keeps it fragment-local when the step to that node stays inside every genuine fragment holding its sink and no step leaves a stub. An intra step and the resumption after a call are the two such extensions.

```
lemma frag_ok_snoc:
  assumes t: "t ∈ 𝒯_{𝒢,compile_prog Π ps,S}" and ok: "frag_ok Π ps t"
    and pu: "path_of u = path_of t @ [(v, s)]"
    and stay: "⋀r d m m' Ep Kp. Π r = Some d ⟹ compile_proc Π r d m = (m', Ep, Kp) ⟹
      (FunctionEntry r, EA_Body r, Statement m) ∈ intra (compile_prog Π ps) ⟹
      sink_node t ∈ pfn r m m' ⟹ v ∈ pfn r m m'"
    and nostub: "⋀q. sink_node t = FunctionEntry q ⟹ Π q = None ⟹ False"
  shows "frag_ok Π ps u"
  using ok
proof (cases rule: frag_okE)
  case (frag r d m m' Ep Kp)
  have "sink_node t ∈ pfn r m m'" using sink_in_path_nodes[OF t] frag(5) by blast
  from stay[OF frag(1,2,3) this] have v_in: "v ∈ pfn r m m'" .
  show ?thesis
    by (rule frag_okI_frag[OF frag(1,2,3)])
       (use frag(4,5) valid_activation_trace_path_nonempty[OF t] v_in in ‹auto simp: pu hd_append›)
next
  case (stub q s0)
  have "sink_node t = FunctionEntry q" using stub(1) by (simp add: sink_node_def)
  from nostub[OF this stub(2)] show ?thesis ..
qed
```

Every activation in the caller chain of a valid activation trace is fragment-local. The property is carried over the whole `callers` chain so that the return case, which resumes the caller, can read the caller's own fragment (`caller ∈ callers callee`).

```
lemma valid_activation_trace_frag_callers:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and t: "t ∈ 𝒯_{𝒢,compile_prog Π ps,S}"
  shows "∀u ∈ callers t. frag_ok Π ps u"
proof -
  let ?g = "compile_prog Π ps"
  show ?thesis
  proof (rule caller_chain_closure)
    fix s assume "s ∈ S"
    have main_decl: "Π prog_main_name = Some (|formals = [], body = (main_body Π)|)"
      by (rule wf_compile_inputD(2)[OF wf])
    obtain m m' Ep Kp where
      cb: "compile_proc Π prog_main_name (|formals = [], body = (main_body Π)|) m = (m', Ep, Kp)"
      and ent: "(FunctionEntry prog_main_name, EA_Body prog_main_name, Statement m) ∈ intra ?g"
      by (rule compile_prog_proc_frag[OF wf main_decl])
    show "frag_ok Π ps (Root [(cfg_entry ?g, s)])"
      by (rule frag_okI_frag[OF main_decl cb ent])
         (simp_all add: pfn_def)
  next
    fix t a v s'
    assume ht: "t ∈ 𝒯_{𝒢,?g,S}" and ch: "∀u ∈ callers t. frag_ok Π ps u"
      and e: "(sink_node t, a, v) ∈ intra ?g"
    have "frag_ok Π ps t" using ch callers_refl by blast
    then show "frag_ok Π ps (extend t (v, s'))"
```

```
      by (rule frag_ok_snoc[OF ht _ _ frag_edge_intra[OF wf _ _ _ _ e]])
         (use e in ‹auto dest: compile_prog_entry_declared[OF wf]›)
  next
    fix caller dst pars args p cont
    assume "(sink_node caller, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls ?g"
    show "frag_ok Π ps (Call caller
            [(FunctionEntry p, call_enter 𝒢 (CallEdge dst pars args) (sink_store caller))])"
    proof (cases "Π p")
      case None
      then show ?thesis unfolding frag_ok_def by fastforce
    next
      case (Some d)
      obtain m m' Ep Kp where
        cb: "compile_proc Π p d m = (m', Ep, Kp)"
        and ent: "(FunctionEntry p, EA_Body p, Statement m) ∈ intra ?g"
        by (rule compile_prog_proc_frag[OF wf Some])
      show ?thesis
        by (rule frag_okI_frag[OF Some cb ent]) (simp_all add: pfn_def)
    qed
  next
    fix callee caller p dst pars args cont
    assume cvcallee: "callee ∈ 𝒯⟦𝒢,?g,S⟧"
      and ch: "∀u ∈ callers callee. frag_ok Π ps u"
      and cof: "caller_of callee = Some caller" and "sink_node callee = FunctionResult p"
      and e: "(sink_node caller, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls ?g"
    have cin: "caller ∈ callers callee" using cof callers_caller_subset callers_refl by blast
    have cv: "caller ∈ 𝒯⟦𝒢,?g,S⟧" using valid_activation_trace_caller_valid[OF cvcallee cof] .
    have "frag_ok Π ps caller" using ch cin by blast
    then show "frag_ok Π ps (Resume caller callee (path_of caller
           @ [(cont, combine_collect 𝒢 dst (sink_store caller) (sink_store callee))]))"
      by (rule frag_ok_snoc[OF cv _ _ frag_edge_calls[OF wf _ _ _ _ e]])
         (use e in ‹auto dest: compile_prog_calls_source_stmt›)
  next
    show "t ∈ 𝒯⟦𝒢,?g,S⟧" by (rule t)
  qed
qed
```

An activation entered at `FunctionEntry p` reaches `FunctionResult q` only for `p = q`.

```
lemma valid_activation_trace_entry_result_eq:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and t: "t ∈ 𝒯⟦𝒢,compile_prog Π ps,S⟧"
    and en: "fst (hd (path_of t)) = FunctionEntry p"
    and sk: "sink_node t = FunctionResult q"
  shows "p = q"
proof -
  have "frag_ok Π ps t"
    using valid_activation_trace_frag_callers[OF wf t] callers_refl by blast
  then show ?thesis
  proof (cases rule: frag_okE)
    case (frag r d m m' Ep Kp)
    have "sink_node t ∈ pfn r m m'" using sink_in_path_nodes[OF t] frag(5) by blast
    with frag(4) en sk show ?thesis by (auto simp: pfn_def)
  next
    case (stub p' s)
    then have "sink_node t = FunctionEntry p'" by (simp add: sink_node_def)
    with sk show ?thesis by simp
  qed
qed

end
```

5.10 Live_Nodes

```
theory Live_Nodes
  imports Procedure_Ownership
begin
```

Every node of a procedure reaches its result

A procedure's nodes are the endpoints of its own intra edges and call continuations. Every edge out of one of them lands on another (`prog_node_intra`, `prog_node_calls`), and every one reaches `FunctionResult q` along intra edges and call-site-to-continuation steps (`prog_node_reaches`) — exactly the steps a routed equation reads backwards. Code after a `Return` is no exception: it still ends at the epilogue, whose return edge `compile_proc` always emits.

The whole program

Each declared procedure's fragment sits inside the compiled program, so its edges and their reachability carry over to the whole graph.

```
lemma compile_prog_proc_frag_sub:
  assumes wf: "wf_compile_input 𝒢 Π ps" and decl: "Π q = Some d"
  obtains m m' Ep Kp where
    "compile_proc Π q d m = (m', Ep, Kp)"
    "(FunctionEntry q, EA_Body q, Statement m) ∈ intra (compile_prog Π ps)"
    "Ep ⊆ intra (compile_prog Π ps)" "Kp ⊆ calls (compile_prog Π ps)"
proof -
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    and mainc: "compile_proc Π prog_main_name (⦇formals = [], body = (main_body Π)⦈) n1
                = (n2, Emain, Kmain)"
    and EI: "intra (compile_prog Π ps) = Eprocs ∪ Emain"
    and KC: "calls (compile_prog Π ps) = Kprocs ∪ Kmain"
    by (rule compile_prog_intra_split)
  show thesis
  proof (cases "q = prog_main_name")
    case True
    have dd: "d = (⦇formals = [], body = (main_body Π)⦈)"
      using decl True wf_compile_inputD(2)[OF wf] by simp
    have "(FunctionEntry q, EA_Body q, Statement n1) ∈ intra (compile_prog Π ps)"
      using EI compile_proc_entry_edge[OF mainc] True by auto
    then show ?thesis using that[of n1 n2 Emain Kmain] mainc True dd EI KC by simp
  next
    case False
    have rin: "q ∈ set ps" using decl wf_compile_inputD(10)[OF wf] False by auto
    from compile_procs_member_frag[OF procs wf_compile_inputD(9)[OF wf] rin decl]
    obtain m m' Ep Kp where cp0: "compile_proc Π q d m = (m', Ep, Kp)"
        and sub: "Ep ⊆ Eprocs" "Kp ⊆ Kprocs"
      by blast
    have "(FunctionEntry q, EA_Body q, Statement m) ∈ intra (compile_prog Π ps)"
      using EI sub compile_proc_entry_edge[OF cp0] by auto
    then show ?thesis using that cp0 sub EI KC by blast
  qed
qed
```

Call targets are declared

A call edge of a well-formed program enters only a declared procedure, so every callee has a fragment.

```
lemma compile_calls_target_declared:
  "compile Π p c k n = (n', en, E, K) ⟹ wf_source_com Π c
   ⟹ (u, ca, FunctionEntry q, k') ∈ K ⟹ ∃d. Π q = Some d"
proof (induction c arbitrary: k n n' en E K)
  case (Seq c1 c2)
  from Seq.prems(1) obtain n1 E1 K1 n2 E2 K2 where
      c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, Statement n, E1, K1)"
    and c2: "compile Π p c2 k (n + csize c1) = (n2, Statement (n + csize c1), E2, K2)"
    and K: "K = K1 ∪ K2"
    by (rule compile_SeqE)
  show ?case using Seq.IH(1)[OF c1] Seq.IH(2)[OF c2] Seq.prems(2,3) K by auto
next
  case (If i b c1 c2)
  from If.prems(1) obtain n1 E1 K1 n2 E2 K2 where
      c1: "compile Π p c1 k (Suc n) = (n1, Statement (Suc n), E1, K1)"
    and c2: "compile Π p c2 k (Suc n + csize c1) = (n2, Statement (Suc n + csize c1), E2, K2)"
```

```
    and K: "K = K1 ∪ K2"
    by (rule compile_IfE)
  show ?case using If.IH(1)[OF c1] If.IH(2)[OF c2] If.prems(2,3) K by auto
next
  case (While i b c)
  from While.prems(1) obtain n1 E1 K1 where
      c1: "compile Π p c (Statement n) (Suc n) = (n1, Statement (Suc n), E1, K1)"
    and K: "K = K1"
    by (rule compile_WhileE)
  show ?case using While.IH[OF c1] While.prems(2,3) K by auto
qed (auto split: option.splits)

lemma compile_prog_calls_target_declared:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and e: "(u, ca, FunctionEntry q, k) ∈ calls (compile_prog Π ps)"
  shows "∃d. Π q = Some d"
proof -
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    and mainc: "compile_proc Π prog_main_name (⦇formals = [], body = (main_body Π)⦈) n1
                = (n2, Emain, Kmain)"
    and KC: "calls (compile_prog Π ps) = Kprocs ∪ Kmain"
    by (rule compile_prog_intra_split)
  from e KC consider (P) "(u, ca, FunctionEntry q, k) ∈ Kprocs"
    | (M) "(u, ca, FunctionEntry q, k) ∈ Kmain"
    by auto
  then show ?thesis
  proof cases
    case P
    from compile_procs_calls_origin[OF procs P] obtain r decl m m' Ep Kp where
        decl: "Π r = Some decl" and cp: "compile_proc Π r decl m = (m', Ep, Kp)"
      and eK: "(u, ca, FunctionEntry q, k) ∈ Kp"
      by blast
    from cp obtain Eb where
        cb: "compile Π r (body decl) (Statement (m + csize (body decl))) m
               = (m + csize (body decl), Statement m, Eb, Kp)"
      by (rule compile_procE)
    have "wf_source_com Π (body decl)"
      using wf_compile_inputD(5)[OF wf decl] by (simp add: wf_proc_decl_def)
    from compile_calls_target_declared[OF cb this eK] show ?thesis .
  next
    case M
    from mainc obtain Eb where
        cb: "compile Π prog_main_name (main_body Π) (Statement (n1 + csize (main_body Π))) n1
               = (n1 + csize (main_body Π), Statement n1, Eb, Kmain)"
      by (rule compile_procE) simp
    from compile_calls_target_declared[OF cb wf_compile_inputD(3)[OF wf] M] show ?thesis .
  qed
qed
```

Every compiled node reaches the continuation or the result

A fragment's nodes all lead out of it: to its continuation or, along an explicit `Return`, to the procedure result. `compile_reaches` states this for the entry; the lemmas below extend it to every edge source and target.

```
lemma compile_src_reaches:
  assumes "compile Π p c k n = (n', en, E, K)" and "E ⊆ intra g" and "K ⊆ calls g"
    and "(u, a, v) ∈ E ∨ (u, ca, ce, v) ∈ K"
  shows "local_reaches g u k ∨ local_reaches g u (FunctionResult p)"
  using assms
proof (induction c arbitrary: k n n' en E K u a v ca ce)
  case (Seq c1 c2)
  from Seq.prems(1) obtain n1 E1 K1 n2 E2 K2 where
      c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, Statement n, E1, K1)"
    and c2: "compile Π p c2 k (n + csize c1) = (n2, Statement (n + csize c1), E2, K2)"
    and E: "E = E1 ∪ E2" and K: "K = K1 ∪ K2"
    by (rule compile_SeqE)
  have sub: "E1 ⊆ intra g" "K1 ⊆ calls g" "E2 ⊆ intra g" "K2 ⊆ calls g"
    using Seq.prems(2,3) E K by auto
  have mid: "local_reaches g (Statement (n + csize c1)) k
             ∨ local_reaches g (Statement (n + csize c1)) (FunctionResult p)"
    by (rule compile_reaches[OF c2 sub(3,4)])
  from Seq.prems(4) E K consider "(u, a, v) ∈ E1 ∨ (u, ca, ce, v) ∈ K1"
    | "(u, a, v) ∈ E2 ∨ (u, ca, ce, v) ∈ K2"
    by blast
  then show ?case
```

```
  proof cases
    case 1
    from Seq.IH(1)[OF c1 sub(1,2) 1] mid show ?thesis
      by (blast intro: local_reaches_trans)
  next
    case 2
    from Seq.IH(2)[OF c2 sub(3,4) 2] show ?thesis .
  qed
next
  case (If i b c1 c2)
  from If.prems(1) obtain n1 E1 K1 n2 E2 K2 where
      c1: "compile Π p c1 k (Suc n) = (n1, Statement (Suc n), E1, K1)"
    and c2: "compile Π p c2 k (Suc n + csize c1) = (n2, Statement (Suc n + csize c1), E2, K2)"
    and E: "E = {(Statement n, EA_Assume b, if is_SKIP c1 then k else Statement (Suc n)),
                (Statement n, EA_AssumeNot b, if is_SKIP c2 then k else Statement (Suc n + csize c1))} ∪ (if is_SKIP
c1 then {} else E1) ∪ (if is_SKIP c2 then {} else E2)"
    and K: "K = K1 ∪ K2"
    by (rule compile_IfE)
  have subK: "K1 ⊆ calls g" "K2 ⊆ calls g" using If.prems(3) K by auto
  have subE1: "¬ is_SKIP c1 ⟹ E1 ⊆ intra g" and subE2: "¬ is_SKIP c2 ⟹ E2 ⊆ intra g"
    using If.prems(2) E by auto
  have K1_skip: "is_SKIP c1 ⟹ K1 = {}" using c1 by auto
  have K2_skip: "is_SKIP c2 ⟹ K2 = {}" using c2 by auto
  have e1: "¬ is_SKIP c1 ⟹ local_reaches g (Statement (Suc n)) k
            ∨ local_reaches g (Statement (Suc n)) (FunctionResult p)"
    using compile_reaches[OF c1 subE1 subK(1)] by blast
  have head: "local_reaches g (Statement n) k ∨ local_reaches g (Statement n) (FunctionResult p)"
  proof (cases "is_SKIP c1")
    case True
    then show ?thesis using If.prems(2) E by (auto intro: local_reaches_intra_step)
  next
    case False
    have "(Statement n, EA_Assume b, Statement (Suc n)) ∈ intra g"
      using If.prems(2) E False by auto
    with e1[OF False] show ?thesis by (blast intro: local_reaches_intra_step)
  qed
  from If.prems(4) E K consider "u = Statement n"
    | "¬ is_SKIP c1" "(u, a, v) ∈ E1 ∨ (u, ca, ce, v) ∈ K1"
    | "¬ is_SKIP c2" "(u, a, v) ∈ E2 ∨ (u, ca, ce, v) ∈ K2"
    using K1_skip K2_skip by (auto split: if_splits)
  then show ?case
  proof cases
    case 1 then show ?thesis using head by simp
  next
    case 2 from If.IH(1)[OF c1 subE1[OF 2(1)] subK(1) 2(2)] show ?thesis .
  next
    case 3 from If.IH(2)[OF c2 subE2[OF 3(1)] subK(2) 3(2)] show ?thesis .
  qed
next
  case (While i b c)
  from While.prems(1) obtain n1 E1 K1 where
      c1: "compile Π p c (Statement n) (Suc n) = (n1, Statement (Suc n), E1, K1)"
    and E: "E = {(Statement n, EA_Assume b, Statement (Suc n)),
                (Statement n, EA_AssumeNot b, k)} ∪ E1"
    and K: "K = K1"
    by (rule compile_WhileE)
  have sub: "E1 ⊆ intra g" "K1 ⊆ calls g" using While.prems(2,3) E K by auto
  have exit: "local_reaches g (Statement n) k"
    using While.prems(2) E by (auto intro: local_reaches_intra_step)
  from While.prems(4) E K consider "u = Statement n" | "(u, a, v) ∈ E1 ∨ (u, ca, ce, v) ∈ K1"
    by blast
  then show ?case
  proof cases
    case 1 then show ?thesis using exit by simp
  next
    case 2
    from While.IH[OF c1 sub 2] exit show ?thesis by (blast intro: local_reaches_trans)
  qed
qed (auto split: option.splits intro: local_reaches_intra_step local_reaches_comb_step)

lemma compile_entry_src:
  assumes "compile Π p c k n = (n', en, E, K)"
  shows "(∃a w. (Statement n, a, w) ∈ E) ∨ (∃ca ce w. (Statement n, ca, ce, w) ∈ K)"
  using assms
proof (induction c arbitrary: k n n' en E K)
  case (Seq c1 c2)
```

```
  from Seq.prems obtain n1 E1 K1 n2 E2 K2 where
      c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, Statement n, E1, K1)"
    and "compile Π p c2 k (n + csize c1) = (n2, Statement (n + csize c1), E2, K2)"
    and E: "E = E1 ∪ E2" and K: "K = K1 ∪ K2"
    by (rule compile_SeqE)
  from Seq.IH(1)[OF c1] E K show ?case by blast
qed (auto simp: Let_def split: prod.splits option.splits)

lemma compile_tgt_src:
  assumes "compile Π p c k n = (n', en, E, K)"
    and "(u, a, v) ∈ E ∨ (u, ca, ce, v) ∈ K"
  shows "v = k ∨ v = FunctionResult p
        ∨ (∃a' w. (v, a', w) ∈ E) ∨ (∃ca' ce' w. (v, ca', ce', w) ∈ K)"
  using assms
proof (induction c arbitrary: k n n' en E K u a v ca ce)
  case (Seq c1 c2)
  from Seq.prems(1) obtain n1 E1 K1 n2 E2 K2 where
      c1: "compile Π p c1 (Statement (n + csize c1)) n = (n1, Statement n, E1, K1)"
    and c2: "compile Π p c2 k (n + csize c1) = (n2, Statement (n + csize c1), E2, K2)"
    and E: "E = E1 ∪ E2" and K: "K = K1 ∪ K2"
    by (rule compile_SeqE)
  from Seq.prems(2) E K consider "(u, a, v) ∈ E1 ∨ (u, ca, ce, v) ∈ K1"
    | "(u, a, v) ∈ E2 ∨ (u, ca, ce, v) ∈ K2"
    by blast
  then show ?case
  proof cases
    case 1
    from Seq.IH(1)[OF c1 1] compile_entry_src[OF c2] E K show ?thesis by blast
  next
    case 2
    from Seq.IH(2)[OF c2 2] E K show ?thesis by blast
  qed
next
  case (If i b c1 c2)
  from If.prems(1) obtain n1 E1 K1 n2 E2 K2 where
      c1: "compile Π p c1 k (Suc n) = (n1, Statement (Suc n), E1, K1)"
    and c2: "compile Π p c2 k (Suc n + csize c1) = (n2, Statement (Suc n + csize c1), E2, K2)"
    and E: "E = {(Statement n, EA_Assume b, if is_SKIP c1 then k else Statement (Suc n)),
                (Statement n, EA_AssumeNot b, if is_SKIP c2 then k else Statement (Suc n + csize c1))} ∪ (if is_SKIP
c1 then {} else E1) ∪ (if is_SKIP c2 then {} else E2)"
    and K: "K = K1 ∪ K2"
    by (rule compile_IfE)
  have K1_skip: "is_SKIP c1 ⟹ K1 = {}" using c1 by auto
  have K2_skip: "is_SKIP c2 ⟹ K2 = {}" using c2 by auto
  have s1:
    "¬ is_SKIP c1 ⟹ (∃a w. (Statement (Suc n), a, w) ∈ E) ∨ (∃ca ce w. (Statement (Suc n), ca, ce, w) ∈ K)"
    using compile_entry_src[OF c1] E K by auto
  have s2: "¬ is_SKIP c2 ⟹ (∃a w. (Statement (Suc n + csize c1), a, w) ∈ E)
             ∨ (∃ca ce w. (Statement (Suc n + csize c1), ca, ce, w) ∈ K)"
    using compile_entry_src[OF c2] E K by auto
  from If.prems(2) E K consider
      "u = Statement n" "v = (if is_SKIP c1 then k else Statement (Suc n))"
    | "u = Statement n" "v = (if is_SKIP c2 then k else Statement (Suc n + csize c1))"
    | "¬ is_SKIP c1" "(u, a, v) ∈ E1 ∨ (u, ca, ce, v) ∈ K1"
    | "¬ is_SKIP c2" "(u, a, v) ∈ E2 ∨ (u, ca, ce, v) ∈ K2"
    using K1_skip K2_skip by (auto split: if_splits)
  then show ?case
  proof cases
    case 1 then show ?thesis using s1 by (auto split: if_splits)
  next
    case 2 then show ?thesis using s2 by (auto split: if_splits)
  next
    case 3 from If.IH(1)[OF c1 3(2)] E K 3(1) show ?thesis by auto
  next
    case 4 from If.IH(2)[OF c2 4(2)] E K 4(1) show ?thesis by auto
  qed
next
  case (While i b c)
  from While.prems(1) obtain n1 E1 K1 where
      c1: "compile Π p c (Statement n) (Suc n) = (n1, Statement (Suc n), E1, K1)"
    and E: "E = {(Statement n, EA_Assume b, Statement (Suc n)),
                (Statement n, EA_AssumeNot b, k)} ∪ E1"
    and K: "K = K1"
    by (rule compile_WhileE)
  have s1: "(∃a w. (Statement (Suc n), a, w) ∈ E) ∨ (∃ca ce w. (Statement (Suc n), ca, ce, w) ∈ K)"
    using compile_entry_src[OF c1] E K by auto
```

```
  from While.prems(2) E K consider "u = Statement n" "v = Statement (Suc n)"
    | "u = Statement n" "v = k" | "(u, a, v) ∈ E1 ∨ (u, ca, ce, v) ∈ K1"
    by blast
  then show ?case
  proof cases
    case 1 then show ?thesis using s1 by simp
  next
    case 2 then show ?thesis by simp
  next
    case 3 from While.IH[OF c1 3] E K show ?thesis by blast
  qed
qed (auto split: option.splits)
```

A procedure's nodes

The endpoints of a procedure's own intra edges and call continuations. Sources reach the continuation or the result by `compile_src_reaches`; a target is the continuation, the result, or itself a source (`compile_tgt_src`); and the continuation of a procedure body is the epilogue, whose return edge is unconditional.

```
definition proc_nodes ::
  "(cfg_node × edge_action × cfg_node) set
   ⇒ (cfg_node × call_action × cfg_node × cfg_node) set ⇒ cfg_node set"
where
  "proc_nodes E K =
     {u. ∃a v. (u, a, v) ∈ E} ∪ {v. ∃u a. (u, a, v) ∈ E}
   ∪ {u. ∃ca ce k. (u, ca, ce, k) ∈ K} ∪ {k. ∃u ca ce. (u, ca, ce, k) ∈ K}"

lemma compile_proc_node_reaches:
  assumes cp: "compile_proc Π q d m = (m', Ep, Kp)"
    and Ep: "Ep ⊆ intra g" and Kp: "Kp ⊆ calls g"
    and x: "x ∈ proc_nodes Ep Kp"
  shows "local_reaches g x (FunctionResult q)"
proof -
  let ?r = "Statement (m + csize (body d))"
  from cp obtain Eb where
      cb: "compile Π q (body d) ?r m = (m + csize (body d), Statement m, Eb, Kp)"
    and E: "Ep = insert (FunctionEntry q, EA_Body q, Statement m)
                 (insert (?r, EA_Ret None q, FunctionResult q) Eb)"
    by (rule compile_procE)
  have Eb: "Eb ⊆ intra g" using E Ep by auto
  have epi: "local_reaches g ?r (FunctionResult q)"
    using E Ep by (auto intro: local_reaches_intra_step)
  have srcE: "local_reaches g u (FunctionResult q)" if "(u, a, v) ∈ Eb" for u a v
    using compile_src_reaches[OF cb Eb Kp, of u a v] that epi local_reaches_trans by blast
  have srcK: "local_reaches g u (FunctionResult q)" if "(u, ca, ce, v) ∈ Kp" for u ca ce v
    using compile_src_reaches[OF cb Eb Kp, of u _ v ca ce] that epi local_reaches_trans by blast
  have src_of: "local_reaches g v (FunctionResult q)"
    if
      "v = ?r ∨ v = FunctionResult q ∨ (∃a' w. (v, a', w) ∈ Eb) ∨ (∃ca' ce' w. (v, ca', ce', w) ∈ Kp)"
    for v
    using that epi srcE srcK by auto
  have tgtE: "local_reaches g v (FunctionResult q)" if "(u, a, v) ∈ Eb" for u a v
    using compile_tgt_src[OF cb, of u a v] that src_of by blast
  have tgtK: "local_reaches g v (FunctionResult q)" if "(u, ca, ce, v) ∈ Kp" for u ca ce v
    using compile_tgt_src[OF cb, of u _ v ca ce] that src_of by blast
  have body: "local_reaches g (Statement m) (FunctionResult q)"
    using compile_entry_src[OF cb] srcE srcK by blast
  have ent: "local_reaches g (FunctionEntry q) (FunctionResult q)"
    by (rule compile_proc_reaches_result[OF cp Ep Kp])
  from x show ?thesis
    unfolding proc_nodes_def E using srcE srcK tgtE tgtK epi body ent by auto
qed

lemma proc_nodes_pfn:
  assumes cp: "compile_proc Π q d m = (m', Ep, Kp)" and x: "x ∈ proc_nodes Ep Kp"
  shows "x ∈ pfn q m m'"
  using x compile_proc_intra_pfn[OF cp] compile_proc_calls_pfn[OF cp]
  unfolding proc_nodes_def by blast

definition prog_node :: "'a ptable ⇒ pname list ⇒ pname ⇒ cfg_node ⇒ bool" where
  "prog_node Π ps q x ⟷
     (∃d m m' Ep Kp. Π q = Some d ∧ compile_proc Π q d m = (m', Ep, Kp)
       ∧ (FunctionEntry q, EA_Body q, Statement m) ∈ intra (compile_prog Π ps)
       ∧ Ep ⊆ intra (compile_prog Π ps) ∧ Kp ⊆ calls (compile_prog Π ps)
       ∧ x ∈ proc_nodes Ep Kp)"
```

```
lemma prog_node_entry:
  assumes wf: "wf_compile_input 𝒢 Π ps" and decl: "Π q = Some d"
  shows "prog_node Π ps q (FunctionEntry q)"
proof -
  from compile_prog_proc_frag_sub[OF wf decl] obtain m m' Ep Kp where
      cp: "compile_proc Π q d m = (m', Ep, Kp)"
    and ent: "(FunctionEntry q, EA_Body q, Statement m) ∈ intra (compile_prog Π ps)"
    and sub: "Ep ⊆ intra (compile_prog Π ps)" "Kp ⊆ calls (compile_prog Π ps)"
    by blast
  have "FunctionEntry q ∈ proc_nodes Ep Kp"
    using compile_proc_entry_edge[OF cp] unfolding proc_nodes_def by blast
  then show ?thesis unfolding prog_node_def using decl cp ent sub by blast
qed

lemma prog_node_result:
  "prog_node Π ps q x ⟹ prog_node Π ps q (FunctionResult q)"
proof -
  assume "prog_node Π ps q x"
  then obtain d m m' Ep Kp where
      decl: "Π q = Some d" and cp: "compile_proc Π q d m = (m', Ep, Kp)"
    and ent: "(FunctionEntry q, EA_Body q, Statement m) ∈ intra (compile_prog Π ps)"
    and sub: "Ep ⊆ intra (compile_prog Π ps)" "Kp ⊆ calls (compile_prog Π ps)"
    unfolding prog_node_def by auto
  from cp have "(Statement (m + csize (body d)), EA_Ret None q, FunctionResult q) ∈ Ep"
    by (rule compile_procE) simp
  then have "FunctionResult q ∈ proc_nodes Ep Kp" unfolding proc_nodes_def by blast
  then show ?thesis unfolding prog_node_def using decl cp ent sub by blast
qed

lemma prog_node_reaches:
  "prog_node Π ps q x ⟹ local_reaches (compile_prog Π ps) x (FunctionResult q)"
  unfolding prog_node_def using compile_proc_node_reaches by auto

lemma prog_node_intra:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and u: "prog_node Π ps q u" and e: "(u, a, v) ∈ intra (compile_prog Π ps)"
  shows "prog_node Π ps q v"
proof -
  from u obtain d m m' Ep Kp where
      decl: "Π q = Some d" and cp: "compile_proc Π q d m = (m', Ep, Kp)"
    and ent: "(FunctionEntry q, EA_Body q, Statement m) ∈ intra (compile_prog Π ps)"
    and sub: "Ep ⊆ intra (compile_prog Π ps)" "Kp ⊆ calls (compile_prog Π ps)"
    and un: "u ∈ proc_nodes Ep Kp"
    unfolding prog_node_def by auto
  have "(u, a, v) ∈ Ep"
    using compile_prog_entry_frag[OF wf decl cp ent] proc_nodes_pfn[OF cp un] e by blast
  then have "v ∈ proc_nodes Ep Kp" unfolding proc_nodes_def by blast
  then show ?thesis unfolding prog_node_def using decl cp ent sub by blast
qed

lemma prog_node_calls:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and u: "prog_node Π ps q u" and e: "(u, ca, ce, k) ∈ calls (compile_prog Π ps)"
  shows "prog_node Π ps q k"
proof -
  from u obtain d m m' Ep Kp where
      decl: "Π q = Some d" and cp: "compile_proc Π q d m = (m', Ep, Kp)"
    and ent: "(FunctionEntry q, EA_Body q, Statement m) ∈ intra (compile_prog Π ps)"
    and sub: "Ep ⊆ intra (compile_prog Π ps)" "Kp ⊆ calls (compile_prog Π ps)"
    and un: "u ∈ proc_nodes Ep Kp"
    unfolding prog_node_def by auto
  have "(u, ca, ce, k) ∈ Kp"
    using compile_prog_entry_frag[OF wf decl cp ent] proc_nodes_pfn[OF cp un] e by blast
  then have "k ∈ proc_nodes Ep Kp" unfolding proc_nodes_def by blast
  then show ?thesis unfolding prog_node_def using decl cp ent sub by blast
qed

lemma prog_node_main_entry:
  assumes wf: "wf_compile_input 𝒢 Π ps"
  shows "prog_node Π ps prog_main_name (cfg_entry (compile_prog Π ps))"
  using prog_node_entry[OF wf wf_compile_inputD(2)[OF wf]] by simp

lemma prog_node_callee_entry:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and e: "(u, ca, FunctionEntry q, k) ∈ calls (compile_prog Π ps)"
  shows "prog_node Π ps q (FunctionEntry q)"
```

```
  using compile_prog_calls_target_declared[OF wf e] prog_node_entry[OF wf] by blast
end
```

5.11 Source_To_Trace

```
theory Source_To_Trace
  imports Procedure_Ownership Simulation_Preservation "Voblint_CFG.Activation_Trace_Collect"
begin
```

A source run is an activation trace

What the session is for. A graph configuration is tied to a valid activation trace: the activation currently running *is* the trace, and the runtime frame stack is exactly its chain of callers. Each `cstep` rule lines up with one `valid_activation_trace` constructor, so advancing the graph by an edge extends the trace by a step.

Composed with the simulation, this turns a run of the source program into a trace, and so turns any store the source can reach into a member of the collecting semantics at that node. Those are the statements the soundness session consumes; everything else in this session exists to make them true.

The representation invariant

`stack_repr` walks `caller_of` in lockstep with the runtime frame list: each `cframe (cont, dst, s)` pins the caller activation `caller` — its frozen store `sink_store caller = s` — and records the concrete `calls` edge that spawned the child, whose callee `FunctionEntry p` equals the child's (path-invariant) entry node.

```
inductive stack_repr :: "cfg ⇒ cframe list ⇒ activation_trace ⇒ bool" for g where
  empty [intro]: "caller_of t = None ⟹ stack_repr g [] t"
| frame [intro]: "caller_of t = Some caller ⟹ sink_store caller = s
         ⟹ fst (hd (path_of t)) = FunctionEntry p
         ⟹ (sink_node caller, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
         ⟹ stack_repr g stk caller
         ⟹ stack_repr g ((cont, dst, s) # stk) t"
```

Inversion by the shape of the runtime stack. A frame's clause recurses on the shorter stack, so it stays plain `[elim]`; an empty stack has only one clause and is `[elim!]`.

```
inductive_cases stack_repr_NilE [elim!]: "stack_repr g [] t"
inductive_cases stack_repr_ConsE [elim]: "stack_repr g (cf # stk) t"
```

`activation_trace_repr` pins a valid activation trace to a located configuration: the trace's sink is the current node/store, and its caller chain is the runtime stack.

```
definition activation_trace_repr ::
  "(vname ⇒ bool) ⇒ cfg ⇒ store set ⇒ cconf ⇒ activation_trace ⇒ bool" where
  "activation_trace_repr 𝒢 g S cf t = (case cf of (v, s, stk) ⇒
     t ∈ 𝒯⟨𝒢,g,S⟩ ∧ sink_node t = v ∧ sink_store t = s ∧ stack_repr g stk t)"

definition located_activation_trace :: "(vname ⇒ bool) ⇒ cfg ⇒ store set ⇒ cconf ⇒ bool" where
  "located_activation_trace 𝒢 g S cf = (∃t. activation_trace_repr 𝒢 g S cf t)"
```

`stack_repr` reads only the top `caller_of` and the path-invariant entry node, so it transfers across activations sharing both (an `Activation_Trace_Def.extend` appends — entry unchanged — and a `Resume` prepends the same activation's path — entry unchanged).

```
lemma stack_repr_cong:
  "stack_repr g stk t1 ⟹ caller_of t2 = caller_of t1
   ⟹ fst (hd (path_of t2)) = fst (hd (path_of t1)) ⟹ stack_repr g stk t2"
  by (cases rule: stack_repr.cases) auto
```

The return step

The load-bearing case. A `cstep` return pops the top frame; `stack_repr` identifies the caller `c` (`caller_of` recovers it) and its spawning `calls` edge, whose callee `FunctionEntry q` matches the returning `FunctionResult q` by `valid_activation_trace_entry_result_eq` — so the trace composes by `valid_activation_trace.ret`, and the combined store equals the `cstep` store.

```
lemma activation_trace_repr_Return:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and rep: "activation_trace_repr 𝒢 (compile_prog Π ps) S
                (FunctionResult q, tst, (cont, dst, caller) # stk) t0"
  shows "activation_trace_repr 𝒢 (compile_prog Π ps) S
          (cont, combine_collect 𝒢 dst caller tst, stk)
          (Resume (the (caller_of t0)) t0
             (path_of (the (caller_of t0)) @ [(cont, combine_collect 𝒢 dst caller tst)]))"
proof -
  let ?g = "compile_prog Π ps"
```

```
  from rep have t0v: "t0 ∈ T_{G,?g,S}" and sn0: "sink_node t0 = FunctionResult q"
    and ss0: "sink_store t0 = tst"
    and stk0: "stack_repr ?g ((cont, dst, caller) # stk) t0"
    by (auto simp: activation_trace_repr_def)
  from stk0 obtain c p pars args where cof: "caller_of t0 = Some c"
    and ssc: "sink_store c = caller" and entryp: "fst (hd (path_of t0)) = FunctionEntry p"
    and edge: "(sink_node c, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls ?g"
    and stkc: "stack_repr ?g stk c"
    by (cases rule: stack_repr.cases) auto
  have cvalid: "c ∈ T_{G,?g,S}" using valid_activation_trace_caller_valid[OF t0v cof] .
  have pq: "p = q" using valid_activation_trace_entry_result_eq[OF wf t0v entryp sn0] .
  have cthe: "the (caller_of t0) = c" using cof by simp
  let ?r = "combine_collect G dst caller tst"
  let ?t' = "Resume c t0 (path_of c @ [(cont, ?r)])"
  have edge_q: "(sink_node c, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls ?g"
    using edge pq by simp
  have valid': "?t' ∈ T_{G,?g,S}"
    using valid_activation_trace.ret[OF t0v cof sn0 edge_q] ssc ss0 by simp
  have sn': "sink_node ?t' = cont" by (simp add: sink_node_def)
  have ss': "sink_store ?t' = ?r" by (simp add: sink_store_def)
  have entry': "fst (hd (path_of ?t')) = fst (hd (path_of c))"
    using valid_activation_trace_path_nonempty[OF cvalid] by simp
  have stk': "stack_repr ?g stk ?t'"
    using stack_repr_cong[OF stkc, of ?t'] entry' by simp
  show ?thesis
    unfolding cthe using valid' sn' ss' stk' by (simp add: activation_trace_repr_def)
qed
```

The invariant is preserved by located CFG steps

Each cstep rule maps to one valid_activation_trace constructor: intra ↦ Activation_Trace_Def.extend, call ↦ activation_trace.Call, return ↦ Resume (activation_trace_repr_Return).

```
lemma cstep_preserves_activation_trace_repr:
  assumes wf: "wf_compile_input G Π ps"
    and step: "G, compile_prog Π ps ⊢ cf →_c cf'"
    and rep: "activation_trace_repr G (compile_prog Π ps) S cf t"
  shows "∃t'. activation_trace_repr G (compile_prog Π ps) S cf' t'"
  using step rep
proof (cases rule: cstep.cases)
  case (Intra u a v s' s stk')
  let ?g = "compile_prog Π ps"
  from rep have tv: "t ∈ T_{G,?g,S}" and sn: "sink_node t = u"
    and ss: "sink_store t = s"
    and str: "stack_repr ?g stk' t"
    using Intra by (auto simp: activation_trace_repr_def)
  have e1: "(sink_node t, a, v) ∈ intra ?g" using Intra sn by simp
  have e2: "s' ∈ edge_step a (sink_store t)" using Intra ss by simp
  have "extend t (v, s') ∈ T_{G,?g,S}" using valid_activation_trace.intra[OF tv e1 e2] .
  moreover have "stack_repr ?g stk' (extend t (v, s'))"
    using stack_repr_cong[OF str] valid_activation_trace_path_nonempty[OF tv] by simp
  ultimately have "activation_trace_repr G ?g S (v, s', stk') (extend t (v, s'))"
    by (simp add: activation_trace_repr_def)
  then show ?thesis using Intra by auto
next
  case (Call u dst pars actuals q cont s stk')
  let ?g = "compile_prog Π ps"
  from rep have tv: "t ∈ T_{G,?g,S}" and sn: "sink_node t = u"
    and ss: "sink_store t = s"
    and str: "stack_repr ?g stk' t"
    using Call by (auto simp: activation_trace_repr_def)
  have edge: "(sink_node t, CallEdge dst pars actuals, FunctionEntry q, cont) ∈ calls ?g"
    using Call sn by simp
  let ?child = "Call t [(FunctionEntry q, call_enter G (CallEdge dst pars actuals) s)]"
  have child_valid: "?child ∈ T_{G,?g,S}"
    using valid_activation_trace.call[OF tv edge] ss by simp
  have "stack_repr ?g ((cont, dst, s) # stk') ?child"
  proof (rule stack_repr.frame)
    show "caller_of ?child = Some t" by simp
    show "sink_store t = s" using ss .
    show "fst (hd (path_of ?child)) = FunctionEntry q" by simp
    show "(sink_node t, CallEdge dst pars actuals, FunctionEntry q, cont) ∈ calls ?g"
      using edge .
    show "stack_repr ?g stk' t" using str .
  qed
```

```
  with child_valid have "activation_trace_repr 𝒢 ?g S
      (FunctionEntry q, call_enter 𝒢 (CallEdge dst pars actuals) s,
       (cont, dst, s) # stk') ?child"
    by (simp add: activation_trace_repr_def sink_node_def sink_store_def)
  then show ?thesis using Call by auto
next
  case (Return q tst cont dst caller stk')
  from rep Return(1)
  have rep': "activation_trace_repr 𝒢 (compile_prog Π ps) S
                (FunctionResult q, tst, (cont, dst, caller) # stk') t" by simp
  from activation_trace_repr_Return[OF wf rep'] show ?thesis using Return(2) by auto
qed

lemma located_activation_trace_entry:
  assumes "s ∈ S"
  shows "located_activation_trace 𝒢 g S (cfg_entry g, s, [])"
proof -
  have "activation_trace_repr 𝒢 g S (cfg_entry g, s, []) (Root [(cfg_entry g, s)])"
    using assms
    by (auto simp: activation_trace_repr_def sink_node_def sink_store_def
      valid_activation_trace.root)
  then show ?thesis by (auto simp: located_activation_trace_def)
qed

lemma cstep_preserves_located_activation_trace:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and "located_activation_trace 𝒢 (compile_prog Π ps) S cf"
    and "𝒢, compile_prog Π ps ⊢ cf →c cf'"
  shows "located_activation_trace 𝒢 (compile_prog Π ps) S cf'"
proof -
  from assms(2) obtain t where "activation_trace_repr 𝒢 (compile_prog Π ps) S cf t"
    by (auto simp: located_activation_trace_def)
  then obtain t' where "activation_trace_repr 𝒢 (compile_prog Π ps) S cf' t'"
    using cstep_preserves_activation_trace_repr[OF wf assms(3)] by blast
  then show ?thesis by (auto simp: located_activation_trace_def)
qed

lemma csteps_preserve_located_activation_trace:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and "located_activation_trace 𝒢 (compile_prog Π ps) S cf"
    and "𝒢, compile_prog Π ps ⊢ cf →c* cf'"
  shows "located_activation_trace 𝒢 (compile_prog Π ps) S cf'"
  using assms(3) assms(2)
proof (induction rule: star.induct)
  case (refl a)
  then show ?case .
next
  case (step a b c)
  have "located_activation_trace 𝒢 (compile_prog Π ps) S b"
    by (rule cstep_preserves_located_activation_trace[OF wf step.prems step.hyps(1)])
  then show ?case by (rule step.IH)
qed
```

The initial main activation

The program entry `FunctionEntry prog_main_name` is an ordinary `csim.Base` activation: the distinguished main procedure is declared in Π (`wf_compile_input`), so its body fragment is certified by `procs_embedded_compile_prog`. One `EA_Body` edge crosses from `FunctionEntry prog_main_name` to the body entry `en`, where the `Base` activation simulates the source `main_body Π`.

```
lemma compile_prog_main_base:
  assumes wf: "wf_compile_input 𝒢 Π ps"
  obtains en where
    "(FunctionEntry prog_main_name, EA_Body prog_main_name, en) ∈ intra (compile_prog Π ps)"
    "Π, compile_prog Π ps, prog_main_name ⊢ (main_body Π, s, []) ≈ (en, s, [])"
proof -
  let ?g = "compile_prog Π ps"
  have pc: "procs_embedded Π ?g" by (rule procs_embedded_compile_prog[OF wf])
  have main_decl: "Π prog_main_name = Some (⦇formals = [], body = (main_body Π)⦈)"
    by (rule wf_compile_inputD(2)[OF wf])
  obtain k m en where
    cacc: "compiled_at Π ?g prog_main_name (body (⦇formals = [], body = (main_body Π)⦈)) k m"
      and ctrl: "control_at Π prog_main_name (body (⦇formals = [], body = (main_body Π)⦈)) k m
                   (body (⦇formals = [], body = (main_body Π)⦈)) en"
      and entry: "(FunctionEntry prog_main_name, EA_Body prog_main_name, en) ∈ intra ?g"
    by (rule procs_embedded_activation[OF pc main_decl])
```

```
  have bodyeq: "body (⦇formals = [], body = (main_body Π)⦈) = (main_body Π)" by simp
  have base: "Π, ?g, prog_main_name ⊢ (main_body Π, s, []) ≈ (en, s, [])"
    by (rule csim.Base[OF ctrl[unfolded bodyeq] cacc[unfolded bodyeq]])
  from entry base show ?thesis ..
qed
```

Top-level structural facts of the invariant

A trace has no caller exactly when its activation is at the top level (empty runtime stack).

```
lemma stack_repr_Nil_iff:
  "stack_repr g stk t ⟹ (stk = []) = (caller_of t = None)"
  by (cases rule: stack_repr.cases) auto
```

A callerless activation carries exactly the initial context, whatever the policy — the `Root` case never consults `adm`, and the `Resume` case only forwards to the same callerless ancestor. No validity is needed: the two clauses involved name no edge.

```
lemma activation_context_rel_caller_of_None:
  "caller_of t = None
   ⟹ activation_context_rel 𝒢 adm c₀ g t c ⟷ c = c₀"
  by (induction t) auto
```

The source bridge

Composing the initial `csim.Base` with `csim_star` (the source-to-`cstep` simulation) and the located invariant: every source run produces a matching valid activation trace at the simulated node.

```
theorem source_run_has_activation_trace:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and s0: "s0 ∈ S"
    and run: "𝒢, Π ⊢ (main_body Π, s0, []) →ₚ* (residual, s, frs)"
  shows "∃v stk t. Π, compile_prog Π ps, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
                 ∧ activation_trace_repr 𝒢 (compile_prog Π ps) S (v, s, stk) t"
proof -
  let ?g = "compile_prog Π ps"
  have pc: "procs_embedded Π ?g" by (rule procs_embedded_compile_prog[OF wf])
  have swf: "return_safe (main_body Π)" by (rule wf_compile_input_return_safe[OF wf])
  obtain en where entry: "(FunctionEntry prog_main_name, EA_Body prog_main_name, en) ∈ intra ?g"
    and base: "Π, ?g, prog_main_name ⊢ (main_body Π, s0, []) ≈ (en, s0, [])"
    by (rule compile_prog_main_base[OF wf])
  have loc0:
    "located_activation_trace 𝒢 (compile_prog Π ps) S (FunctionEntry prog_main_name, s0, [])"
    using located_activation_trace_entry[where g = "compile_prog Π ps", OF s0] by simp
  have step0: "𝒢, ?g ⊢ (FunctionEntry prog_main_name, s0, []) →c (en, s0, [])"
    by (rule cstep_body[OF entry])
  have loc_en: "located_activation_trace 𝒢 ?g S (en, s0, [])"
    by (rule cstep_preserves_located_activation_trace[OF wf loc0 step0])
  from csim_star[OF base pc swf run] obtain cf'
    where run_c: "𝒢, ?g ⊢ (en, s0, []) →c* cf'"
      and sim': "Π, ?g, prog_main_name ⊢ (residual, s, frs) ≈ cf'"
    by blast
  obtain v s' stk where cf': "cf' = (v, s', stk)" by (cases cf')
  have store: "s' = s" using csim_store_eq[OF sim'[unfolded cf']] by simp
  have loc_v: "located_activation_trace 𝒢 ?g S (v, s, stk)"
    using csteps_preserve_located_activation_trace[OF wf loc_en run_c] cf' store by simp
  from loc_v obtain t where "activation_trace_repr 𝒢 ?g S (v, s, stk) t"
    by (auto simp: located_activation_trace_def)
  then show ?thesis using sim' cf' store by blast
qed
```

The plain projected source bridge: a reachable source store lies in the activation-trace collecting `activation_collect` at the simulated node, in a context the witness trace carries. A relational policy may leave a trace without any context, so the bridge asks for one: `has_ctx` is what `activation_coverage`'s `TOTAL` buys downstream, and what a functional policy has outright (`source_store_in_activation_collect_of_fun`).

```
theorem source_store_in_activation_collect:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and s0: "s0 ∈ S"
    and run: "𝒢, Π ⊢ (main_body Π, s0, []) →ₚ* (residual, s, frs)"
    and has_ctx: "⋀t. t ∈ 𝒯𝒢,compile_prog Π ps,S
                  ⟹ ∃c. activation_context_rel 𝒢 adm c₀ (compile_prog Π ps) t c"
  shows "∃v stk t c. Π, compile_prog Π ps, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
                  ∧ activation_trace_repr 𝒢 (compile_prog Π ps) S (v, s, stk) t
                  ∧ activation_context_rel 𝒢 adm c₀ (compile_prog Π ps) t c
```

```
                ∧ s ∈ 𝒜𝒢,adm,c₀,compile_prog Π ps,S ᵛ c"
proof -
  let ?g = "compile_prog Π ps"
  from source_run_has_activation_trace[OF wf s0 run] obtain v stk t
    where sim: "Π, ?g, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)"
      and rep: "activation_trace_repr 𝒢 ?g S (v, s, stk) t" by blast
  from rep have tv: "t ∈ 𝒯𝒢,?g,S" and sn: "sink_node t = v"
    and ss: "sink_store t = s"
    by (auto simp: activation_trace_repr_def)
  from has_ctx[OF tv] obtain c where tc: "activation_context_rel 𝒢 adm c₀ ?g t c" by blast
  have "s ∈ 𝒜𝒢,adm,c₀,?g,S ᵛ c"
    using activation_collect_I[OF tv sn tc] ss by simp
  then show ?thesis using sim rep tc by blast
qed
```

A functional policy carries `activation_context_of`'s context on every valid activation trace, so it needs no context witness.

```
corollary source_store_in_activation_collect_of_fun:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and s0: "s0 ∈ S"
    and run: "𝒢, Π ⊢ (main_body Π, s0, []) →ₚ* (residual, s, frs)"
  shows "∃v stk t c. Π, compile_prog Π ps, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
                ∧ activation_trace_repr 𝒢 (compile_prog Π ps) S (v, s, stk) t
                ∧ activation_context_of f c₀ t = c
                ∧ s ∈ 𝒜𝒢,context_policy_of_fun f,c₀,compile_prog Π ps,S ᵛ c"
proof -
  let ?g = "compile_prog Π ps"
  from source_run_has_activation_trace[OF wf s0 run] obtain v stk t
    where sim: "Π, ?g, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)"
      and rep: "activation_trace_repr 𝒢 ?g S (v, s, stk) t" by blast
  from rep have tv: "t ∈ 𝒯𝒢,?g,S" and sn: "sink_node t = v"
    and ss: "sink_store t = s"
    by (auto simp: activation_trace_repr_def)
  have tc:
    "activation_context_rel 𝒢 (context_policy_of_fun f) c₀ ?g t (activation_context_of f c₀ t)"
    by (simp add: activation_context_rel_of_fun_iff[OF tv])
  have "s ∈ 𝒜𝒢,context_policy_of_fun f,c₀,?g,S ᵛ
             (activation_context_of f c₀ t)"
    using activation_collect_I[OF tv sn tc] ss by simp
  then show ?thesis using sim rep by blast
qed
```

The witness-free top-level result: a store reached with an empty source frame stack lies in the activation collecting at the fixed seed context — no `activation_trace` witness and no context existential. This is the shape a user reads for main-level program points.

```
theorem source_toplevel_in_activation_collect:
  assumes wf: "wf_compile_input 𝒢 Π ps"
    and s0: "s0 ∈ S"
    and run: "𝒢, Π ⊢ (main_body Π, s0, []) →ₚ* (residual, s, [])"
  shows "∃v. Π, compile_prog Π ps, prog_main_name ⊢ (residual, s, []) ≈ (v, s, [])
           ∧ s ∈ 𝒜𝒢,adm,c₀,compile_prog Π ps,S ᵛ c₀"
proof -
  let ?g = "compile_prog Π ps"
  from source_run_has_activation_trace[OF wf s0 run] obtain v stk t
    where sim: "Π, ?g, prog_main_name ⊢ (residual, s, []) ≈ (v, s, stk)"
      and rep: "activation_trace_repr 𝒢 ?g S (v, s, stk) t" by blast
  have stk0: "stk = []" using sim by blast
  from rep stk0 have tv: "t ∈ 𝒯𝒢,?g,S" and sn: "sink_node t = v"
    and ss: "sink_store t = s" and sr: "stack_repr ?g [] t"
    by (auto simp: activation_trace_repr_def)
  have cof: "caller_of t = None" using stack_repr_Nil_iff[OF sr] by simp
  have covered: "activation_context_rel 𝒢 adm c₀ ?g t c₀"
    by (simp add: activation_context_rel_caller_of_None[OF cof])
  have "s ∈ 𝒜𝒢,adm,c₀,?g,S ᵛ c₀"
    using activation_collect_I[OF tv sn covered] ss by simp
  then show ?thesis using sim stk0 by blast
qed
```

The bridge is inhabited

`source_run_has_activation_trace` and the collecting corollaries all assume a well-formed compiler input, so the session ends by exhibiting one and running the bridge on it. The zero-step run of the empty `main` is enough:

it yields an actual node of an actual compiled graph and an actual `valid_activation_trace` witness, so none of the above is vacuously true.

```
theorem source_run_has_activation_trace_witness:
  "∃v stk t. witness_pi, witness_cfg, prog_main_name ⊢ (SKIP (), s0, []) ≈ (v, s0, stk)
          ∧ activation_trace_repr (λ_. False) witness_cfg {s0} (v, s0, stk) t"
  using source_run_has_activation_trace[OF witness_wf singletonI star.refl] by simp

end
```

5.12 Elaborated_Compilation

```
theory Elaborated_Compilation
  imports Source_To_Trace
begin
```

Elaborated compilation: every residual names its CFG node

Elaboration annotates each statement with the nodes the compiler emits for it. On an elaborated program the forward simulation is functional: the node a source configuration is related to is read off its residual (`csim_point_compile_prog`).

Erasure

```
definition erase :: "'a acom ⇒ com" where
  "erase = map_gcom (λ_. ()) id id"

definition erase_decl :: "'a gproc_decl ⇒ proc_decl" where
  "erase_decl d = (|formals = formals d, body = erase (body d)|)"

definition erase_pt :: "'a ptable ⇒ proc_table" where
  "erase_pt Π = map_option erase_decl ∘ Π"

lemma erase_simps [simp]:
  "erase (SKIP i) = SKIP ()"
  "erase (Assign i x a) = Assign () x a"
  "erase (Check i l b) = Check () l b"
  "erase (Seq c1 c2) = Seq (erase c1) (erase c2)"
  "erase (If i b c1 c2) = If () b (erase c1) (erase c2)"
  "erase (While i b c) = While () b (erase c)"
  "erase (gcom.Call i dst q args) = gcom.Call () dst q args"
  "erase (gcom.Return i e) = gcom.Return () e"
  "erase (Restore i) = Restore ()"
  "erase (Unwind i) = Unwind ()"
  by (simp_all add: erase_def option.map_id)

lemma erase_decl_simps [simp]:
  "formals (erase_decl d) = formals d" "body (erase_decl d) = erase (body d)"
  by (simp_all add: erase_decl_def)

lemma is_Restore_erase [simp]: "is_Restore (erase c) = is_Restore c"
  by (cases c) simp_all
```

Locations and elaboration

Compiler metadata for one statement occurrence: the node the statement starts at, the node control reaches once it completes, and the result node a return through it reaches.

```
record loc =
  loc_here :: cfg_node
  loc_done :: cfg_node
  loc_exit :: cfg_node

definition stmt_loc :: "pname ⇒ cfg_node ⇒ nat ⇒ loc" where
  "stmt_loc p k n = (|loc_here = Statement n, loc_done = k, loc_exit = FunctionResult p|)"
```

Offsets are threaded exactly as `compile` threads them. A source `SKIP` is its own node; a `SKIP` branch of an `gcom.If` emits no edge, so it is done at the continuation. A call's exit is the callee's result node.

```
fun elab :: "pname ⇒ cfg_node ⇒ nat ⇒ 'a acom ⇒ loc acom" where
  "elab p k n (SKIP _) = SKIP (stmt_loc p (Statement n) n)"
| "elab p k n (Assign _ x a) = Assign (stmt_loc p k n) x a"
| "elab p k n (VIMP_Proc.gcom.Check _ l b) = VIMP_Proc.gcom.Check (stmt_loc p k n) l b"
| "elab p k n (Seq c1 c2) =
     Seq (elab p (Statement (n + csize c1)) n c1) (elab p k (n + csize c1) c2)"
| "elab p k n (If _ b c1 c2) =
     If (stmt_loc p k n) b
       (if is_SKIP c1 then SKIP (stmt_loc p k (Suc n)) else elab p k (Suc n) c1)
       (if is_SKIP c2 then SKIP (stmt_loc p k (Suc n + csize c1))
        else elab p k (Suc n + csize c1) c2)"
| "elab p k n (While _ b c) = While (stmt_loc p k n) b (elab p (Statement n) (Suc n) c)"
| "elab p k n (gcom.Call _ dst q args) =
     gcom.Call ((stmt_loc p k n)(|loc_exit := FunctionResult q|)) dst q args"
| "elab p k n (gcom.Return _ e) = gcom.Return (stmt_loc p k n) e"
| "elab p k n (Restore _) = Restore (stmt_loc p k n)"
```

```
| "elab p k n (Unwind _) = Unwind (stmt_loc p k n)"

lemma csize_elab [simp]: "csize (elab p k n c) = csize c"
  by (induction p k n c rule: elab.induct) (simp_all, safe, simp_all)

lemma elab_eq_SKIP: "elab p k n c = SKIP a ⟷ (∃i. c = SKIP i ∧ stmt_loc p (Statement n) n = a)"
  by (cases c) simp_all


lemma elab_source_not_Restore [simp]: "source_com c ⟹ elab p k n c ≠ Restore a"
  by (cases c) simp_all

lemma csize_elab_branch [simp]:
  "csize (if is_SKIP c then SKIP x else elab p k n c) = csize c"
  by (cases "is_SKIP c") (auto simp del: csize.simps(5))
```

The node a residual occupies. A `Seq` whose head is unwinding into a `Restore` sits at the call's exit, the callee's result node.

```
fun point :: "loc acom ⇒ cfg_node" where
  "point (SKIP a) = loc_done a"
| "point (Seq c1 c2) = (if is_Restore c2 ∧ unwinding c1 then loc_exit (stmt_ann c2) else point c1)"
| "point (Restore a) = loc_exit a"
| "point (Unwind a) = loc_exit a"
| "point c = loc_here (stmt_ann c)"
```

On an elaborated fragment the residual alone fixes the node: `point` reads it off the residual's annotation.

```
lemma control_at_point:
  assumes "control_at Π p c0 k n r v" "c0 = elab p k n c" "source_com c"
  shows "v = point r"
  using assms
proof (induction arbitrary: c rule: control_at.induct)
  case (SeqLeft c1 n r v c2 k)
  from SeqLeft.prems(1) obtain d1 d2 where c: "c = Seq d1 d2"
    by (cases c) simp_all
  with SeqLeft.prems have "v = point r"
    by (intro SeqLeft.IH[of d1]) simp_all
  with SeqLeft.prems c show ?case by simp
next
  case (SeqRight c2 k n c1 r v)
  from SeqRight.prems(1) obtain d1 d2 where c: "c = Seq d1 d2"
    by (cases c) simp_all
  with SeqRight.prems show ?case
    by (intro SeqRight.IH[of d2]) simp_all
next
  case (IfLeft c1 k n r v i b c2)
  from IfLeft.prems(1) obtain j d1 d2 where c: "c = If j b d1 d2"
    by (cases c) simp_all
  with IfLeft.prems IfLeft.hyps(2) have "¬ is_SKIP d1" by auto
  with IfLeft.prems c show ?case
    by (intro IfLeft.IH[of d1]) simp_all
next
  case (IfRight c2 k n c1 r v i b)
  from IfRight.prems(1) obtain j d1 d2 where c: "c = If j b d1 d2"
    by (cases c) simp_all
  with IfRight.prems IfRight.hyps(2) have "¬ is_SKIP d2" by auto
  moreover from IfRight.prems c have "csize c1 = csize d1"
    using csize_elab_branch[of d1 "stmt_loc p k (Suc n)" p k "Suc n"] by simp
  ultimately show ?case using IfRight.prems c
    by (intro IfRight.IH[of d2]) simp_all
next
  case (IfDone c1 j c2 i b k n)
  from IfDone.prems(1) obtain i' d1 d2 where c: "c = If i' b d1 d2"
    by (cases c) simp_all
  with IfDone.hyps IfDone.prems show ?case
    by (cases "is_SKIP d1"; cases "is_SKIP d2") (auto simp: elab_eq_SKIP stmt_loc_def)
next
  case (WhileBody c n r v i b k)
  from WhileBody.prems(1) obtain j d where c: "c = While j b d"
    by (cases c) simp_all
  with WhileBody.prems have "v = point r"
    by (intro WhileBody.IH[of d]) simp_all
  then show ?case by simp
next
  case (Skip i n k) then show ?case by (cases c) (simp_all add: stmt_loc_def)
next
```

```
  case (Assign i x a k n) then show ?case by (cases c) (simp_all add: stmt_loc_def)
next
  case (AssignDone i x a k n) then show ?case by (cases c) (simp_all add: stmt_loc_def)
next
  case (Check i l b k n) then show ?case by (cases c) (simp_all add: stmt_loc_def)
next
  case (CheckDone i l b k n) then show ?case by (cases c) (simp_all add: stmt_loc_def)
next
  case (IfHead i b c1 c2 k n) then show ?case by (cases c) (simp_all add: stmt_loc_def)
next
  case (WhileHead i b c' k n) then show ?case by (cases c) (simp_all add: stmt_loc_def)
next
  case (WhileUnfolded i b c' k n) then show ?case by (cases c) (simp_all add: stmt_loc_def)
next
  case (WhileDone i b c' k n) then show ?case by (cases c) (simp_all add: stmt_loc_def)
next
  case (CallHead i dst q actuals k n) then show ?case by (cases c) (simp_all add: stmt_loc_def)
next
  case (CallDone i dst q actuals k n) then show ?case by (cases c) (simp_all add: stmt_loc_def)
next
  case (ReturnHead i e k n) then show ?case by (cases c) (simp_all add: stmt_loc_def)
qed
```

The simulation determines the node

Every call annotation exits at its callee's result node. Elaboration establishes it and every residual `control_at` reaches inherits it.

```
fun calls_ok :: "loc acom ⇒ bool" where
  "calls_ok (Seq c1 c2) = (calls_ok c1 ∧ calls_ok c2)"
| "calls_ok (If _ b c1 c2) = (calls_ok c1 ∧ calls_ok c2)"
| "calls_ok (While _ b c) = calls_ok c"
| "calls_ok (gcom.Call a dst q args) = (loc_exit a = FunctionResult q)"
| "calls_ok _ = True"

lemma calls_ok_elab: "calls_ok (elab p k n c)"
  by (induction p k n c rule: elab.induct) simp_all

lemma control_at_calls_ok: "control_at Π p c0 k n r v ⟹ calls_ok c0 ⟹ calls_ok r"
  by (induction rule: control_at.induct) simp_all

lemma calls_ok_seq_after: "calls_ok (seq_after c afters) ⟹ calls_ok c"
  by (induction afters arbitrary: c) fastforce+

lemma point_seq_after [simp]:
  "¬ unwinding c ⟹ point (seq_after c afters) = point c ∧ ¬ unwinding (seq_after c afters)"
  by (induction afters arbitrary: c) simp_all

lemma pop_ready_point:
  "pop_ready w ⟹ pop_ann w = Some i ⟹ point w = loc_exit i ∧ ¬ unwinding w"
  by (cases w rule: pop_ready.cases) simp_all
```

Every compiled body is the elaboration of a source body at its own offsets.

```
definition located :: "loc ptable ⇒ cfg ⇒ bool" where
  "located Π g ⟷
     (∀p body k n. compiled_at Π g p body k n ⟶
        (∃c :: com. source_com c ∧ body = elab p k n c))"

lemma locatedE:
  assumes "located Π g" "compiled_at Π g p b k n"
  obtains c :: com where "source_com c" "b = elab p k n c"
  using assms unfolding located_def by blast
```

On a located program the simulation names its node, `point c`; that the residual is not unwinding is carried along for the `Restore` case.

```
lemma csim_point:
  "Π, g, p ⊢ x ≈ y ⟹ located Π g ⟹ fst y = point (fst x) ∧ ¬ unwinding (fst x)"
proof (induction rule: csim.induct)
  case (Base p body_p k n c v s)
  from Base.prems Base.hyps(2) obtain c0 :: com where c0: "source_com c0" "body_p = elab p k n c0"
    by (rule locatedE)
  have "v = point c"
    by (rule control_at_point[OF Base.hyps(1) c0(2) c0(1)])
  moreover have "¬ unwinding c"
    using control_at_not_returning[OF Base.hyps(1)] unwinding_is_returning by blast
```

```
  ultimately show ?case by simp
next
  case (Nested q inner s frs v stk pc body_pc kc nc i afters cont dst args site caller)
  from Nested.IH[OF Nested.prems] show ?case by simp
next
  case (Returning w i pc body_pc kc nc afters cont dst q args site callee caller)
  from Returning.prems Returning.hyps(5) obtain c0 :: com where c0: "body_pc = elab pc kc nc c0"
    by (rule locatedE)
  have "calls_ok (seq_after (gcom.Call i dst q args) afters)"
    by (rule control_at_calls_ok[OF Returning.hyps(4)]) (simp add: c0 calls_ok_elab)
  then have "loc_exit i = FunctionResult q"
    using calls_ok_seq_after by fastforce
  with pop_ready_point[OF Returning.hyps(1,2)] show ?case by simp
qed
```

Compilation ignores annotations

The compiler reads neither the annotations nor anything of the table but the formals, so two programs with the same erasure compile to the same graph.

```
lemma csize_erase [simp]: "csize (erase c) = csize c"
  by (induction c) simp_all

lemma erase_eq_SKIP [simp]: "erase c = SKIP i ⟷ is_SKIP c"
  by (cases c) simp_all

lemma compile_erase:
  assumes "call_formals Π = call_formals Π'"
  shows "compile Π p c k n = compile Π' p (erase c) k n"
proof (induction c arbitrary: k n)
  case (Call i dst q args)
  have "call_formals Π q = call_formals Π' q" using assms by (rule fun_cong)
  then show ?case by (simp only: erase_simps compile.simps)
qed (simp_all del: call_formals_def)

lemma call_formals_erase_pt: "call_formals (erase_pt Π) = call_formals Π"
  by (rule ext) (simp add: erase_pt_def erase_decl_def split: option.split)
```

Two programs with the same erasure compile to the same graph.

```
lemma compile_erase_cong:
  assumes "erase_pt Π = erase_pt Π'" "erase c = erase c'"
  shows "compile Π p c k n = compile Π' p c' k n"
proof -
  have "compile Π p c k n = compile (erase_pt Π) p (erase c) k n"
    by (rule compile_erase) (simp add: call_formals_erase_pt)
  also have "... = compile (erase_pt Π') p (erase c') k n"
    using assms by simp
  also have "... = compile Π' p c' k n"
    by (rule compile_erase[symmetric]) (simp add: call_formals_erase_pt)
  finally show ?thesis .
qed

lemma compile_proc_erase_cong:
  assumes "erase_pt Π = erase_pt Π'" "erase_decl d = erase_decl d'"
  shows "compile_proc Π p d n = compile_proc Π' p d' n"
proof -
  have b: "erase (body d) = erase (body d')"
    using assms(2) by (simp add: erase_decl_def)
  then have "csize (body d) = csize (body d')"
    by (metis csize_erase)
  with b show ?thesis
    by (simp add: compile_proc_def compile_erase_cong[OF assms(1) b])
qed

lemma compile_procs_erase_cong:
  assumes "erase_pt Π = erase_pt Π'"
  shows "compile_procs Π ps n = compile_procs Π' ps n"
proof (induction ps arbitrary: n)
  case (Cons q qs)
  have "map_option erase_decl (Π q) = map_option erase_decl (Π' q)"
    using fun_cong[OF assms, of q] by (simp add: erase_pt_def)
  then show ?case
    by (cases "Π q"; cases "Π' q") (simp_all add: Cons.IH compile_proc_erase_cong[OF assms])
qed simp

lemma main_body_erase_cong:
```

```
  assumes "erase_pt Π = erase_pt Π'"
  shows "erase (main_body Π) = erase (main_body Π')"
proof -
  have "map_option erase_decl (Π prog_main_name) = map_option erase_decl (Π' prog_main_name)"
    using fun_cong[OF assms, of prog_main_name] by (simp add: erase_pt_def)
  then show ?thesis
    by (cases "Π prog_main_name"; cases "Π' prog_main_name")
       (simp_all add: main_body_def erase_decl_def)
qed

lemma compile_prog_erase_cong:
  assumes "erase_pt Π = erase_pt Π'"
  shows "compile_prog Π ps = compile_prog Π' ps"
proof -
  have "compile_proc Π prog_main_name (|formals = [], body = main_body Π|) n
      = compile_proc Π' prog_main_name (|formals = [], body = main_body Π'|) n" for n
    using main_body_erase_cong[OF assms]
    by (intro compile_proc_erase_cong[OF assms]) (simp add: erase_decl_def)
  then show ?thesis
    by (simp add: compile_prog_def compile_procs_erase_cong[OF assms])
qed
```

Elaborating a procedure table

A table is elaborated procedure by procedure, each body at the offset at which `compile_procs` places it, and `main` after all of them.

```
lemma erase_elab [simp]: "erase (elab p k n c) = erase c"
  by (induction p k n c rule: elab.induct) (simp_all, safe, simp_all)

definition elab_decl :: "pname ⇒ nat ⇒ 'a gproc_decl ⇒ loc gproc_decl" where
  "elab_decl p n d =
     (|formals = formals d, body = elab p (Statement (n + csize (body d))) n (body d)|)"

lemma erase_decl_elab_decl [simp]: "erase_decl (elab_decl p n d) = erase_decl d"
  by (simp add: elab_decl_def erase_decl_def)
```

Each procedure is elaborated at the offset `compile_procs` places it at.

```
fun elab_procs :: "'a ptable ⇒ pname list ⇒ nat ⇒ loc ptable" where
  "elab_procs Π [] n = Map.empty"
| "elab_procs Π (p # ps) n =
     (case Π p of
        None ⇒ elab_procs Π ps n
      | Some d ⇒ (elab_procs Π ps (Suc (n + csize (body d))))(p ↦ elab_decl p n d))"

definition elab_pt :: "'a ptable ⇒ pname list ⇒ loc ptable" where
  "elab_pt Π ps =
     (elab_procs Π ps 0)(prog_main_name :=
        map_option (elab_decl prog_main_name (fst (compile_procs Π ps 0))) (Π prog_main_name))"

lemma elab_procs_None: "Π p = None ⟹ elab_procs Π ps n p = None"
  by (induction ps arbitrary: n) (auto split: option.split)

lemma elab_procs_erase:
  "map_option erase_decl (elab_procs Π ps n p)
     = (if p ∈ set ps then map_option erase_decl (Π p) else None)"
  by (induction ps arbitrary: n) (auto simp: elab_procs_None split: option.split)

lemma erase_pt_elab_pt:
  assumes "set ps = {p. Π p ≠ None} - {prog_main_name}"
  shows "erase_pt (elab_pt Π ps) = erase_pt Π"
proof
  fix p
  show "erase_pt (elab_pt Π ps) p = erase_pt Π p"
    using assms elab_procs_erase[of Π ps 0 p]
    by (cases "Π p") (auto simp: erase_pt_def elab_pt_def option.map_comp comp_def)
qed
```

The compiler output is located

The elaborated table satisfies `located` for its own compiled graph, which is what makes the simulation functional on it.

```
lemma elab_procs_entry:
  assumes "compile_procs Π ps n = (n', E, K)" "elab_procs Π ps n p = Some d'"
```

```
  shows "∃d m. Π p = Some d ∧ d' = elab_decl p m d ∧ (FunctionEntry p, EA_Body p, Statement m) ∈ E"
  using assms
proof (induction ps arbitrary: n n' E K)
  case Nil then show ?case by simp
next
  case (Cons q qs)
  show ?case
  proof (cases "Π q")
    case None
    with Cons.prems show ?thesis by (intro Cons.IH) simp_all
  next
    case (Some dq)
    obtain n1 Eq Kq where cq: "compile_proc Π q dq n = (n1, Eq, Kq)"
      by (rule prod_cases3)
    obtain n2 E' K' where cr: "compile_procs Π qs n1 = (n2, E', K')"
      by (rule prod_cases3)
    have E: "E = Eq ∪ E'" using Cons.prems(1) Some cq cr by simp
    have n1: "n1 = Suc (n + csize (body dq))" using cq by blast
    show ?thesis
    proof (cases "p = q")
      case True
      then show ?thesis using Cons.prems(2) Some compile_proc_entry_edge[OF cq] E by auto
    next
      case False
      with Cons.prems(2) Some n1 have "elab_procs Π qs n1 p = Some d'" by simp
      from Cons.IH[OF cr this] show ?thesis using E by blast
    qed
  qed
qed
```

The elaborated table compiles to the same graph as the source table, and every procedure's entry edge fixes the offset its body was elaborated at.

```
theorem located_compile_prog:
  fixes Π :: proc_table
  assumes wf: "wf_compile_input 𝒢 Π ps"
  shows "located (elab_pt Π ps) (compile_prog (elab_pt Π ps) ps)"
proof -
  from wf have setps: "set ps = {p. Π p ≠ None} - {prog_main_name}"
    by (simp add: wf_compile_input_def)
  have g: "compile_prog (elab_pt Π ps) ps = compile_prog Π ps"
    by (rule compile_prog_erase_cong) (rule erase_pt_elab_pt[OF setps])
  obtain n1 Eprocs Kprocs n2 Emain Kmain where
      procs: "compile_procs Π ps 0 = (n1, Eprocs, Kprocs)"
    and mainc: "compile_proc Π prog_main_name (|formals = [], body = main_body Π|) n1
                = (n2, Emain, Kmain)"
    and EI: "intra (compile_prog Π ps) = Eprocs ∪ Emain"
    by (rule compile_prog_intra_split)
  have entry: "∃d m. Π p = Some d ∧ decl = elab_decl p m d
                ∧ (FunctionEntry p, EA_Body p, Statement m) ∈ intra (compile_prog Π ps)"
    if "elab_pt Π ps p = Some decl" for p decl
  proof (cases "p = prog_main_name")
    case True
    from wf have md: "Π prog_main_name = Some (|formals = [], body = main_body Π|)"
      by (simp add: wf_compile_input_def wf_source_program_def)
    with that True procs
    have "decl = elab_decl prog_main_name n1 (|formals = [], body = main_body Π|)"
      by (simp add: elab_pt_def)
    with True md compile_proc_entry_edge[OF mainc] EI show ?thesis by blast
  next
    case False
    with that have "elab_procs Π ps 0 p = Some decl" by (simp add: elab_pt_def)
    from elab_procs_entry[OF procs this] EI show ?thesis by blast
  qed
  show ?thesis
    unfolding located_def g
  proof (intro allI impI)
    fix p b k n
    assume "compiled_at (elab_pt Π ps) (compile_prog Π ps) p b k n"
    then obtain decl where decl: "elab_pt Π ps p = Some decl" "b = body decl"
      and ent: "(FunctionEntry p, EA_Body p, Statement n) ∈ intra (compile_prog Π ps)"
      and k: "k = Statement (n + csize b)"
      unfolding compiled_at_def by blast
    from entry[OF decl(1)] obtain d m where d: "Π p = Some d" "decl = elab_decl p m d"
      and ent': "(FunctionEntry p, EA_Body p, Statement m) ∈ intra (compile_prog Π ps)"
      by blast
    have "Statement n = Statement m"
```

```
    by (rule compile_prog_entry_out_unique[OF wf ent ent'])
  then have nm: "n = m" by simp
  from wf d(1) have "wf_source_com Π (body d)"
    by (simp add: wf_compile_input_def wf_source_program_def wf_proc_decl_def)
  then have src: "source_com (body d)" by (rule wf_source_com_source_com)
  have "b = elab p k n (body d)"
    using decl(2) d(2) k nm by (simp add: elab_decl_def)
  with src show "∃c :: com. source_com c ∧ b = elab p k n c" by blast
 qed
qed

corollary csim_point_compile_prog:
 fixes Π :: proc_table
 assumes "wf_compile_input 𝒢 Π ps"
   and "elab_pt Π ps, compile_prog (elab_pt Π ps) ps, p ⊢ (c, s, frs) ≈ (v, t, stk)"
 shows "v = point c"
 using csim_point[OF assms(2) located_compile_prog[OF assms(1)]] by simp
```

Well-formedness ignores annotations

Every well-formedness predicate is invariant under erasure, so an elaborated table is well-formed exactly when its source table is.

```
lemma source_com_erase [simp]: "source_com (erase c) = source_com c"
 by (induction c) simp_all

lemma no_return_erase [simp]: "no_return (erase c) = no_return c"
 by (induction c) simp_all

lemma may_fallthrough_erase [simp]: "may_fallthrough (erase c) = may_fallthrough c"
 by (induction c) simp_all

lemma may_return_none_erase [simp]: "may_return_none (erase c) = may_return_none c"
 by (induction c) simp_all

lemma may_return_value_erase [simp]: "may_return_value (erase c) = may_return_value c"
 by (induction c) simp_all

lemma value_providing_erase [simp]: "value_providing (erase c) = value_providing c"
 by (simp add: value_providing_def)

lemma wf_source_com_erase: "wf_source_com (erase_pt Π) (erase c) = wf_source_com Π c"
 by (induction c) (simp_all add: erase_pt_def split: option.split)

lemma erase_pt_apply: "erase_pt Π p = map_option erase_decl (Π p)"
 by (simp add: erase_pt_def)

lemma main_body_erase_pt [simp]: "main_body (erase_pt Π) = erase (main_body Π)"
 by (cases "Π prog_main_name") (simp_all add: main_body_def erase_pt_def unit_eq)

lemma wf_proc_decl_erase:
 "wf_proc_decl 𝒢 (erase_pt Π) (erase_decl d) = wf_proc_decl 𝒢 Π d"
 by (simp add: wf_proc_decl_def wf_source_com_erase)

lemma wf_source_program_erase_pt:
 "wf_source_program 𝒢 (erase_pt Π) = wf_source_program 𝒢 Π"
proof -
 have main: "erase_pt Π prog_main_name = Some (|formals = [], body = main_body (erase_pt Π)|)
     ⟷ Π prog_main_name = Some (|formals = [], body = main_body Π|)"
 proof (cases "Π prog_main_name")
   case (Some d)
   then show ?thesis by (cases d) (simp add: erase_pt_def erase_decl_def main_body_def)
 qed (simp add: erase_pt_def)
 have procs: "(∀p decl. erase_pt Π p = Some decl ⟶ wf_proc_decl 𝒢 (erase_pt Π) decl)
     ⟷ (∀p decl. Π p = Some decl ⟶ wf_proc_decl 𝒢 Π decl)"
 proof
   assume l: "∀p decl. erase_pt Π p = Some decl ⟶ wf_proc_decl 𝒢 (erase_pt Π) decl"
   show "∀p decl. Π p = Some decl ⟶ wf_proc_decl 𝒢 Π decl"
   proof (intro allI impI)
     fix p decl assume "Π p = Some decl"
     then have "wf_proc_decl 𝒢 (erase_pt Π) (erase_decl decl)"
       using l[rule_format, of p "erase_decl decl"] by (simp add: erase_pt_apply)
     then show "wf_proc_decl 𝒢 Π decl" by (simp only: wf_proc_decl_erase)
   qed
 qed (auto simp: erase_pt_apply wf_proc_decl_erase)
 have dom: "(∀p. erase_pt Π p ≠ None ⟶ special_table p = None)
```

```
      ⟷ (∀p. Π p ≠ None ⟶ special_table p = None)"
    by (simp add: erase_pt_def)
  show ?thesis
    unfolding wf_source_program_def main procs dom by (simp add: wf_source_com_erase)
qed

lemma wf_compile_input_erase_pt:
  "wf_compile_input 𝒢 (erase_pt Π) ps = wf_compile_input 𝒢 Π ps"
  by (simp add: wf_compile_input_def wf_source_program_erase_pt erase_pt_apply)

lemma wf_compile_input_elab_pt:
  assumes "wf_compile_input 𝒢 Π ps"
  shows "wf_compile_input 𝒢 (elab_pt Π ps) ps"
  using assms wf_compile_input_erase_pt[of 𝒢 "elab_pt Π ps" ps] wf_compile_input_erase_pt[of 𝒢 Π ps]
    erase_pt_elab_pt[OF wf_compile_inputD(10)[OF assms]]
  by simp

lemma compile_prog_elab_pt:
  "wf_compile_input 𝒢 Π ps ⟹ compile_prog (elab_pt Π ps) ps = compile_prog Π ps"
  by (rule compile_prog_erase_cong) (rule erase_pt_elab_pt[OF wf_compile_inputD(10)])
```

Lifting plain runs to annotated runs

The semantics never reads an annotation, so every step of the erased program can be replayed by the annotated one.

```
lemma erase_eq_Unwind [simp]: "erase c = Unwind u ⟷ (∃j. c = Unwind j)"
  by (cases c) simp_all

lemma erase_eq_Restore [simp]: "erase c = Restore u ⟷ (∃j. c = Restore j)"
  by (cases c) simp_all
```

Every step of the erased program is the erasure of a step of the annotated one. Frames carry no annotation, so only the command is lifted; a call enters the annotated callee body.

```
theorem pstep_lift:
  assumes "𝒢, erase_pt Π ⊢ (erase c, s, frs) →p (c0', s', frs')"
  shows "∃c'. 𝒢, Π ⊢ (c, s, frs) →p (c', s', frs') ∧ erase c' = c0'"
  using assms
proof (induction c arbitrary: c0' s' frs')
  case (Seq c1 c2)
  from Seq.prems have "𝒢, erase_pt Π ⊢ (Seq (erase c1) (erase c2), s, frs) →p (c0', s', frs')"
    by simp
  then show ?case
  proof (elim SeqSE)
    fix i
    assume h: "erase c1 = SKIP i" "(c0', s', frs') = (erase c2, s, frs)"
    then obtain j where "c1 = SKIP j" by (cases c1) simp_all
    with h show ?thesis by (intro exI[of _ c2] conjI) simp_all
  next
    fix c1'' t fs
    assume eq: "(c0', s', frs') = (Seq c1'' (erase c2), t, fs)"
      and st: "𝒢, erase_pt Π ⊢ (erase c1, s, frs) →p (c1'', t, fs)"
    from Seq.IH(1)[OF st] obtain d where "𝒢, Π ⊢ (c1, s, frs) →p (d, t, fs)" "erase d = c1''"
      by blast
    with eq show ?thesis by (intro exI[of _ "Seq d c2"] conjI) simp_all
  next
    fix i
    assume h: "erase c1 = Unwind i" "(c0', s', frs') = (Unwind i, s, frs)"
      "∀i. erase c2 ≠ Restore i"
    then obtain j where j: "c1 = Unwind j" by auto
    from h(3) have "¬ is_Restore c2" by (cases c2) simp_all
    with h j show ?thesis by (intro exI[of _ "Unwind j"] conjI) (simp_all add: unit_eq)
  next
    fix j i fr dst frsa
    assume h:
      "(c0', s', frs') = (SKIP i, combine_assign dst (s ret_var) (combine_env 𝒢 fr s), frsa)"
      "erase c1 = Unwind j" "erase c2 = Restore i" "frs = Frame fr dst # frsa"
    then obtain j' i' where "c1 = Unwind j'" "c2 = Restore i'" by auto
    with h show ?thesis by (intro exI[of _ "SKIP i'"] conjI) (simp_all add: unit_eq)
  qed
next
  case (If i b c1 c2)
  from If.prems
  have step: "𝒢, erase_pt Π ⊢ (If () b (erase c1) (erase c2), s, frs) →p (c0', s', frs')"
    by simp
```

```
  show ?case
  proof (cases "truthy (⟦b⟧e s)")
    case True
    with step have "c0' = erase c1" "s' = s" "frs' = frs" by auto
    with True show ?thesis by (intro exI[of _ c1] conjI) simp_all
  next
    case False
    with step have "c0' = erase c2" "s' = s" "frs' = frs" by auto
    with False show ?thesis by (intro exI[of _ c2] conjI) simp_all
  qed
next
  case (While i b c)
  from While.prems have "𝒢, erase_pt Π ⊢ (While () b (erase c), s, frs) →p (c0', s', frs')"
    by simp
  then have "c0' = erase (If i b (Seq c (While i b c)) (SKIP i))" "s' = s" "frs' = frs"
    by (auto elim: WhileSE[elim_format])
  then show ?case by (intro exI[of _ "If i b (Seq c (While i b c)) (SKIP i)"] conjI) simp_all
next
  case (Call i dst p args)
  from Call.prems have "𝒢, erase_pt Π ⊢ (gcom.Call () dst p args, s, frs) →p (c0', s', frs')"
    by simp
  then show ?case
  proof (elim CallSE)
    fix decl0
    assume c': "(c0', s', frs') = (Seq (body decl0) (Restore ()),
                  bind_formals (formals decl0) (map (λe. ⟦e⟧e s) args) (enter_state 𝒢 s),
                  Frame s dst # frs)"
      and d0: "erase_pt Π p = Some decl0"
      and len: "length args = length (formals decl0)" and dist: "distinct (formals decl0)"
    from d0 obtain decl where d: "Π p = Some decl" "decl0 = erase_decl decl"
      by (auto simp: erase_pt_apply)
    have "𝒢, Π ⊢ (gcom.Call i dst p args, s, frs)
           →p (Seq (body decl) (Restore i),
              bind_formals (formals decl0) (map (λe. ⟦e⟧e s) args) (enter_state 𝒢 s),
              Frame s dst # frs)"
      by (rule pstep.Call[where decl = decl]) (use len dist d in simp_all)
    then show ?thesis
      using c' d(2) by (intro exI[of _ "Seq (body decl) (Restore i)"] conjI) simp_all
  next
    fix desc sc v x
    assume dx: "dst = Some x" and c': "(c0', s', frs') = (SKIP (), s(x := v), frs)"
      and st: "special_table p = Some desc" and cl: "classify_special desc args = Some sc"
      and sr: "special_result sc s v"
    have step: "𝒢, Π ⊢ (gcom.Call i (Some x) p args, s, frs) →p (SKIP i, s(x := v), frs)"
      by (rule pstep.Special[OF st cl sr])
    from c' have eqs: "c0' = SKIP ()" "s' = s(x := v)" "frs' = frs" by simp_all
    show ?thesis unfolding eqs dx by (rule exI[of _ "SKIP i"], rule conjI[OF step]) simp
  qed
qed auto

theorem psteps_lift:
  assumes "𝒢, erase_pt Π ⊢ (c0, s, frs) →p* (c0', s', frs')" and "erase c = c0"
  shows "∃c'. 𝒢, Π ⊢ (c, s, frs) →p* (c', s', frs') ∧ erase c' = c0'"
  using assms
proof (induction c0 s frs c0' s' frs' arbitrary: c rule: star_pstep_induct)
  case (refl c0 s frs)
  then show ?case by (intro exI[of _ c] conjI star.refl)
next
  case (step c0 s frs c1 s1 frs1 c2 s2 frs2)
  from pstep_lift[of 𝒢 Π c s frs c1 s1 frs1] step.hyps(1) step.prems
  obtain d where d: "𝒢, Π ⊢ (c, s, frs) →p (d, s1, frs1)" "erase d = c1"
    by blast
  from step.IH[OF d(2)] obtain c' where "𝒢, Π ⊢ (d, s1, frs1) →p* (c', s2, frs2)" "erase c' = c2"
    by blast
  with d(1) show ?case by (blast intro: star.step)
qed

lemma erase_com [simp]: "erase (c :: com) = c"
  by (induction c) simp_all

lemma erase_pt_proc_table [simp]: "erase_pt (Π :: proc_table) = Π"
proof
  fix p
  show "erase_pt Π p = Π p"
  proof (cases "Π p")
    case (Some d)
```

```
    then show ?thesis by (cases d) (simp add: erase_pt_apply erase_decl_def)
  qed (simp add: erase_pt_apply)
qed
```

The elaboration of a plain table erases back to it, so `psteps_lift` lifts every run of the plain program from `main` to a run of its elaboration.

```
lemma erase_elab_pt:
  fixes Π :: proc_table
  assumes "wf_compile_input 𝒢 Π ps"
  shows "erase_pt (elab_pt Π ps) = Π" and "erase (main_body (elab_pt Π ps)) = main_body Π"
proof -
  show ept: "erase_pt (elab_pt Π ps) = Π"
    using erase_pt_elab_pt[OF wf_compile_inputD(10)[OF assms]] by simp
  show "erase (main_body (elab_pt Π ps)) = main_body Π"
    using main_body_erase_pt[of "elab_pt Π ps"] ept by simp
qed
```

Erasing a simulation

Annotations are invisible to the simulation too: an annotated configuration related to a CFG configuration erases to a plain one related to the same CFG configuration. This is what lets a located result imply the plain one.

```
lemma erase_seq_after [simp]: "erase (seq_after c afters) = seq_after (erase c) (map erase afters)"
  by (induction afters arbitrary: c) simp_all

lemma control_at_erase:
  "control_at Π p c0 k n r v ⟹ control_at Π' p (erase c0) k n (erase r) v"
proof (induction rule: control_at.induct)
  case (SeqLeft c1 n r v c2 k)
  then show ?case using control_at.SeqLeft[of Π' p "erase c1" n "erase r" v "erase c2" k] by simp
next
  case (SeqRight c2 k n c1 r v)
  then show ?case using control_at.SeqRight[of Π' p "erase c2" k n "erase c1" "erase r" v] by simp
next
  case (IfLeft c1 k n r v i b c2)
  then show ?case
    using control_at.IfLeft[of Π' p "erase c1" k n "erase r" v "()" b "erase c2"] by simp
next
  case (IfRight c2 k n c1 r v i b)
  then show ?case
    using control_at.IfRight[of Π' p "erase c2" k n "erase c1" "erase r" v "()" b] by simp
next
  case (IfDone c1 j c2 i b k n)
  then show ?case using control_at.IfDone[of "erase c1" "()" "erase c2" Π' p "()" b k n] by auto
next
  case (WhileBody c n r v i b k)
  then show ?case using control_at.WhileBody[of Π' p "erase c" n "erase r" v "()" b k] by simp
qed (simp_all add: control_at.intros)

lemma compiled_at_erase:
  "compiled_at Π g p body_p k n ⟹ compiled_at (erase_pt Π) g p (erase body_p) k n"
  unfolding compiled_at_def
proof (elim exE conjE)
  fix decl n' en E K
  assume d: "Π p = Some decl" and b: "body_p = body decl"
    and c: "compile Π p body_p k n = (n', en, E, K)"
    and rest: "E ⊆ intra g" "K ⊆ calls g" "(k, EA_Ret None p, FunctionResult p) ∈ intra g"
      "(FunctionEntry p, EA_Body p, Statement n) ∈ intra g" "k = Statement (n + csize body_p)"
  have "compile (erase_pt Π) p (erase body_p) k n = compile Π p body_p k n"
    by (rule compile_erase_cong) simp_all
  with d b c rest show "∃decl n' en E K. erase_pt Π p = Some decl ∧ erase body_p = body decl
      ∧ compile (erase_pt Π) p (erase body_p) k n = (n', en, E, K)
      ∧ E ⊆ intra g ∧ K ⊆ calls g ∧ (k, EA_Ret None p, FunctionResult p) ∈ intra g
      ∧ (FunctionEntry p, EA_Body p, Statement n) ∈ intra g ∧ k = Statement (n + csize (erase body_p))"
    by (intro exI[of _ "erase_decl decl"] exI conjI) (simp_all add: erase_pt_apply)
qed

lemma unwinding_erase [simp]: "unwinding (erase u) = unwinding u"
  by (induction u rule: unwinding.induct) simp_all

lemma pop_ready_erase: "pop_ready w ⟹ pop_ready (erase w) ∧ pop_ann (erase w) = Some ()"
  by (cases w rule: pop_ready.cases) simp_all

theorem csim_erase:
```

```
  "Π, g, q ⊢ (c, s, frs) ≈ cfg ⟹ erase_pt Π, g, q ⊢ (erase c, s, frs) ≈ cfg"
proof (induction "(c, s, frs)" cfg arbitrary: c s frs rule: csim.induct)
  case (Base p body_p k n c v s)
  show ?case
    by (rule csim.Base[OF control_at_erase[OF Base(1)] compiled_at_erase[OF Base(2)]])
next
  case (Nested q inner s frs v stk pc body_pc kc nc i afters cont dst args site caller)
  have c1: "control_at (erase_pt Π) pc (erase body_pc) kc nc
              (seq_after (SKIP ()) (map erase afters)) cont"
    using control_at_erase[OF Nested(3)] by simp
  have c2: "control_at (erase_pt Π) pc (erase body_pc) kc nc
              (seq_after (gcom.Call () dst q args) (map erase afters)) site"
    using control_at_erase[OF Nested(4)] by simp
  show ?case
    using csim.Nested[OF Nested(2) c1 c2 compiled_at_erase[OF Nested(5)]] by simp
next
  case (Returning w i pc body_pc kc nc afters cont dst q args site callee caller)
  have c1: "control_at (erase_pt Π) pc (erase body_pc) kc nc
              (seq_after (SKIP ()) (map erase afters)) cont"
    using control_at_erase[OF Returning(3)] by simp
  have c2: "control_at (erase_pt Π) pc (erase body_pc) kc nc
              (seq_after (gcom.Call () dst q args) (map erase afters)) site"
    using control_at_erase[OF Returning(4)] by simp
  from pop_ready_erase[OF Returning(1)]
  have "pop_ready (erase w)" "pop_ann (erase w) = Some ()" by simp_all
  then show ?case
    using csim.Returning[OF _ _ c1 c2 compiled_at_erase[OF Returning(5)]] by simp
qed
```

The nodes a source configuration is observed at

A source configuration reached from `s0` is observed at node `v` when some run of the elaboration from `s0` reaches an annotated configuration that erases to it, names `v`, and is related to `v` by the simulation. The relation is not functional: a configuration reachable along two paths can be observed at two nodes.

```
fun at_node :: "imp_prog ⇒ store ⇒ com × store × frame list ⇒ cfg_node ⇒ bool"
    ("(_,_ ⊢/ _ ▷/ _)" [51, 51, 51, 51] 50) where
  "p, s0 ⊢ (r, s, frs) ▷ v ⟷
     (∃c stk. erase c = r ∧ v = point c
        ∧ declared_global p, elab_pt (prog_table p) (prog_procs p)
            ⊢ (main_body (elab_pt (prog_table p) (prog_procs p)), s0, []) →ₚ* (c, s, frs)
        ∧ elab_pt (prog_table p) (prog_procs p), prog_cfg p, prog_main_name
            ⊢ (c, s, frs) ≈ (v, s, stk))"
```

Erased, the simulation relates the plain configuration to the same node.

```
lemma at_node_csim:
  assumes wf: "wf_program_compile_input p" and at: "p, s0 ⊢ (r, s, frs) ▷ v"
  shows "∃stk. prog_table p, prog_cfg p, prog_main_name ⊢ (r, s, frs) ≈ (v, s, stk)"
proof -
  from at obtain c stk where ec: "erase c = r"
    and m: "elab_pt (prog_table p) (prog_procs p), prog_cfg p, prog_main_name
              ⊢ (c, s, frs) ≈ (v, s, stk)"
    by auto
  from csim_erase [OF m] ec erase_elab_pt(1) [OF wf]
  show ?thesis by auto
qed

end
```

Chapter 6

Voblint_Framework

6.1 CFG_Enumeration

```
theory CFG_Enumeration
  imports "Voblint_CFG.CFG_Def"
begin
```

Solver-facing CFG enumeration

Backward predecessor relations and deterministic edge enumeration over a compiled two-relation CFG. These are the equation-generation and code-generation views the TD solver reads to build and evaluate the constraint system; they are not part of the concrete CFG semantics — `valid_activation_trace` never refers to them. Ordinary control flow is enumerated over `intra`; procedure calls — entry routing and return combining — over `calls`. There is no unified edge set and no separate combine relation: a return is recovered from the same `calls` tuple that created the activation.

Every enumeration appears twice: as a set, which the proofs reason over, and as a list, which code generation and the solver's folds consume, identified by a `set_..._list` lemma whenever the edge set is finite. On the return side there is one of each — the set `call_targets` and the list `call_target_list` — and every other return view is an image or a `map` of those, so the views cannot come to disagree about which edges they enumerate.

Intra predecessors

The ordinary incoming transitions of a node: an intra edge is a total store transformer within one activation, so this is the transfer fold the equation system runs at every node — including the `EA_Ret e p` edges into `FunctionResult p`, by which return values are summarised as ordinary predecessor folding.

```
definition intra_predecessors :: "cfg ⇒ cfg_node ⇒ (cfg_node × edge_action) set" where
  "intra_predecessors g v = {(u, a). (u, a, v) ∈ intra g}"

lemma intra_predecessors_iff [simp]:
  "(u, a) ∈ intra_predecessors g v ⟷ (u, a, v) ∈ intra g"
  by (simp add: intra_predecessors_def)

lemma finite_intra_predecessors [intro]:
  assumes "finite (intra g)"
  shows "finite (intra_predecessors g v)"
proof -
  have "intra_predecessors g v
        ⊆ (λe :: cfg_node × edge_action × cfg_node. (fst e, fst (snd e))) ‘ intra g"
    unfolding intra_predecessors_def by force
  then show ?thesis using assms finite_subset finite_imageI by blast
qed

definition intra_predecessor_list ::
    "cfg ⇒ cfg_node ⇒ (cfg_node × edge_action) list" where
  "intra_predecessor_list g v =
     map (λ(u, a, w). (u, a)) (filter (λ(u, a, w). w = v) (cfg_intra_list g))"

lemma set_intra_predecessor_list [simp]:
  assumes "finite (intra g)"
  shows "set (intra_predecessor_list g v) = intra_predecessors g v"
  unfolding intra_predecessor_list_def intra_predecessors_def
  using set_cfg_intra_list[OF assms] by (force simp: image_iff)
```

Call-entry enumeration

The call tuples whose callee entry is the queried node `v` (a `FunctionEntry p`). The continuation does not affect the callee-entry state, so it is projected away; the entry contribution is the caller state routed through the call action's parameter binding, which is what an analysis's entry operation computes.

```
definition entry_calls :: "cfg ⇒ cfg_node ⇒ (cfg_node × call_action) set" where
  "entry_calls g v = {(c, ca). ∃k. (c, ca, v, k) ∈ calls g}"

lemma entry_calls_iff [simp]:
  "(c, ca) ∈ entry_calls g v ⟷ (∃k. (c, ca, v, k) ∈ calls g)"
  by (simp add: entry_calls_def)

lemma finite_entry_calls [intro]:
  assumes "finite (calls g)"
  shows "finite (entry_calls g v)"
proof -
  have "entry_calls g v
        ⊆ (λe :: cfg_node × call_action × cfg_node × cfg_node. (fst e, fst (snd e))) ‘ calls g"
```

```
    unfolding entry_calls_def by force
  then show ?thesis using assms finite_subset finite_imageI by blast
qed

definition entry_call_list :: "cfg ⇒ cfg_node ⇒ (cfg_node × call_action) list" where
  "entry_call_list g v =
     map (λ(c, ca, ce, k). (c, ca)) (filter (λ(c, ca, ce, k). ce = v) (cfg_calls_list g))"

lemma set_entry_call_list [simp]:
  assumes "finite (calls g)"
  shows "set (entry_call_list g v) = entry_calls g v"
  unfolding entry_call_list_def entry_calls_def
  using set_cfg_calls_list[OF assms] by (force simp: image_iff)
```

Outgoing call enumeration (caller perspective)

The call tuples whose call site is the queried node `v`: the entry, action, and continuation of every call `v` makes. This is the caller-side counterpart of `entry_calls` (callee-indexed) — routing a call-entry seed publication at its call site needs the callee entry and the whole `call_action` together, which the continuation-indexed enumerations below do not expose.

```
definition call_successors ::
    "cfg ⇒ cfg_node ⇒ (cfg_node × call_action × cfg_node) set" where
  "call_successors g v = {(ce, ca, k). (v, ca, ce, k) ∈ calls g}"

lemma call_successors_iff [simp]:
  "(ce, ca, k) ∈ call_successors g v ⟷ (v, ca, ce, k) ∈ calls g"
  by (simp add: call_successors_def)

lemma finite_call_successors [intro]:
  assumes "finite (calls g)"
  shows "finite (call_successors g v)"
proof -
  have "call_successors g v
        ⊆ (λ(c, ca, ce, k). (ce, ca, k)) ‘ calls g"
    unfolding call_successors_def by force
  then show ?thesis using assms finite_subset finite_imageI by blast
qed

definition call_successor_list ::
    "cfg ⇒ cfg_node ⇒ (cfg_node × call_action × cfg_node) list" where
  "call_successor_list g v =
     map (λ(c, ca, ce, k). (ce, ca, k)) (filter (λ(c, ca, ce, k). c = v) (cfg_calls_list g))"

lemma set_call_successor_list [simp]:
  assumes "finite (calls g)"
  shows "set (call_successor_list g v) = call_successors g v"
  unfolding call_successor_list_def call_successors_def
  using set_cfg_calls_list[OF assms] by (force simp: image_iff)
```

Call targets at a continuation

The call edges whose continuation is `v`, each paired with the procedure its callee entry names. This is the one continuation-indexed enumeration that reads `calls` directly; every return-side view below is an image or a map of it.

`call_target_at` decodes a single call edge, selecting on the continuation and reading the callee's procedure off `FunctionEntry p` in one total function, so there is no unreachable-but-present branch and no later definition has to re-establish that the node it is looking at really is a callee entry.

Naming the callee by its procedure rather than by the `FunctionResult p` node its result is read at keeps the call site's own data — where the call is, what it passes, where it returns to — separate from the callee, which is what lets a resolver choose among candidates rather than read a callee the enumeration already fixed. The result node is then `FunctionResult p` at the point of use.

```
definition call_targets ::
    "cfg ⇒ cfg_node ⇒ (cfg_node × call_action × pname) set" where
  "call_targets g v = {(c, ca, p). (c, ca, FunctionEntry p, v) ∈ calls g}"

lemma call_targets_iff [simp]:
  "(c, ca, p) ∈ call_targets g v ⟷ (c, ca, FunctionEntry p, v) ∈ calls g"
  by (simp add: call_targets_def)

fun call_target_at ::
    "cfg_node ⇒ (cfg_node × call_action × cfg_node × cfg_node)
```

```
      ⇒ (cfg_node × call_action × pname) option" where
  "call_target_at v (c, ca, FunctionEntry p, k) =
     (if k = v then Some (c, ca, p) else None)"
| "call_target_at v _ = None"

definition call_target_list ::
    "cfg ⇒ cfg_node ⇒ (cfg_node × call_action × pname) list" where
  "call_target_list g v = List.map_filter (call_target_at v) (cfg_calls_list g)"

lemma set_map_filter_call_target_at:
  "set (List.map_filter (call_target_at v) es)
     = {(c, ca, p). (c, ca, FunctionEntry p, v) ∈ set es}"
proof (induction es)
  case Nil
  show ?case by simp
next
  case (Cons e es)
  obtain c ca ce k where e: "e = (c, ca, ce, k)" by (cases e) auto
  show ?case by (cases ce) (auto simp: e Cons.IH)
qed
```

Membership, in the form a call-site obligation states it: the enumeration lists exactly the call edges of `g` whose continuation is `v`, each paired with the procedure its callee entry names.

```
lemma set_call_target_list [simp]:
  assumes "finite (calls g)"
  shows "set (call_target_list g v) = call_targets g v"
  unfolding call_target_list_def call_targets_def
  by (simp add: set_map_filter_call_target_at set_cfg_calls_list[OF assms])

lemma finite_call_targets [intro]:
  assumes "finite (calls g)"
  shows "finite (call_targets g v)"
proof -
  have "call_targets g v = set (call_target_list g v)"
    using set_call_target_list[OF assms] by simp
  then show ?thesis by simp
qed
```

The enumeration lists each call edge once: `cfg_calls_list` is duplicate-free and `call_target_at` discards exactly the components a decoded entry does not determine. This is what makes the site and target lists below duplicate-free too, so the generator builds one contribution per site-target pair rather than one per pair of co-listed callees.

```
lemma distinct_map_filter_call_target_at:
  assumes "distinct es"
  shows "distinct (List.map_filter (call_target_at v) es)"
  using assms
proof (induction es)
  case Nil
  show ?case by simp
next
  case (Cons e es)
  obtain c ca ce k where e: "e = (c, ca, ce, k)" by (cases e) auto
  show ?case
    using Cons by (cases ce) (auto simp: e set_map_filter_call_target_at)
qed

lemma distinct_call_target_list [simp]:
  assumes "finite (calls g)"
  shows "distinct (call_target_list g v)"
  unfolding call_target_list_def
  by (rule distinct_map_filter_call_target_at) (simp add: cfg_calls_list_def assms)

lemma call_target_list_iff:
  assumes "finite (calls g)"
  shows "(c, ca, p) ∈ set (call_target_list g v)
         ⟷ (c, ca, FunctionEntry p, v) ∈ calls g"
  by (simp add: set_call_target_list[OF assms])
```

Return enumeration

The return edges, as an image of `call_targets` that keeps the triggering `call_action` whole. The return contribution is the caller state and the callee-exit state assembled by `combine`$^\sharp$. The callee's `FunctionResult p` node comes back directly — recovered from the callee entry `FunctionEntry p` — so no separate combine relation is needed and no client has to rebuild the result node.

The action is kept whole rather than packaged into the call's `call_info`, because the generator must route the callee-entry seed publication and the return-side context read from the *same* action, exactly as `entry_call_list` keeps `ca` whole on the entry side. This layer is the only one holding the `calls` relation, so whatever a consumer needs about a call site is projected here: a combine tree never sees the CFG, and `wf_cfg` does not force a call site's outgoing call edge to be unique, so there is no well-defined downstream lookup that could recover it instead.

```
definition return_call_actions ::
    "cfg ⇒ cfg_node ⇒ (cfg_node × call_action × cfg_node) set" where
  "return_call_actions g v =
     (λ(c, ca, p). (c, ca, FunctionResult p)) ‘ call_targets g v"

lemma return_call_actions_iff [simp]:
  "(c, ca, r) ∈ return_call_actions g v
     ⟷ (∃p. r = FunctionResult p ∧ (c, ca, FunctionEntry p, v) ∈ calls g)"
  by (auto simp: return_call_actions_def image_iff Bex_def)

lemma finite_return_call_actions [intro]:
  assumes "finite (calls g)"
  shows "finite (return_call_actions g v)"
  using finite_call_targets[OF assms] by (simp add: return_call_actions_def)

definition return_call_action_list ::
    "cfg ⇒ cfg_node ⇒ (cfg_node × call_action × cfg_node) list" where
  "return_call_action_list g v =
     map (λ(c, ca, p). (c, ca, FunctionResult p)) (call_target_list g v)"

lemma set_return_call_action_list [simp]:
  assumes "finite (calls g)"
  shows "set (return_call_action_list g v) = return_call_actions g v"
  unfolding return_call_action_list_def return_call_actions_def
  by (simp add: set_call_target_list[OF assms])
```

Call sites and the resolver adapter

The call site alone, with no callee: what a call-site-owned resolver is given. The list is duplicate-free, so a site with several statically known callees is listed once; with `distinct_static_targets` below, resolving every listed site therefore visits each site-target pair exactly once.

`static_targets` below is the resolver that answers from the CFG and ignores the caller's abstract state — the behaviour a statically enumerated generator already has. A resolver reading that state instead is what makes an indirect call expressible, and the shape of the equation does not change when it does. Two limits are worth stating, because the resolver type `cfg ⇒ cfg_node ⇒ cfg_node ⇒ call_action ⇒ 'd ⇒ pname list` does not: a resolver may answer with any procedures at all, so a sound instance needs an explicit coverage assumption relating its answer to the concrete callees reachable at the site; and the sites themselves come from `call_target_list`, so a call with no statically enumerated target is not listed and no resolver is consulted for it. What this supports is therefore state-dependent choice among enumerated candidates, not a fully late-bound call whose site exists independently of any known target.

```
definition call_site_list ::
    "cfg ⇒ cfg_node ⇒ (cfg_node × call_action) list" where
  "call_site_list g v = remdups (map (λ(c, ca, p). (c, ca)) (call_target_list g v))"

lemma distinct_call_site_list [simp]:
  "distinct (call_site_list g v)"
  by (simp add: call_site_list_def)

lemma set_call_site_list [simp]:
  assumes "finite (calls g)"
  shows "set (call_site_list g v)
           = {(c, ca) | c ca. ∃p. (c, ca, FunctionEntry p, v) ∈ calls g}"
  unfolding call_site_list_def
  by (force simp: set_call_target_list[OF assms] call_targets_def image_iff)

definition static_targets ::
    "cfg ⇒ cfg_node ⇒ cfg_node ⇒ call_action ⇒ pname list" where
  "static_targets g v cc ca =
     map (λ(c, a, p). p)
       (filter (λ(c, a, p). c = cc ∧ a = ca) (call_target_list g v))"

lemma static_targets_iff [simp]:
  assumes "finite (calls g)"
  shows "p ∈ set (static_targets g v cc ca)
           ⟷ (cc, ca, FunctionEntry p, v) ∈ calls g"
```

```
  unfolding static_targets_def
  by (force simp: call_target_list_iff[OF assms, symmetric] image_iff)

lemma distinct_static_targets [simp]:
  assumes "finite (calls g)"
  shows "distinct (static_targets g v cc ca)"
proof -
  let ?es = "filter (λ(c, a, p). c = cc ∧ a = ca) (call_target_list g v)"
  have "inj_on (λ(c, a, p). p) (set ?es)" by (auto simp: inj_on_def)
  moreover have "distinct ?es" using distinct_call_target_list[OF assms] by simp
  ultimately show ?thesis unfolding static_targets_def by (simp add: distinct_map)
qed
```

Exactly when a site has a statically known callee, which is the form a coverage obligation stated over the enumeration needs.

```
lemma static_targets_neq_Nil_iff [simp]:
  assumes "finite (calls g)"
  shows "static_targets g v cc ca ≠ []
         ⟷ (∃p. (cc, ca, FunctionEntry p, v) ∈ calls g)"
proof -
  have "static_targets g v cc ca ≠ [] ⟷ (∃p. p ∈ set (static_targets g v cc ca))"
    by (cases "static_targets g v cc ca") auto
  thus ?thesis using static_targets_iff[OF assms] by blast
qed

definition static_resolve ::
  "cfg ⇒ cfg_node ⇒ cfg_node ⇒ call_action ⇒ 'd ⇒ pname list" where
  "static_resolve g v cc ca d = static_targets g v cc ca"

lemma static_resolve_iff [simp]:
  assumes "finite (calls g)"
  shows "p ∈ set (static_resolve g v cc ca d)
         ⟷ (cc, ca, FunctionEntry p, v) ∈ calls g"
  unfolding static_resolve_def by (rule static_targets_iff[OF assms])
```

Which unknowns a run has to have solved

`vars_cover g vars` is the one recurring premise every post-solution soundness theorem here needs: the solved key set `vars` contains the CFG entry, the target of every `intra` edge, and both halves of every call — the callee's entry and the caller's continuation.

Read it as a statement about a solver run, not about the graph. It quantifies over every edge `g` has, so a graph carrying a call the run never activated does not satisfy it: in a routed system a callee's entry unknown is solved only once some caller publishes its seed, and an unreachable call site publishes nothing. That is why a run establishes it by deciding `vars_cover_exec` against the keys it actually solved, on a graph *pruned* to what it can reach, rather than by an argument about connectivity.

The four components travel together, so callers state and discharge one premise instead of four positional ones. Global, not locale-local: every analysis instance and the executable pipeline cite it under this name.

```
definition vars_cover :: "cfg ⇒ (cfg_node × unit) set ⇒ bool" where
  "vars_cover g vars ⟷
     (cfg_entry g, ()) ∈ vars
   ∧ (∀u a v. (u, a, v) ∈ intra g ⟶ (v, ()) ∈ vars)
   ∧ (∀c dst fs as q k. (c, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g
        ⟶ (FunctionEntry q, ()) ∈ vars ∧ (k, ()) ∈ vars)"

lemma vars_coverI [intro]:
  assumes "(cfg_entry g, ()) ∈ vars"
    and "⋀u a v. (u, a, v) ∈ intra g ⟹ (v, ()) ∈ vars"
    and "⋀c dst fs as q k. (c, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g
           ⟹ (FunctionEntry q, ()) ∈ vars"
    and "⋀c dst fs as q k. (c, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g
           ⟹ (k, ()) ∈ vars"
  shows "vars_cover g vars"
  unfolding vars_cover_def using assms by blast

lemma vars_cover_entryD [dest]: "vars_cover g vars ⟹ (cfg_entry g, ()) ∈ vars"
  unfolding vars_cover_def by blast

lemma vars_cover_edgeD [dest]:
  "vars_cover g vars ⟹ (u, a, v) ∈ intra g ⟹ (v, ()) ∈ vars"
  unfolding vars_cover_def by blast

lemma vars_cover_enterD [dest]:
```

```
  assumes cover: "vars_cover g vars"
  shows "⋀c dst fs as q k. (c, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g
     ⟹ (FunctionEntry q, ()) ∈ vars"
  using cover unfolding vars_cover_def by blast

lemma vars_cover_combineD [dest]:
  assumes cover: "vars_cover g vars"
  shows "⋀c dst fs as q k. (c, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g
     ⟹ (k, ()) ∈ vars"
  using cover unfolding vars_cover_def by blast
```

Coverage of a finite graph is testable: walk the two edge enumerations and test membership. The test is sufficient, not equivalent, and deliberately so. `vars_cover` asks only about calls of the shape `(c, CallEdge dst fs as, FunctionEntry q, k)`, while the test asks about every tuple the enumeration yields, so it is the stronger of the two: it also covers a `calls` entry whose action or target has some other shape. Turning it into an equivalence would mean assuming that `g` has no such entry, which is a well-formedness fact about the graph and not something a coverage check should carry.

```
definition vars_cover_exec :: "cfg ⇒ (cfg_node × unit) set ⇒ bool" where
  "vars_cover_exec g vars ⟷
     (cfg_entry g, ()) ∈ vars
   ∧ list_all (λ(u, a, v). (v, ()) ∈ vars) (cfg_intra_list g)
   ∧ list_all (λ(c, ca, ce, k). (ce, ()) ∈ vars ∧ (k, ()) ∈ vars) (cfg_calls_list g)"
```

Coverage when the unknowns carry a context

`vars_cover` is fixed at the unit context: its unknowns are nodes, so it can demand that every successor of every edge was solved. A context-sensitive system cannot demand that. Its unknowns are node-context pairs, and which contexts a node was solved at is decided by the run, so coverage becomes *closure*: whenever a pair was solved, the pairs its edges lead to were solved too. Hence the conditional shape here against the unconditional shape above.

Where a call leads is not a fact about the graph. A context policy computes the callee's context from the call site, the caller's context, the callee, and in general the state published at the call — which is why `succ_ctx` is a parameter rather than something read off `g`. An instance supplies its own route already applied to its own solved environment.

The return side takes no such argument: a combine resumes the caller's own activation, so the continuation is covered at the context the call was made in.

```
definition ctx_vars_cover ::
    "cfg ⇒ (cfg_node ⇒ 'c ⇒ call_action ⇒ pname ⇒ 'c) ⇒ 'c
       ⇒ (cfg_node × 'c) set ⇒ bool" where
  "ctx_vars_cover g succ_ctx rc vars ⟷
     (cfg_entry g, rc) ∈ vars
   ∧ (∀u a v ctx. (u, ctx) ∈ vars ⟶ (u, a, v) ∈ intra g ⟶ (v, ctx) ∈ vars)
   ∧ (∀u ctx dst fs as q k. (u, ctx) ∈ vars
        ⟶ (u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g
        ⟶ (FunctionEntry q, succ_ctx u ctx (CallEdge dst fs as) q) ∈ vars
          ∧ (k, ctx) ∈ vars)"

lemma ctx_vars_coverI [intro]:
  assumes "(cfg_entry g, rc) ∈ vars"
    and "⋀u a v ctx. (u, ctx) ∈ vars ⟹ (u, a, v) ∈ intra g ⟹ (v, ctx) ∈ vars"
    and "⋀u ctx dst fs as q k. (u, ctx) ∈ vars
           ⟹ (u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g
           ⟹ (FunctionEntry q, succ_ctx u ctx (CallEdge dst fs as) q) ∈ vars"
    and "⋀u ctx dst fs as q k. (u, ctx) ∈ vars
           ⟹ (u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g ⟹ (k, ctx) ∈ vars"
  shows "ctx_vars_cover g succ_ctx rc vars"
  unfolding ctx_vars_cover_def using assms by blast

lemma ctx_vars_cover_entryD [dest]:
  "ctx_vars_cover g succ_ctx rc vars ⟹ (cfg_entry g, rc) ∈ vars"
  unfolding ctx_vars_cover_def by blast

lemma ctx_vars_cover_edgeD [dest]:
  "ctx_vars_cover g succ_ctx rc vars ⟹ (u, ctx) ∈ vars ⟹ (u, a, v) ∈ intra g
     ⟹ (v, ctx) ∈ vars"
  unfolding ctx_vars_cover_def by blast

lemma ctx_vars_cover_enterD [dest]:
  assumes cover: "ctx_vars_cover g succ_ctx rc vars"
  shows "⋀u ctx dst fs as q k. (u, ctx) ∈ vars
     ⟹ (u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g
```

```
    ⟹ (FunctionEntry q, succ_ctx u ctx (CallEdge dst fs as) q) ∈ vars"
  using cover unfolding ctx_vars_cover_def by blast

lemma ctx_vars_cover_combineD [dest]:
  assumes cover: "ctx_vars_cover g succ_ctx rc vars"
  shows "⋀u ctx dst fs as q k. (u, ctx) ∈ vars
    ⟹ (u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g ⟹ (k, ctx) ∈ vars"
  using cover unfolding ctx_vars_cover_def by blast
```

Contextual coverage is testable the way the unit one is, with one difference that decides the shape: closure quantifies over the pairs the run solved, and those are not enumerable from the graph. So the test walks the solved keys against the two edge enumerations rather than walking the graph alone, which is why it takes them as the set the solver returns and iterates over it.

Sufficient rather than equivalent, and deliberately so. At a call the test asks for the continuation at every action shape, while `ctx_vars_cover` asks only about `CallEdge`; demanding the callee entry off a non-call action would need a context the policy never computes, so that one stays guarded.

```
definition ctx_vars_cover_exec ::
    "cfg ⇒ (cfg_node ⇒ 'c ⇒ call_action ⇒ pname ⇒ 'c) ⇒ 'c
       ⇒ (cfg_node × 'c) set ⇒ bool" where
  "ctx_vars_cover_exec g succ_ctx rc vs ⟷
     (cfg_entry g, rc) ∈ vs
   ∧ (∀(u, ctx) ∈ vs.
        list_all (λ(u', a, v). u' ≠ u ∨ (v, ctx) ∈ vs) (cfg_intra_list g)
      ∧ list_all (λ(c, ca, ce, k). c ≠ u ∨ ((k, ctx) ∈ vs
            ∧ (case ce of FunctionEntry q ⇒ (ce, succ_ctx u ctx ca q) ∈ vs
                 | _ ⇒ True)))
          (cfg_calls_list g))"

lemma ctx_vars_cover_of_exec:
  assumes finE: "finite (intra g)" and finC: "finite (calls g)"
    and cover: "ctx_vars_cover_exec g succ_ctx rc (vs :: (cfg_node × 'c) set)"
  shows "ctx_vars_cover g succ_ctx rc vs"
proof (intro ctx_vars_coverI)
  show "(cfg_entry g, rc) ∈ vs"
    using cover unfolding ctx_vars_cover_exec_def by blast
next
  fix u a v ctx
  assume "(u, ctx) ∈ vs" and "(u, a, v) ∈ intra g"
  then show "(v, ctx) ∈ vs"
    using cover unfolding ctx_vars_cover_exec_def
    by (fastforce simp: list_all_iff set_cfg_intra_list [OF finE])
next
  fix u ctx dst fs as q k
  assume "(u, ctx) ∈ vs"
    and "(u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g"
  then show "(FunctionEntry q, succ_ctx u ctx (CallEdge dst fs as) q) ∈ vs"
    using cover unfolding ctx_vars_cover_exec_def
    by (fastforce simp: list_all_iff set_cfg_calls_list [OF finC])
next
  fix u ctx dst fs as q k
  assume "(u, ctx) ∈ vs"
    and "(u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g"
  then show "(k, ctx) ∈ vs"
    using cover unfolding ctx_vars_cover_exec_def
    by (fastforce simp: list_all_iff set_cfg_calls_list [OF finC])
qed
```

Coverage along live dependencies

`ctx_vars_cover` demands closure from every solved pair, including pairs whose solved state is bottom and calls whose entered state is bottom. A demand-driven solver never solves what such a pair would lead to, so the unconditional closure fails on dead call sites and on procedures nothing calls, while soundness only ever follows a dependency from a state that holds a concrete store. The live form asks for exactly that much.

`live u ctx` says the state solved at `(u, ctx)` may hold a store; an intra successor is demanded only from a live pair. `succ u ctx ca q` is the callee context a call routes to, or `None` when the entered state is bottom and the call contributes nothing; the callee entry is demanded only when it is `Some`. The continuation stays unconditional: a call site is queried only by its own continuation's call tree, so a solver never solves the one without the other.

```
definition ctx_vars_cover_live ::
    "cfg ⇒ (cfg_node ⇒ 'c ⇒ bool) ⇒ (cfg_node ⇒ 'c ⇒ call_action ⇒ pname ⇒ 'c option)
       ⇒ 'c ⇒ (cfg_node × 'c) set ⇒ bool" where
  "ctx_vars_cover_live g live succ rc vars ⟷
```

```
    (cfg_entry g, rc) ∈ vars
  ∧ (∀u a v ctx. (u, ctx) ∈ vars ⟶ live u ctx ⟶ (u, a, v) ∈ intra g
      ⟶ (v, ctx) ∈ vars)
  ∧ (∀u ctx dst fs as q k. (u, ctx) ∈ vars
      ⟶ (u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g
      ⟶ (k, ctx) ∈ vars
        ∧ (∀ctx'. succ u ctx (CallEdge dst fs as) q = Some ctx'
            ⟶ (FunctionEntry q, ctx') ∈ vars))"

lemma ctx_vars_cover_liveI [intro]:
  assumes "(cfg_entry g, rc) ∈ vars"
    and "⋀u a v ctx. (u, ctx) ∈ vars ⟹ live u ctx ⟹ (u, a, v) ∈ intra g
          ⟹ (v, ctx) ∈ vars"
    and "⋀u ctx dst fs as q k ctx'. (u, ctx) ∈ vars
          ⟹ (u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g
          ⟹ succ u ctx (CallEdge dst fs as) q = Some ctx'
          ⟹ (FunctionEntry q, ctx') ∈ vars"
    and "⋀u ctx dst fs as q k. (u, ctx) ∈ vars
          ⟹ (u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g ⟹ (k, ctx) ∈ vars"
  shows "ctx_vars_cover_live g live succ rc vars"
  unfolding ctx_vars_cover_live_def using assms by blast

lemma ctx_vars_cover_live_entryD [dest]:
  "ctx_vars_cover_live g live succ rc vars ⟹ (cfg_entry g, rc) ∈ vars"
  unfolding ctx_vars_cover_live_def by blast

lemma ctx_vars_cover_live_edgeD [dest]:
  "ctx_vars_cover_live g live succ rc vars ⟹ (u, ctx) ∈ vars ⟹ live u ctx
    ⟹ (u, a, v) ∈ intra g ⟹ (v, ctx) ∈ vars"
  unfolding ctx_vars_cover_live_def by blast

lemma ctx_vars_cover_live_enterD [dest]:
  "ctx_vars_cover_live g live succ rc vars ⟹ (u, ctx) ∈ vars
    ⟹ (u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g
    ⟹ succ u ctx (CallEdge dst fs as) q = Some ctx' ⟹ (FunctionEntry q, ctx') ∈ vars"
  unfolding ctx_vars_cover_live_def by blast

lemma ctx_vars_cover_live_combineD [dest]:
  "ctx_vars_cover_live g live succ rc vars ⟹ (u, ctx) ∈ vars
    ⟹ (u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls g ⟹ (k, ctx) ∈ vars"
  unfolding ctx_vars_cover_live_def by blast

end
```

6.2 DG_State

```
theory DG_State
  imports "Voblint_Domain.State_Concretization"
begin
```

The value a D/G unknown carries

An analysis chooses a flow-sensitive answer domain `D` and a flow-insensitive side-effect domain `G`. The framework keeps them opaque and stores them in separate components of `dg_state`; it never copies a global component into a local answer.

This theory is that carrier and its order, and nothing else: no address, no solver tree, no transfer. Everything above it only projects and repacks the two components, which is what makes those constructions independent of the concrete domains.

A lattice copy type for D-times-G unknown values

The solver's single value type must order local and global halves componentwise. Raw pairs cannot: this repository loads `HOL-Library.Product_Lexorder`, so every `'l * 'g` in scope already carries the lexicographic order, and a componentwise instance for the same type would clash with it. `dg_state` is a distinct carrier so that both orders can coexist.

```
datatype ('l, 'g) dg_state = DG (dg_local: 'l) (dg_global: 'g)

instantiation dg_state :: (ord, ord) ord
begin

definition less_eq_dg_state :: "('a, 'b) dg_state ⇒ ('a, 'b) dg_state ⇒ bool" where
  "less_eq_dg_state d1 d2 = (dg_local d1 ≤ dg_local d2 ∧ dg_global d1 ≤ dg_global d2)"

definition less_dg_state :: "('a, 'b) dg_state ⇒ ('a, 'b) dg_state ⇒ bool" where
  "less_dg_state d1 d2 = (d1 ≤ d2 ∧ ¬ d2 ≤ d1)"

instance ..

end

instance dg_state :: (order, order) order
proof intro_classes
  fix x y z :: "('a, 'b) dg_state"
  show "x < y ⟷ x ≤ y ∧ ¬ y ≤ x" by (simp add: less_dg_state_def)
  show "x ≤ x" by (simp add: less_eq_dg_state_def)
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
    by (auto simp: less_eq_dg_state_def intro: order_trans)
  show "x ≤ y ⟹ y ≤ x ⟹ x = y"
    by (auto simp: less_eq_dg_state_def intro: dg_state.expand antisym)
qed

instantiation dg_state :: (semilattice_sup, semilattice_sup) semilattice_sup
begin

definition sup_dg_state :: "('a, 'b) dg_state ⇒ ('a, 'b) dg_state ⇒ ('a, 'b) dg_state" where
  "sup_dg_state d1 d2 = DG (dg_local d1 ⊔ dg_local d2) (dg_global d1 ⊔ dg_global d2)"

instance
  by standard (auto simp: less_eq_dg_state_def sup_dg_state_def)

end

instantiation dg_state :: (order_bot, order_bot) order_bot
begin

definition bot_dg_state :: "('a, 'b) dg_state" where
  "bot_dg_state = DG bot bot"

instance
  by standard (simp add: less_eq_dg_state_def bot_dg_state_def)

end

instance dg_state ::
  (bounded_semilattice_sup_bot, bounded_semilattice_sup_bot) bounded_semilattice_sup_bot ..
```

Reading the two components back

Every construction above this theory projects and repacks, so the four projection equations and the constructor comparison are what its proofs should meet instead of the instance definitions. The order itself stays unfolded: `less_eq_dg_state_def` is not a simp rule, because a bound between two values that are not both constructor terms reads better as one $\leq$ than as a conjunction of two.

```
lemma dg_local_sup [simp]:
  fixes x y :: "('a::semilattice_sup, 'b::semilattice_sup) dg_state"
  shows "dg_local (x ⊔ y) = dg_local x ⊔ dg_local y"
  by (simp add: sup_dg_state_def)

lemma dg_global_sup [simp]:
  fixes x y :: "('a::semilattice_sup, 'b::semilattice_sup) dg_state"
  shows "dg_global (x ⊔ y) = dg_global x ⊔ dg_global y"
  by (simp add: sup_dg_state_def)

lemma dg_local_bot [simp]:
  "dg_local (bot :: ('a::order_bot, 'b::order_bot) dg_state) = bot"
  by (simp add: bot_dg_state_def)

lemma dg_global_bot [simp]:
  "dg_global (bot :: ('a::order_bot, 'b::order_bot) dg_state) = bot"
  by (simp add: bot_dg_state_def)

lemma DG_le_DG [simp]:
  fixes d d' :: "'a::ord" and g g' :: "'b::ord"
  shows "DG d g ≤ DG d' g' ⟷ d ≤ d' ∧ g ≤ g'"
  by (simp add: less_eq_dg_state_def)

instantiation dg_state ::
  ("{bounded_semilattice_sup_bot, warrowing}", "{bounded_semilattice_sup_bot, warrowing}") warrowing
begin

definition widen_dg_state ::
  "('a, 'b) dg_state ⇒ ('a, 'b) dg_state ⇒ ('a, 'b) dg_state"
where
  "widen_dg_state a b = DG (dg_local a ∇ dg_local b) (dg_global a ∇ dg_global b)"

definition narrow_dg_state ::
  "('a, 'b) dg_state ⇒ ('a, 'b) dg_state ⇒ ('a, 'b) dg_state"
where
  "narrow_dg_state a b = DG (dg_local a △ (dg_local b)) (dg_global a △ (dg_global b))"

instance
  by standard
    (simp_all add: less_eq_dg_state_def widen_dg_state_def narrow_dg_state_def
      narrowing_class.narrow_ge narrowing_class.narrow_le
      widening_class.widen_ge1 widening_class.widen_ge2)

end

end
```

6.3 DG_Constraint_Programs

```
theory DG_Constraint_Programs
  imports DG_State "Voblint_Solver.Strategy_Program_Fold"
begin

context
begin
```

This theory states its observations at the program level — `traverse_program`, `sides_of_program`, `dep_program` — reaching for `sp_compile` only where a statement must meet the vendored solver's own tree vocabulary, and does not unfold it globally: the named observation equations in `Voblint_Solver.Strategy_Tree_Program` are stated against `sp_compile` itself, and proofs meet those rather than the compiler's representation.

What a right-hand side over the packed D/G carrier is

What a generator hands the solver, and how several of those combine. An edge former takes its source as an address in the solver's own valuation space and its published slot as an explicit key, and produces a strategy program the solver can run: the program reads what it needs, publishes what it must, and answers with a `dg_state`. A generator with several such programs for one unknown — several predecessors, several call sites — folds them here, and the fold's own answer, publications and dependencies are each the join, respectively union, over the members.

`Voblint_Framework.DG_State` supplies the value type and its lattice. Nothing here inspects a domain, a variable, a global classifier, or a CFG: an address and a key are all a former sees, which is why the same former reads a contribution-only unknown and an equation-driven one.

Low-level edge formers over a solution address

The two formers in this subsection take a step in the direct shape `'dl ⇒ 'dg ⇒ 'dg × 'dl`, which no analysis writes: a specification supplies manager-native transfers, and `DG_Spec` compiles those. They survive because the keyed-solver update-rule regression needs a step it can vary directly, without a `dg_spec` in the way, and that regression is their only consumer. Read them as the minimal thing a right-hand side can be, not as the shape production equations take.

An edge or combine former takes its source as an *address* in the solver's own valuation space `'x + 'k` and its published slot as an explicit key, rather than fixing the source to a local unknown and the slot to the `unit` key – what lets a keyed generator target an arbitrary activation-indexed unknown directly, with no separate address-rewriting pass.

A program point whose value is carried by a contribution-only unknown – one with no equation of its own, so that its value is exactly the join of what was published to it – is then read by exactly the same former as one carried by an equation-driven unknown, since `QueryL` and `QueryG` project the same valuation.

`sp_read_at` itself (`Voblint_Solver.Strategy_Tree_Program`) is a generic solver-address dispatcher with no notion of `D`/`G`; here it is instantiated at `'d = ('dl, 'dg) dg_state`, so the value it returns is already one of this file's packed `DG _ bot` / `DG bot _` slots, not a raw `D`. Reading through an address is therefore an operation on already-packed DG state, not a general "query something of unknown kind" primitive.

```
definition dg_edge_program_at ::
  "('dl::bounded_semilattice_sup_bot ⇒ 'dg::bounded_semilattice_sup_bot ⇒ 'dg × 'dl)
   ⇒ 'x + 'k ⇒ 'k
   ⇒ ('x, 'k, ('dl, 'dg) dg_state, ('dl, 'dg) dg_state) strategy_program"
where
  "dg_edge_program_at step src gk =
     (do {
       d ← sp_read_at src;
       g ← sp_read_global gk;
       _ ← sp_publish gk (DG bot (fst (step (dg_local d) (dg_global g))));
       sp_return (DG (snd (step (dg_local d) (dg_global g))) bot)
     })"

definition dg_edge_contribution_program_at ::
  "('dl::bounded_semilattice_sup_bot ⇒ 'dg::bounded_semilattice_sup_bot ⇒ 'dg × 'dl)
   ⇒ 'x + 'k ⇒ 'k
   ⇒ ('x, 'k, ('dl, 'dg) dg_state, ('dl, 'dg) dg_state) strategy_program"
where
  "dg_edge_contribution_program_at step src gk =
     (do {
       d ← sp_read_at src;
       g ← sp_read_global gk;
```

```
      sp_return (DG (snd (step (dg_local d) (dg_global g))) (fst (step (dg_local d) (dg_global g))))
    })"

lemma dep_dg_edge_program_at:
  "dep_program tau (dg_edge_program_at step src gk) = {src, Inr gk}"
  by (cases src) (simp_all add: dg_edge_program_at_def sp_compile_def)

lemma sp_wf_dg_edge_program_at [intro, simp]: "sp_wf (dg_edge_program_at step src gk)"
  unfolding dg_edge_program_at_def by (intro sp_wf_bind) simp_all

lemma sp_wf_dg_edge_contribution_program_at [intro, simp]:
  "sp_wf (dg_edge_contribution_program_at step src gk)"
  unfolding dg_edge_contribution_program_at_def by (intro sp_wf_bind) simp_all
```

The framework boundary, as theorems

Two independent opaque domains. An analysis provides `step : D ⇒ G ⇒ G × D` — transfer plus publication; the framework reads the source address's `D` and the published slot's `G`, and transports the step's `Side : G` and `Answer : D`. Slot packing (`DG _ bot` / `DG bot _`) encodes the two-typed unknown space into the solver's single value type; the framework never looks inside `D` or `G`. The two theorems say so for *every* step and every address: an answer carries no `G`, a publication no `D`.

Why a Side-free analogue exists

`dg_edge_program_at` publishes its `G` contribution as soon as it is evaluated (`Side`). When a generator folds several such programs into one equation's right-hand side — several intra predecessors, or several return sites — each one's `Side` is published at a different moment within the same evaluation, and the vendored solver's warrowing/APINIS update rule gates convergence per *origin*: repeated writes to one key from one origin can destabilize the equation's own dependency and never converge.

`dg_edge_contribution_program_at` is the Side-free analogue. It answers the *unsplit* `(G, D)` result as one `dg_state` and publishes nothing, so a caller folds several of them with `fold_rhs_program` — whose own `Side` is empty — and splits the aggregate once, after every contribution has been read. The two lemmas below are what makes that substitution legitimate: each half of the contribution answer is exactly the observation the Side form makes.

Folding a list of programs into one D/G value

One unknown can have several right-hand sides — one per intra predecessor, one per call site — and the solver takes one tree per equation. `side_rhs_fold_dg` is that reduction: it runs the programs in order, joins their answers into a running local accumulator, and answers `DG acc bot`, letting each program's own publications pass through untouched. `side_acc_dg` is the same accumulator read declaratively off a valuation, and it is what the characterization lemmas below state the fold's three observations — answer, publications, dependencies — in terms of.

Both this and `fold_rhs_program` are specializations of one fold, `fold_rhs_program_projected`, and the choice of specialization is the whole direct/buffered split. The identity specialization accumulates complete answers, so a contribution's global half joins into the node's answer; this one accumulates only `dg_local` and embeds the result as `DG d bot`, so a global half never reaches the answer and each program publishes its own `Side` as it runs. The buffered generator wants the first — it gathers every global contribution into the answer precisely so it can publish once per node — and the direct generator wants the second. Replacing either fold by the other would exchange the two generators, not simplify them.

The narrow definitions here exist to keep the D/G vocabulary the generators and the routed soundness development actually reason in, rather than spelling out `fold_rhs_program_projected dg_local (λd. DG d bot)` in every statement. Their algebra is inherited: each observation and monotonicity fact below is a specialization of the generic one.

```
definition side_rhs_fold_dg ::
  "'d::bounded_semilattice_sup_bot
   ⇒ ('x, 'g, ('d, 'h::bounded_semilattice_sup_bot) dg_state,
        ('d, 'h) dg_state) strategy_program list
   ⇒ ('x, 'g, ('d, 'h) dg_state, ('d, 'h) dg_state) strategy_program"
where
  "side_rhs_fold_dg acc ps = fold_rhs_program_projected dg_local (λd. DG d bot) acc ps"

lemma side_rhs_fold_dg_simps [simp, code]:
  "side_rhs_fold_dg acc [] = sp_return (DG acc bot)"
  "side_rhs_fold_dg acc (p # ps) =
     do {
```

```
      res ← p;
      side_rhs_fold_dg (acc ⊔ dg_local res) ps
    }"
  by (simp_all add: side_rhs_fold_dg_def)

lemma sp_wf_side_rhs_fold_dg [intro]:
  "∀p ∈ set ps. sp_wf p ⟹ sp_wf (side_rhs_fold_dg acc ps)"
  unfolding side_rhs_fold_dg_def by (rule sp_wf_fold_rhs_program_projected)

definition side_acc_dg ::
  "'d::bounded_semilattice_sup_bot
   ⇒ ('x + 'g ⇒ ('d, 'h::bounded_semilattice_sup_bot) dg_state)
   ⇒ ('x, 'g, ('d, 'h) dg_state, ('d, 'h) dg_state) strategy_program list ⇒ 'd"
where
  "side_acc_dg acc τ ps = fold_acc_projected dg_local acc τ ps"

lemma side_acc_dg_simps [simp, code]:
  "side_acc_dg acc τ [] = acc"
  "side_acc_dg acc τ (p # ps) =
     side_acc_dg (acc ⊔ dg_local (traverse_program p τ)) τ ps"
  by (simp_all add: side_acc_dg_def)

lemma traverse_side_rhs_fold_dg:
  assumes wf: "∀p ∈ set ps. sp_wf p"
  shows "traverse_program (side_rhs_fold_dg acc ps) τ = DG (side_acc_dg acc τ ps) bot"
  unfolding side_rhs_fold_dg_def side_acc_dg_def
  by (rule traverse_fold_rhs_program_projected_char[OF wf])
```

`side_rhs_fold_dg`'s accumulator only ever reaches the terminal `Answer`, so its dependency set is independent of the accumulator: the recursion skeleton over `ps` never branches on `acc`.

```
lemma dep_program_side_rhs_fold_dg_char:
  assumes wf: "∀p ∈ set ps. sp_wf p"
  shows "dep_program sigma (side_rhs_fold_dg acc ps) = (⋃p∈set ps. dep_program sigma p)"
  unfolding side_rhs_fold_dg_def
  by (rule dep_program_fold_rhs_program_projected_char[OF wf])

lemma dep_program_side_rhs_fold_dg_acc_indep:
  assumes wf: "∀p ∈ set ps. sp_wf p"
  shows "dep_program sigma (side_rhs_fold_dg acc1 ps)
     = dep_program sigma (side_rhs_fold_dg acc2 ps)"
  by (simp add: dep_program_side_rhs_fold_dg_char[OF wf])
```

The two order-preservation facts the generic fold asks for before it will carry its contributions' environment-monotonicity: the D/G instance's projection and its embedding.

```
lemma mono_dg_local: "mono (dg_local :: ('d::order, 'h::order) dg_state ⇒ 'd)"
  by (rule monoI) (simp add: less_eq_dg_state_def)

lemma mono_DG_bot:
  "mono (λd. DG d bot :: ('d::order, 'h::order_bot) dg_state)"
  by (rule monoI) simp
```

Environment-monotonicity of the fold, given every folded program is itself environment-monotone. The content is `traverse_fold_rhs_program_projected_mono`'s; this instance only supplies the two monotonicity facts above.

```
lemma side_rhs_fold_dg_mono:
  assumes wf: "∀p ∈ set ps. sp_wf p"
    and prog_mono: "∀p ∈ set ps. ∀s1 s2. s1 ≤ s2 ⟶ traverse_program p s1 ≤ traverse_program p s2"
  shows "⋀acc s1 s2. s1 ≤ s2 ⟹
           traverse_program (side_rhs_fold_dg acc ps) s1
             ≤ traverse_program (side_rhs_fold_dg acc ps) s2"
  using prog_mono
  unfolding side_rhs_fold_dg_def
  by (auto intro!: traverse_fold_rhs_program_projected_mono[OF wf mono_dg_local mono_DG_bot])
```

The vendored solver's own `mono_deps` precondition only needs the weaker, order-conditional guarantee `mono_tree_deps`, not the full equality `env_indep_deps`; the generic fold already carries it, so this instance is one line.

```
lemma side_rhs_fold_dg_mono_tree_deps:
  assumes wf: "∀p ∈ set ps. sp_wf p"
    and prog_mono: "⋀p. p ∈ set ps ⟹ mono_tree_deps (sp_compile p)"
  shows "mono_tree_deps (sp_compile (side_rhs_fold_dg acc ps))"
  unfolding side_rhs_fold_dg_def
  by (rule mono_tree_deps_fold_rhs_program_projected[OF wf prog_mono])
```

The fold's contributions are carried only by the per-contribution Side nodes; the accumulator flows into the final `Answer` (whose own sides are `bot`), so the side map is acc-independent — the same fact `∀p∈set ?ps. sp_wf p ⟹`

`dep_program ?sigma (side_rhs_fold_dg ?acc1.0 ?ps) = dep_program ?sigma (side_rhs_fold_dg ?acc2.0 ?ps)` established for dependencies, mirrored here for sides. This is what lets `⟦∀p∈set ?ps. sp_wf p; ∀p∈set ?ps. ∀s1 s2. s1 ≤ s2 ⟶ traverse_program p s1 ≤ traverse_program p s2; ?s1.0 ≤ ?s2.0⟧ ⟹ traverse_program (side_rhs_fold ?acc ?ps) ?s1.0 ≤ traverse_program (side_rhs_fold_dg ?acc ?ps) ?s2.0`'s proof strategy repeat for `sides_of_rhs` without also needing `traverse_rhs`-monotonicity as a hypothesis.

```
lemma sides_of_program_side_rhs_fold_dg_acc_indep:
  assumes wf: "∀p ∈ set ps. sp_wf p"
  shows "sides_of_program (side_rhs_fold_dg acc1 ps) sigma
     = sides_of_program (side_rhs_fold_dg acc2 ps) sigma"
  unfolding side_rhs_fold_dg_def
  by (rule sides_of_program_fold_rhs_program_projected_acc_indep[OF wf])
```

The declarative twin of `side_acc_dg`: `side_rhs_fold_dg`'s contribution at any one key is a plain fold over each element's own `sides_of_rhs`, seeded at `bot` rather than the running local accumulator – `∀p∈set ?ps. sp_wf p ⟹ sides_of_program (side_rhs_fold_dg ?acc1.0 ?ps) ?sigma = sides_of_program (side_rhs_fold_dg ?acc2.0 ?ps) ?sigma` already shows the accumulator never affects a side read, so unfolding `sp_lift_tree`'s side equation and re-seeding at `bot` after each step is exact, not merely a bound.

```
lemma sides_of_program_side_rhs_fold_dg_char:
  fixes ps :: "('x, 'k, ('d::bounded_semilattice_sup_bot, 'h::bounded_semilattice_sup_bot) dg_state,
                ('d, 'h) dg_state) strategy_program list"
  assumes wf: "∀p ∈ set ps. sp_wf p"
  shows "sides_of_program (side_rhs_fold_dg acc ps) τ z
         = (⨆p←ps. sides_of_program p τ z)"
  unfolding side_rhs_fold_dg_def
  by (rule sides_of_program_fold_rhs_program_projected_char[OF wf])

lemma side_acc_dg_as_foldr:
  "side_acc_dg acc τ ps = acc ⊔ (⨆p←ps. dg_local (traverse_program p τ))"
  by (simp add: side_acc_dg_def fold_acc_projected_as_foldr)
```

The same equation with the accumulator left as the fold's seed, which is how a proof that already has a right fold in hand meets it.

```
lemma side_acc_dg_as_foldr_seeded:
  "side_acc_dg acc τ ps = foldr (λp a. dg_local (traverse_program p τ) ⊔ a) ps acc"
  by (simp only: side_acc_dg_as_foldr foldr_join_seed_out)

lemma side_rhs_fold_dg_sides_mono:
  assumes wf: "∀p ∈ set ps. sp_wf p"
    and prog_sides_mono:
      "∀p ∈ set ps. ∀s1 s2. s1 ≤ s2 ⟶ sides_of_program p s1 ≤ sides_of_program p s2"
  shows "⋀acc s1 s2. s1 ≤ s2 ⟹
           sides_of_program (side_rhs_fold_dg acc ps) s1
             ≤ sides_of_program (side_rhs_fold_dg acc ps) s2"
  using prog_sides_mono
  unfolding side_rhs_fold_dg_def
  by (auto intro!: sides_of_program_fold_rhs_program_projected_mono[OF wf])
```

Pushing a projection through a join-fold

A `dg_state` join is componentwise, so either projection commutes with a join-fold. These are what let a proof that has folded whole states read off one component afterwards, rather than carrying two folds along.

```
lemma dg_local_foldr_generic:
  "dg_local (foldr (λt acc'. h t ⊔ acc') L
      (acc::('d::bounded_semilattice_sup_bot, 'h::bounded_semilattice_sup_bot) dg_state))
    = foldr (λt acc'. dg_local (h t) ⊔ acc') L (dg_local acc)"
  by (induction L) simp_all

lemma dg_global_foldr_generic:
  "dg_global (foldr (λt acc'. h t ⊔ acc') L
      (acc::('d::bounded_semilattice_sup_bot, 'h::bounded_semilattice_sup_bot) dg_state))
    = foldr (λt acc'. dg_global (h t) ⊔ acc') L (dg_global acc)"
  by (induction L) simp_all

lemma dg_local_join_list:
  "dg_local (⨆t←L. h t :: ('d::bounded_semilattice_sup_bot, 'h::bounded_semilattice_sup_bot) dg_state)
     = (⨆t←L. dg_local (h t))"
  by (induction L) simp_all

lemma dg_global_join_list:
  "dg_global (⨆t←L. h t :: ('d::bounded_semilattice_sup_bot, 'h::bounded_semilattice_sup_bot) dg_state)
     = (⨆t←L. dg_global (h t))"
```

```
  by (induction L) simp_all

lemma foldr_dg_global_sides_char:
  fixes L :: "('x, 'k, ('d::bounded_semilattice_sup_bot, 'h::bounded_semilattice_sup_bot) dg_state,
               ('d, 'h) dg_state) strategy_program list"
  shows "(⨆t←L. dg_global (sides_of_program t τ z))
       = dg_global (⨆t←L. sides_of_program t τ z)"
  by (simp add: dg_global_foldr_generic foldr_join_list[symmetric])

lemma foldr_sides_dg_local_bot:
  fixes L :: "('x, 'k, ('d::bounded_semilattice_sup_bot, 'h::bounded_semilattice_sup_bot) dg_state,
               ('d, 'h) dg_state) strategy_program list"
  assumes "⋀t. t ∈ set L ⟹ dg_local (sides_of_program t τ z) = bot"
  shows "dg_local (⨆t←L. sides_of_program t τ z) = bot"
  using assms
  by (induction L) simp_all

lemma DG_sup_bot_left:
  "DG (bot::'d::bounded_semilattice_sup_bot) a ⊔ DG bot b = DG bot (a ⊔ b)"
  by (simp add: sup_dg_state_def bot_dg_state_def)
```

What one folded program contributes

Lower bounds, the counterpart of the upper bounds above: every folded program's answer and contribution is below the fold's, and the accumulator only grows. A soundness proof reaches one selected contribution through these, having reached the fold itself through the generator.

All three are the same fact about a join-fold — `foldr_sup_member_le`, proved once with the rest of that algebra where the fold itself is defined — read off the two characterizations above.

```
lemma side_acc_dg_ge_acc:
  "acc ≤ side_acc_dg acc σ ts"
  by (simp add: side_acc_dg_as_foldr)

lemma dg_local_traverse_le_side_acc_dg:
  assumes "p ∈ set ps"
  shows "dg_local (traverse_program p σ) ≤ side_acc_dg acc σ ps"
  unfolding side_acc_dg_as_foldr
  using foldr_sup_member_le[where h = "λp. dg_local (traverse_program p σ)", OF assms]
  by (rule le_supI2)

lemma sides_le_side_rhs_fold_dg:
  assumes wf: "∀p ∈ set ps. sp_wf p"
    and mem: "p ∈ set ps"
  shows "sides_of_program p σ k ≤ sides_of_program (side_rhs_fold_dg acc ps) σ k"
  unfolding sides_of_program_side_rhs_fold_dg_char[OF wf]
  by (rule foldr_sup_member_le[OF mem])

end

end
```

6.4 DG_Manager

```
theory DG_Manager
  imports DG_State "Voblint_Solver.Strategy_Tree_Program" Analysis_Query
begin
```

A manager capability interface for the D/G packed carrier

`man` is a capability fragment of Goblint's own manager record: `man_local` is the current local value, `man_global`/`man_sideg` are *capabilities* a transfer runs to read or publish shared state, without naming which solver key that state lives at, and `man_ask` is the query channel, Goblint's `man.ask`. Goblint's manager carries more – an event emitter, the current node and edge, spawn and split – and none of that is modelled here, because no transfer in this development asks for it. Every `Spec` transfer takes a manager, including ones whose global component is trivial – Goblint's default `Spec` sets `G = Lattice.Unit`, `V = EmptyV`, and its transfers still take the manager and mostly return `man.local`.

`man_ask` answers a query about the stores the transfer's predecessor state describes. Like `man_global` it is a program, so a query handler may read shared state and the dependency shows up in the compiled equations. A handler may also ask in turn; the specification layer (`DG_Spec`) bounds the recursion and answers $\top$ on a cycle, as Goblint's `MCP` does.

Five type parameters run through this theory and every one built on it:

- `'x` — a local solver unknown, the name of a program point under a context.
- `'k` — a solver global key, the whole side-effect address space.
- `'v` — an analysis-visible global name, Goblint's `V`. It embeds into `'k`, and is exactly the part of that space an analysis may address.
- `'dl` — the local domain, Goblint's `D`.
- `'dg` — the analysis-global domain, Goblint's `G`.

Both capabilities take a global name rather than a key: an analysis says `man_global m v`, naming which of its globals it means, and never builds a solver key. Which key that name lives at is the manager's business — `mk_dg_man` takes the embedding `'v ⇒ 'k` and closes it into both fields. An analysis with a single global instantiates `'v = unit` and writes `man_global m ()`. The separation matters because `'k` also carries the routed activation slots, which belong to the generator and to no analysis.

A Base-style specification ignores the global channel entirely: it calls neither capability, so its compiled equations carry no `QueryG` and no `Side`. An effectful specification reaches the current routed global slot through `man_global`/`man_sideg`. Which of the two a given analysis is therefore shows up in its compiled trees, not in this interface.

`dg_read_global` and `dg_sideg` are what interpret those capabilities against the packed `('dl,'dg) dg_state` carrier; `mk_dg_man` is where the embedding `unknown_of` is closed into them, so it is the only place a name becomes a key. Either way a transfer written against `man`'s fields sees neither the packed carrier nor a key.

The manager record

```
record ('x,'k,'v,'dl,'dg) man =
  man_local :: 'dl
  man_global :: "'v ⇒ ('x,'k,('dl,'dg) dg_state,'dg) strategy_program"
  man_sideg :: "'v ⇒ 'dg ⇒ ('x,'k,('dl,'dg) dg_state,unit) strategy_program"
  man_ask :: "query ⇒ ('x,'k,('dl,'dg) dg_state,answer) strategy_program"
```

Packed-carrier primitives

A solver unknown holds a whole `dg_state`, so reading one yields both halves and only one of them is ever wanted. `dg_read_at` reads any unknown and keeps `dg_local`; `dg_read_global` reads a global key and keeps `dg_global`; `dg_sideg` publishes a global contribution, padding the local half it does not write with `bot`.

These three are where the packed carrier is taken apart and put back together. The framework's own transfer constructors use only these and `man`'s fields, and an analysis author is expected to do the same; the transfer type does not enforce it, since a transfer may build any `strategy_program` it likes. The tree formers that assemble a right-hand side do construct `DG` values directly.

```
definition dg_read_at :: "'x + 'k ⇒ ('x,'k,('dl,'dg) dg_state,'dl) strategy_program" where
  "dg_read_at src = sp_read_at src ⤜ (sp_return o dg_local)"
```

```
definition dg_read_global :: "'k ⇒ ('x,'k,('dl,'dg) dg_state,'dg) strategy_program" where
  "dg_read_global gk = sp_read_global gk ≫= (sp_return o dg_global)"

definition dg_sideg :: "'k ⇒ 'dg ⇒ ('x,'k,('dl::bot,'dg) dg_state,unit) strategy_program" where
  "dg_sideg gk gd = sp_publish gk (DG bot gd)"
```

What the primitives do, observed

Each primitive contributes one step to a compiled tree, and the three observations a solver makes of that tree — its answer, its publications, its dependencies — see exactly that step and then continue into the continuation. Proving it once here is what keeps later proofs off the encoding: without these, every fact about a compiled transfer has to case on the address and unfold `(≫=)`, `sp_return` and the read primitives to reach its own subject. They are `simp` rules in the direction that eliminates a primitive and exposes the value it read, and they terminate because the program strictly shrinks at each step.

```
lemma traverse_dg_read_at [simp]:
  "traverse_rhs (dg_read_at src K) τ = traverse_rhs (K (dg_local (τ src))) τ"
  by (cases src) (simp_all add: dg_read_at_def sp_bind_def sp_read_local_def
      sp_read_global_def sp_return_def)

lemma sides_dg_read_at [simp]:
  "sides_of_rhs (dg_read_at src K) τ = sides_of_rhs (K (dg_local (τ src))) τ"
  by (cases src) (simp_all add: dg_read_at_def sp_bind_def sp_read_local_def
      sp_read_global_def sp_return_def)

lemma dep_aux_dg_read_at [simp]:
  "dep_aux τ (dg_read_at src K) = insert src (dep_aux τ (K (dg_local (τ src))))"
  by (cases src) (simp_all add: dg_read_at_def sp_bind_def sp_read_local_def
      sp_read_global_def sp_return_def)

lemma traverse_dg_read_global [simp]:
  "traverse_rhs (dg_read_global gk K) τ = traverse_rhs (K (dg_global (τ (Inr gk)))) τ"
  by (simp add: dg_read_global_def sp_bind_def sp_read_global_def sp_return_def)

lemma sides_dg_read_global [simp]:
  "sides_of_rhs (dg_read_global gk K) τ = sides_of_rhs (K (dg_global (τ (Inr gk)))) τ"
  by (simp add: dg_read_global_def sp_bind_def sp_read_global_def sp_return_def)

lemma dep_aux_dg_read_global [simp]:
  "dep_aux τ (dg_read_global gk K)
     = insert (Inr gk) (dep_aux τ (K (dg_global (τ (Inr gk)))))"
  by (simp add: dg_read_global_def sp_bind_def sp_read_global_def sp_return_def)

lemma traverse_dg_sideg [simp]:
  "traverse_rhs (dg_sideg gk gd K) τ = traverse_rhs (K ()) τ"
  by (simp add: dg_sideg_def sp_publish_def)

lemma dep_aux_dg_sideg [simp]:
  "dep_aux τ (dg_sideg gk gd K) = dep_aux τ (K ())"
  by (simp add: dg_sideg_def sp_publish_def)
```

Publication is the one primitive an observation does not simply pass through: it raises the contribution at its own key and leaves every other key to the continuation.

```
lemma sides_dg_sideg [simp]:
  "sides_of_rhs (dg_sideg gk gd K) τ = sides_of_rhs (K ()) τ ⊔ bot(Inr gk := DG bot gd)"
  by (simp add: dg_sideg_def sp_publish_def Let_def fun_eq_iff sup_fun_def)
```

Each primitive runs its continuation once, so a transfer assembled from them does too and the fold's well-formedness side conditions discharge by `simp` wherever a concrete transfer is in hand.

```
lemma sp_wf_dg_read_at [intro, simp]: "sp_wf (dg_read_at src)"
  unfolding dg_read_at_def by (intro sp_wf_bind) (simp_all add: comp_def)

lemma sp_wf_dg_read_global [intro, simp]: "sp_wf (dg_read_global gk)"
  unfolding dg_read_global_def by (intro sp_wf_bind) (simp_all add: comp_def)

lemma sp_wf_dg_sideg [intro, simp]: "sp_wf (dg_sideg gk gd)"
  by (simp add: dg_sideg_def)
```

Constructing a manager

`mk_dg_man` closes the name-to-key embedding into both effectful fields, so a transfer built against `man` never sees a key itself. A manager-native transfer should reach shared state through the record fields rather than call

`dg_read_global`/`dg_sideg` or name a key directly, so that a future manager (instrumented, differently routed, or backed by distinct global unknowns) is a second interpretation of the same fields rather than a change to any transfer.

A built manager answers every query with $\top$, the claim that holds of every store. Whoever runs a specification installs a real query channel on top: the equation generator installs the specification's own query (`dg_spec_edge_program`), and a product of specifications installs the meet of its components' answers.

```
definition mk_dg_man :: "'dl::bot ⇒ ('v ⇒ 'k) ⇒ ('x,'k,'v,'dl,'dg) man" where
  "mk_dg_man d unknown_of = (|
     man_local = d,
     man_global = (λv. dg_read_global (unknown_of v)),
     man_sideg = (λv. dg_sideg (unknown_of v)),
     man_ask = (λ_. sp_return ⊤) |)"
```

What a built manager answers, and the one way it is rebuilt: a second transfer stage runs from the point the first reached, and since only the local value changes, that update is again a manager built at the new value. These are the rules the proofs need; `mk_dg_man` itself stays folded, so a manager travels through a goal as one term instead of a three-field record literal. Keeping `mk_dg_man` folded also preserves matching: an assumption about a built manager would stop matching if the goal decayed into a record literal.

The two capability rules are stated unapplied. Applying one is the same rule plus a beta step, so the unapplied form subsumes the applied one and also rewrites the occurrences that carry no argument — a manager passed whole to an entry transfer, for instance.

```
lemma mk_dg_man_simps [simp]:
  "man_local (mk_dg_man d unknown_of) = d"
  "man_global (mk_dg_man d unknown_of) = (λv. dg_read_global (unknown_of v))"
  "man_sideg (mk_dg_man d unknown_of) = (λv. dg_sideg (unknown_of v))"
  "man_ask (mk_dg_man d unknown_of) = (λ_. sp_return ⊤)"
  by (simp_all add: mk_dg_man_def)

lemma mk_dg_man_local_update [simp]:
  "mk_dg_man d unknown_of(|man_local := e|) = mk_dg_man e unknown_of"
  "mk_dg_man d unknown_of(|man_ask := A, man_local := e|) = mk_dg_man e unknown_of(|man_ask := A|)"
  by (simp_all add: mk_dg_man_def)

type_synonym ('x,'k,'v,'dl,'dg) man_transfer =
  "('x,'k,'v,'dl,'dg) man ⇒ ('x,'k,('dl,'dg) dg_state,'dl) strategy_program"
```

A query handler takes the same manager and answers the query in the answer lattice, Goblint's `Spec.query man q`.

```
type_synonym ('x,'k,'v,'dl,'dg) man_query =
  "('x,'k,'v,'dl,'dg) man ⇒ query ⇒ ('x,'k,('dl,'dg) dg_state,answer) strategy_program"
```

A combine-shaped transfer takes the same manager plus the callee-exit value as an ordinary trailing argument, the way Goblint's `combine_env`/`combine_assign` take the same `man` plus a `D.t`.

```
type_synonym ('x,'k,'v,'dl,'dg) man_combine_transfer =
  "('x,'k,'v,'dl,'dg) man ⇒ 'dl ⇒ ('x,'k,('dl,'dg) dg_state,'dl) strategy_program"
```

A call answers a list of alternatives, and each alternative is a pair: the value the caller resumes with once the callee returns, and the value the callee starts from. This is Goblint's `Spec.enter`, whose result is a list of such pairs. The two halves must be produced by one run of the transfer, because an alternative's continuation is only meaningful against the callee entry it was computed with; a program answering the pairs keeps them together and leaves the caller free to consume them however the equation shape needs.

The result is not a `'dl`, so an entry transfer is not a `man_transfer`: it cannot be the answer of an equation on its own, and whatever consumes it must reduce the list to one local value first.

```
type_synonym 'dl enter_result = "'dl × 'dl"

type_synonym ('x,'k,'v,'dl,'dg) man_enter_transfer =
  "('x,'k,'v,'dl,'dg) man
   ⇒ ('x,'k,('dl,'dg) dg_state,'dl enter_result list) strategy_program"
```

A second transfer stage runs from the point the first reached by updating `man_local` in place — `m(|man_local := d|)` — leaving both capabilities as they were. Nothing is extracted out of the program and re-wrapped, and the record's own update equations are all the later proofs need about it.

```
end
```

6.5 DG_Spec

```
theory DG_Spec
  imports DG_Manager "Voblint_CFG.CFG_Def"
begin
```

What an analysis supplies: manager-native transfers

An analysis supplies one manager-native transfer per edge action and the pieces of the call/return protocol. A transfer takes a `man` and returns a `strategy_program`: global reads are `man_global` calls, global publications are `man_sideg` calls, and a transfer that makes neither compiles to a tree with no `QueryG` and no `Side` at all. This replaces the state-threading shape `'dl ⇒ 'dg ⇒ 'dg × 'dl`, which forced every transfer to republish the global it read even where the global channel is provably inert.

What a program answers with depends on the field. An edge transfer and a combine stage answer the successor local value, so their compiled tree is an equation's right-hand side outright. Entry answers a list of alternatives instead, which no equation carries: the routed call program consumes that list, running one activation and return combine per alternative and joining what they contribute.

There is one interface here, not a hierarchy of them. A specification is *local-only* when every field is a pure function of the values it is handed — `local_transfer` for the edge fields, `local_enter_transfer` for entry — and *effectful* when some field reads or publishes through `man_global`/`man_sideg`. Those words describe what a given specification's compiled trees do, and are read off those trees – they are not a classification the framework branches on.

An `analysis_event` is an analyzer-visible occurrence distinct from an ordinary control-flow transfer: a domain may observe it, but it must not by itself refine execution. The channel mirrors Goblint's separation of its ordinary `Spec` transfer methods (`assign`/`branch`/`skip`/...) from `Spec.event`. The check itself does not: Goblint handles `__goblint_check` in `special`, as the library description `Assert {check = true; refine = false}`, and its `Events.Assert` is emitted only by the `abortUnless` analysis and refines the state in Base (goblint/analyzer `5320a6b7`: `libraryFunctions.ml`, `assert.ml`, `base.ml`). Voblint's sole current event is a check's condition, which never refines; the vocabulary is deliberately left open rather than pre-populated, so that a future VIMP source construct with no current counterpart (e.g. a diagnostic-only annotation) adds a constructor here instead of a new specification field. A construct that narrows feasible execution, such as `assume`, is not a candidate: it belongs on `dgs_branch` or on a new refining field, not here.

It is declared here, beside the `dgs_event` field it types, because the specification record and the edge-action dispatcher that feeds it are the only things that ever case on it.

```
datatype analysis_event =
  Check_Event check_label exp

record ('x,'k,'v,'dl,'dg) dg_spec =
  dgs_skip          :: "('x,'k,'v,'dl,'dg) man_transfer" ("skip♯")
  dgs_assign        :: "vname ⇒ exp ⇒ ('x,'k,'v,'dl,'dg) man_transfer" ("assign♯")
  dgs_special       :: "special_call ⇒ vname ⇒ ('x,'k,'v,'dl,'dg) man_transfer" ("special♯")
  dgs_branch        :: "exp ⇒ bool ⇒ ('x,'k,'v,'dl,'dg) man_transfer" ("branch♯")
  dgs_body          :: "pname ⇒ ('x,'k,'v,'dl,'dg) man_transfer" ("body♯")
  dgs_return        :: "exp option ⇒ pname ⇒ ('x,'k,'v,'dl,'dg) man_transfer" ("return♯")
  dgs_enter         :: "call_info ⇒ ('x,'k,'v,'dl,'dg) man_enter_transfer" ("enter♯")
  dgs_event         :: "analysis_event ⇒ ('x,'k,'v,'dl,'dg) man_transfer" ("event♯")
  dgs_combine_env   :: "call_info ⇒ ('x,'k,'v,'dl,'dg) man_combine_transfer" ("combine'_env♯")
  dgs_combine_assign :: "call_info ⇒ ('x,'k,'v,'dl,'dg) man_combine_transfer" ("combine'_assign♯")
  dgs_query         :: "('x,'k,'v,'dl,'dg) man_query"
```

`dgs_query` is Goblint's `Spec.query`: it answers a query from the manager it is given, and may read globals or ask further queries through it. What an answer claims is fixed by `eval_holds`; answering ⊤ claims nothing.

Every field carries the Goblint `Spec` method name it answers to, marked ♯. A specification therefore reads in the analyzer's vocabulary, and the marker keeps the abstract operation apart from the concrete one it over-approximates: `combine_env` merges two stores, whereas `combine_env`♯ is the field a domain fills to describe that merge. The whole-state combine wears the same marker as `combine`♯, so it means one thing on both sides of the transfer boundary.

Two of the names contain an underscore, which a mixfix template would otherwise read as an argument slot, so they escape it as `'_`. The selector names stay available for record construction and update, neither of which takes notation.

Procedure-return combine is split the same way Goblint's `Spec` splits it: an environment merge followed by a return-value assign, both taking the same manager plus the callee exit. The composed form sequences

them monadically – `combine_assign` runs from the point `combine_env` reached, against the same manager with `man_local` updated to that point, so any effects `combine_env` emitted stay in the program and nothing is extracted into a pure pair in between.

```
definition dg_spec_combine_transfer ::
  "('x,'k,'v,'dl,'dg) dg_spec ⇒ call_info ⇒ ('x,'k,'v,'dl,'dg) man_combine_transfer"
where
  "dg_spec_combine_transfer S ci m exit =
     do {
       d_env ← combine_env♯ S ci m exit;
       combine_assign♯ S ci (m(|man_local := d_env|)) exit
     }"
```

`EA_Check` routes through `event♯` rather than `skip♯`: a check is an analysis event (on the channel that mirrors Goblint's `Spec.event`; Goblint's own check goes through `special`, see `analysis_event` above), and conflating it with skip would make a future domain's non-identity skip silently change what a check edge does. A concrete `dg_spec` therefore supplies its own notion of a check event directly, the same way it already supplies `body♯`/`return♯`.

```
fun dg_spec_step ::
  "('x,'k,'v,'dl,'dg,'z) dg_spec_scheme ⇒ edge_action ⇒ ('x,'k,'v,'dl,'dg) man_transfer"
where
  "dg_spec_step S EA_Nop            = skip♯ S"
| "dg_spec_step S (EA_Assign x e)   = assign♯ S x e"
| "dg_spec_step S (EA_Special sc x) = special♯ S sc x"
| "dg_spec_step S (EA_Assume b)     = branch♯ S b True"
| "dg_spec_step S (EA_AssumeNot b)  = branch♯ S b False"
| "dg_spec_step S (EA_Body p)       = body♯ S p"
| "dg_spec_step S (EA_Ret e p)      = return♯ S e p"
| "dg_spec_step S (EA_Check l cnd)  = event♯ S (Check_Event l cnd)"
```

Compiling a specification to right-hand sides

An edge equation reads the source unknown, builds the manager around it with the routed key closed in, and runs the specification's transfer. Every generator's edge right-hand side is a `dg_spec_edge_program`.

The driver underneath: read the source unknown, close the key into a fresh manager around what came back, and run the transfer once. The transfer itself never sees the key, only whatever the manager's fields already close over.

```
definition transfer_program_at ::
  "('x,'k,'v,'dl,'dg) man_transfer ⇒ 'x + 'k ⇒ ('v ⇒ 'k)
   ⇒ ('x,'k,('dl::bot,'dg) dg_state,'dl) strategy_program"
where
  "transfer_program_at transfer src unknown_of =
     do {
       d ← dg_read_at src;
       transfer (mk_dg_man d unknown_of)
     }"

definition transfer_program ::
  "('x,'k,'v,'dl::bot,'dg::bot) man_transfer ⇒ 'x + 'k ⇒ ('v ⇒ 'k)
   ⇒ ('x,'k,('dl,'dg) dg_state,('dl,'dg) dg_state) strategy_program"
where
  "transfer_program T src unknown_of = sp_map (λd. DG d bot) (transfer_program_at T src unknown_of)"
```

Before the specification's edge transfer runs, its query channel is set to the specification's own answers about the state the edge starts from, as Goblint's `MCP` builds `outer_man` around the predecessor. Every transfer on the edge sees the same channel, and nothing it computes changes it.

A handler may ask in turn, through a manager whose channel is the same construction one level down. As in Goblint's `MCP.query'`, a query already being asked answers $\top$ instead of recursing, the result lattice's top on a cycle. The depth bound has no counterpart in Goblint: queries range over an infinite type, so cycle detection alone does not make the recursion total. Exhausting it aborts the generated program, so a bound that is too small shows up as a failed run and never as a silent loss of precision. Logically `Code.abort` is its fallback, here $\top$, which holds of every store, so the bound plays no part in soundness.

The channel depends on the specification only through its handler, so a specification derived by overriding other fields keeps the same channel.

```
definition query_depth :: nat where
  "query_depth = 1024"

fun ask_with ::
  "('x,'k,'v,'dl,'dg) man_query ⇒ nat ⇒ query set ⇒ ('x,'k,'v,'dl,'dg) man
```

```
   ⇒ query ⇒ ('x,'k,('dl,'dg) dg_state,answer) strategy_program"
where
  "ask_with Q 0 asked m q =
     Code.abort (STR ''query recursion exceeded query_depth'') (λ_. sp_return ⊤)"
| "ask_with Q (Suc n) asked m q =
     (if q ∈ asked then sp_return ⊤
      else Q (m⦇man_ask := ask_with Q n (insert q asked) m⦈) q)"

definition outer_man ::
  "('x,'k,'v,'dl,'dg) man_query ⇒ ('x,'k,'v,'dl,'dg) man ⇒ ('x,'k,'v,'dl,'dg) man"
where
  "outer_man Q m = m⦇man_ask := ask_with Q query_depth {} m⦈"

lemma outer_man_simps [simp]:
  "man_local (outer_man Q m) = man_local m"
  "man_global (outer_man Q m) = man_global m"
  "man_sideg (outer_man Q m) = man_sideg m"
  "man_ask (outer_man Q m) = ask_with Q query_depth {} m"
  by (simp_all add: outer_man_def)

definition dg_spec_edge_program ::
  "('x,'k,'v,'dl::bot,'dg::bot) dg_spec ⇒ edge_action ⇒ 'x + 'k ⇒ ('v ⇒ 'k)
   ⇒ ('x,'k,('dl,'dg) dg_state,('dl,'dg) dg_state) strategy_program"
where
  "dg_spec_edge_program S a src unknown_of =
     transfer_program (λm. dg_spec_step S a (outer_man (dgs_query S) m)) src unknown_of"
```

What compilation costs an observation: exactly the source read, and then the transfer run at the value that read produced. Everything a proof about a compiled edge needs from the encoding is here, so nothing downstream has to case on the address or unfold the program monad again.

Left untagged. Rewriting a compiled tree back into a manager application grows the term, and the direction is only wanted where a proof is about to reason about the transfer itself; the primitive laws in `Voblint_Framework.DG_Manager` are the ones that should fire everywhere.

```
lemma traverse_transfer_program:
  "traverse_program (transfer_program T src unknown_of) τ
     = traverse_rhs (sp_compile_with (λd. DG d bot) (T (mk_dg_man (dg_local (τ src)) unknown_of))) τ"
  by (simp add: transfer_program_def transfer_program_at_def sp_compile_with_def sp_bind_def)

lemma sides_transfer_program:
  "sides_of_program (transfer_program T src unknown_of) τ
     = sides_of_rhs (sp_compile_with (λd. DG d bot) (T (mk_dg_man (dg_local (τ src)) unknown_of))) τ"
  by (simp add: transfer_program_def transfer_program_at_def sp_compile_with_def sp_bind_def)

lemma dep_transfer_program:
  "dep_program τ (transfer_program T src unknown_of)
     = insert src
         (dep_aux τ (sp_compile_with (λd. DG d bot) (T (mk_dg_man (dg_local (τ src)) unknown_of))))"
  by (simp add: transfer_program_def transfer_program_at_def sp_compile_with_def sp_bind_def)
```

Specifications whose transfers run their continuation once

A `man_transfer` is a function from a continuation to a tree, so a specification could in principle write one that drops or duplicates it. `sp_wf` rules that out, and the generator's fold needs it: what a contribution publishes is only well posed for a transfer that runs its continuation once.

`dg_spec_wf` asks it of the four ways a specification is run — an edge step, a query, the entry alternatives, and the two-stage combine — and only at the managers that actually occur, the ones `mk_dg_man` builds. An edge step and a query handler run under a query channel installed around that manager, so they are asked for every channel that is itself well formed; a wrapper that reruns a step at a rebuilt manager needs exactly that generality. That restriction is not a weakening: a transfer never meets any other manager, and an arbitrary one has opaque capability fields about which nothing could be said. A specification written with the manager's own primitives satisfies all four by `simp`, since every primitive and every combinator carries a closure lemma.

```
definition dg_spec_wf :: "('x,'k,'v,'dl::bot,'dg) dg_spec ⇒ bool" where
  "dg_spec_wf S ⟷
     (∀a d unknown_of ch. (∀q. sp_wf (ch q))
          ⟶ sp_wf (dg_spec_step S a ((mk_dg_man d unknown_of)⦇man_ask := ch⦈)))
     ∧ (∀d unknown_of ch q. (∀q'. sp_wf (ch q'))
          ⟶ sp_wf (dgs_query S ((mk_dg_man d unknown_of)⦇man_ask := ch⦈) q))
     ∧ (∀ci d unknown_of. sp_wf (enter♯ S ci (mk_dg_man d unknown_of)))
     ∧ (∀ci d unknown_of ex. sp_wf (dg_spec_combine_transfer S ci (mk_dg_man d unknown_of) ex))"

lemma dg_spec_wf_step_ask:
```

```
  "dg_spec_wf S ⟹ (⋀q. sp_wf (ch q))
   ⟹ sp_wf (dg_spec_step S a ((mk_dg_man d unknown_of)⦇man_ask := ch⦈)))"
  by (simp add: dg_spec_wf_def)

lemma dg_spec_wf_query:
  "dg_spec_wf S ⟹ (⋀q. sp_wf (ch q))
   ⟹ sp_wf (dgs_query S ((mk_dg_man d unknown_of)⦇man_ask := ch⦈) q)"
  by (simp add: dg_spec_wf_def)

lemma sp_wf_ask_with:
  assumes "dg_spec_wf S"
  shows "sp_wf (ask_with (dgs_query S) n asked (mk_dg_man d unknown_of) q)"
proof (induction n arbitrary: asked q)
  case (Suc n)
  then show ?case
    using assms by (simp add: dg_spec_wf_def)
qed simp

lemma dg_spec_wf_step:
  "dg_spec_wf S ⟹ sp_wf (dg_spec_step S a (outer_man (dgs_query S) (mk_dg_man d unknown_of)))"
  unfolding outer_man_def by (intro dg_spec_wf_step_ask sp_wf_ask_with)

lemma dg_spec_wf_enter:
  "dg_spec_wf S ⟹ sp_wf (enter♯ S ci (mk_dg_man d unknown_of))"
  by (simp add: dg_spec_wf_def)

lemma dg_spec_wf_combine:
  "dg_spec_wf S ⟹ sp_wf (dg_spec_combine_transfer S ci (mk_dg_man d unknown_of) ex)"
  by (simp add: dg_spec_wf_def)

lemma sp_wf_transfer_program [intro]:
  assumes "⋀d. sp_wf (T (mk_dg_man d unknown_of))"
  shows "sp_wf (transfer_program T src unknown_of)"
  unfolding transfer_program_def transfer_program_at_def
  by (intro sp_wf_map sp_wf_bind sp_wf_dg_read_at assms)

lemma sp_wf_dg_spec_edge_program [intro]:
  "dg_spec_wf S ⟹ sp_wf (dg_spec_edge_program S a src unknown_of)"
  unfolding dg_spec_edge_program_def
  by (intro sp_wf_transfer_program) (rule dg_spec_wf_step)
```

Writing a transfer

An analysis author writes manager-native transfers and nothing else. The framework turns them into solver right-hand sides; `QueryL`, `QueryG` and `Side` belong to that step, and a specification field that names one has reached past the interface into the solver's. That is a discipline, not something the types rule out: a transfer returns a `strategy_program` and could build any of them by hand. Three recipes cover every field of a `dg_spec` without doing so:

- A transfer that only transforms the local value is `local_transfer f`, and its compiled tree is one read and an answer: no `QueryG`, no `Side`.
- A transfer that reads or publishes shared state runs the capabilities in sequence and answers the new local value — `do {g <- man_global m v; _ <- man_sideg m v g'; sp_return d'}` — which is the same shape as the Goblint method that reads `man.global`, calls `man.sideg`, and returns a `D.t`.
- An entry transfer answers the list of alternatives instead of one value: `sp_return [(continuation, entry), ...]`, possibly after the same capability calls. It never folds that list; the routed call program does.

Local-only transfers

The local-only shape: a pure transformation of the local value, touching no global at all. Its compiled edge tree is one source read followed directly by the answer – no `QueryG`, no `Side` – so its side contribution is `bot` at every key and its dependency set is exactly the source unknown. The global channel simply does not appear in these compile-down facts, which is what makes a local-only specification's soundness argument collapse to plain inclusions on pure functions.

```
definition local_transfer :: "('dl ⇒ 'dl) ⇒ ('x,'k,'v,'dl,'dg) man_transfer" where
  "local_transfer f m = sp_return (f (man_local m))"

lemma local_transfer_outer_man [simp]:
```

```
  "local_transfer f (outer_man Q m) = local_transfer f m"
  by (simp add: local_transfer_def)
```

Answers and the local query handler

A transfer that consults the query channel receives the answers as an ordinary function argument, so a proof about the transfer quantifies over them. A question nobody answers is answered $\top$, which claims nothing.

```
type_synonym channel = "query ⇒ answer"
```

The local query handler: answers are a pure function of the local value. Such a handler never asks in turn, so the recursive channel the generator installs around it reduces to the handler itself.

```
definition local_query :: "('dl ⇒ channel) ⇒ ('x,'k,'v,'dl,'dg) man_query" where
  "local_query h m q = sp_return (h (man_local m) q)"

lemma local_query_update [simp]:
  "local_query h (m(|man_ask := ch|)) = local_query h m"
  by (simp add: local_query_def fun_eq_iff)

lemma local_transfer_update [simp]:
  "local_transfer f (m(|man_ask := ch|)) = local_transfer f m"
  by (simp add: local_transfer_def)

lemma ask_with_local_query:
  "ask_with (local_query h) n asked m q
     = sp_return (if n = 0 ∨ q ∈ asked then ⊤ else h (man_local m) q)"
  by (cases n) (simp_all add: local_query_def)

lemma outer_man_local_query [simp]:
  "outer_man (local_query h) m = m(|man_ask := local_query h m|)"
  unfolding outer_man_def
  by (simp add: ask_with_local_query query_depth_def local_query_def fun_eq_iff)
```

The entry counterpart: the alternatives are a pure function of the caller value, so the program reads no unknown and publishes nothing. `f` answers the whole list, which is what lets a local-only specification still offer several alternatives.

```
definition local_enter_transfer ::
  "('dl ⇒ 'dl enter_result list) ⇒ ('x,'k,'v,'dl,'dg) man_enter_transfer"
where
  "local_enter_transfer f m = sp_return (f (man_local m))"

lemma sp_compile_transfer_program_local_transfer:
  "sp_compile (transfer_program (local_transfer f) (Inl x) unknown_of)
     = QueryL x (λa. Answer (DG (f (dg_local a)) bot))"
  "sp_compile (transfer_program (local_transfer f) (Inr g) unknown_of)
     = QueryG g (λa. Answer (DG (f (dg_local a)) bot))"
  by (simp_all add: transfer_program_def transfer_program_at_def local_transfer_def
      dg_read_at_def sp_bind_assoc sp_compile_def sp_compile_with_def
      sp_bind_def sp_return_def sp_read_local_def sp_read_global_def)

lemma traverse_local_transfer_program [simp]:
  "traverse_program (transfer_program (local_transfer f) src unknown_of) τ = DG (f (dg_local (τ src))) bot"
  by (cases src) (simp_all add: sp_compile_transfer_program_local_transfer)

lemma sides_local_transfer_program [simp]:
  "sides_of_program (transfer_program (local_transfer f) src unknown_of) τ k = bot"
  by (cases src) (simp_all add: sp_compile_transfer_program_local_transfer)

lemma dep_local_transfer_program [simp]:
  "dep_program τ (transfer_program (local_transfer f) src unknown_of) = {src}"
  by (cases src) (simp_all add: sp_compile_transfer_program_local_transfer)
```

Local-only combine

The combine counterpart: a pure function of the caller-continuation and callee exit states, no global contact. When both stages — `combine_env`$^\sharp$ and `combine_assign`$^\sharp$ — are local, the whole return pipeline collapses monadically to one pure composition – the sequencing updates `man_local` and extracts nothing – and the compiled combine tree is two reads and an answer, with no contribution and dependencies exactly the two sources.

```
definition local_combine_transfer ::
  "('dl ⇒ 'dl ⇒ 'dl) ⇒ ('x,'k,'v,'dl,'dg) man_combine_transfer"
where
  "local_combine_transfer f m exit = sp_return (f (man_local m) exit)"
```

```
lemma dg_spec_combine_transfer_local:
  assumes "combine_env♯ S ci = local_combine_transfer ce"
    and "combine_assign♯ S ci = local_combine_transfer ca"
  shows "dg_spec_combine_transfer S ci m exit
          = sp_return (ca (ce (man_local m) exit) exit)"
  by (simp add: dg_spec_combine_transfer_def assms local_combine_transfer_def)
```

The reads every compiled transfer makes

A compiled edge tree begins by reading its source unknown, and a compiled combine tree by reading the call site and the callee exit, before the transfer runs at all. Those reads are therefore dependencies of *any* specification's tree, effectful or not – unlike the global key, which appears only when the transfer actually queries it. Coverage arguments need exactly these, so they are stated once here rather than as an exact dependency set that would hold only for one kind of specification.

```
lemma dep_transfer_program_source:
  "src ∈ dep_program τ (transfer_program T src gk)"
  by (simp add: dep_transfer_program)

lemma dep_dg_spec_edge_program_source:
  "src ∈ dep_program τ (dg_spec_edge_program S a src gk)"
  unfolding dg_spec_edge_program_def by (rule dep_transfer_program_source)
```

A default specification, and overriding its fields

`local_dg_spec_template` is the identity local-only specification: every transfer hands back the local value it was given, and a return combine keeps the caller's. An analysis starts from it and overrides the fields it implements – `local_dg_spec_template⦇dgs_assign := ..., dgs_branch := ...⦈` – the way a Goblint `Spec` includes `DefaultSpec` and defines only the transfers it needs.

It is named a template, not a default, because it is not Goblint's default behavior and nothing here treats it as one: it is the record every field of which is still to be chosen.

This is a construction scaffold, not a soundness result. Leaving a field at the identity is a claim about that analysis: that ignoring the edge is sound for it. It is false in general – an assignment analysis that leaves `dgs_assign` at the identity does not track assignments – and nothing here checks it. Whether a specification, defaulted fields included, means anything is settled by `analysis_contract` together with the entry obligation that locale deliberately leaves open — never by how the record was assembled.

It is also not a transcription of Goblint's `IdentitySpec`, and the return combine is where the two differ: both stages here keep the value they are handed, so the composed default keeps the *caller* continuation, whereas Goblint's `combine_env` answers the callee exit and its `combine_assign` answers `man.local`, so the composed default there ends at the callee's value. Both are neutral for their own purpose; this one is neutral for *record update*. A wrapper that overrides only the final stage, such as `ownership_split_dg_spec_st_for` in `Ownership_Split_Exec`, relies on that: were the default to substitute the callee exit first, the wrapper would run against a continuation the caller never had. Changing the default is therefore an audit of every partial override, not a one-line edit.

```
definition local_dg_spec_template :: "('x,'k,'v,'D,'G) dg_spec" where
  "local_dg_spec_template = ⦇
     dgs_skip = local_transfer id,
     dgs_assign = (λx e. local_transfer id),
     dgs_special = (λsc x. local_transfer id),
     dgs_branch = (λb pol. local_transfer id),
     dgs_body = (λp. local_transfer id),
     dgs_return = (λe p. local_transfer id),
     dgs_enter = (λci. local_enter_transfer (λd. [(d, d)])),
     dgs_event = (λev. local_transfer id),
     dgs_combine_env = (λci. local_combine_transfer (λd de. d)),
     dgs_combine_assign = (λci. local_combine_transfer (λd de. d)),
     dgs_query = (λm q. sp_return ⊤) ⦈"
```

The argument-free template must generalize in generated ML. Expanding the local transfer makes its skip field a function value, avoiding the value restriction on the application `local_transfer id` while keeping the template itself named.

```
lemmas [code] = local_dg_spec_template_def[unfolded local_transfer_def]

lemma local_dg_spec_template_simps [simp]:
  "skip♯ local_dg_spec_template = local_transfer id"
  "assign♯ local_dg_spec_template x e = local_transfer id"
  "special♯ local_dg_spec_template sc x = local_transfer id"
  "branch♯ local_dg_spec_template b pol = local_transfer id"
```

```
  "body♯ local_dg_spec_template p = local_transfer id"
  "return♯ local_dg_spec_template eo p = local_transfer id"
  "enter♯ local_dg_spec_template ci = local_enter_transfer (λd. [(d, d)])"
  "event♯ local_dg_spec_template ev = local_transfer id"
  "combine_env♯ local_dg_spec_template ci = local_combine_transfer (λd de. d)"
  "combine_assign♯ local_dg_spec_template ci = local_combine_transfer (λd de. d)"
  "dgs_query local_dg_spec_template m q = sp_return ⊤"
  by (simp_all add: local_dg_spec_template_def)
```

The pure edge dispatch

`local_spec_step` is the pure counterpart of `dg_spec_step`'s dispatch: it selects the operation the edge names. A component's step on an arbitrary edge is this dispatch over its per-edge fields.

```
fun local_spec_step ::
  "('D ⇒ 'D) ⇒ (vname ⇒ exp ⇒ 'D ⇒ 'D) ⇒ (special_call ⇒ vname ⇒ 'D ⇒ 'D)
   ⇒ (exp ⇒ bool ⇒ 'D ⇒ 'D) ⇒ (pname ⇒ 'D ⇒ 'D)
   ⇒ (exp option ⇒ pname ⇒ 'D ⇒ 'D)
   ⇒ (analysis_event ⇒ 'D ⇒ 'D) ⇒ edge_action ⇒ 'D ⇒ 'D"
where
  "local_spec_step sk asn sp br bd rt ev EA_Nop = sk"
| "local_spec_step sk asn sp br bd rt ev (EA_Assign x e) = asn x e"
| "local_spec_step sk asn sp br bd rt ev (EA_Special sc x) = sp sc x"
| "local_spec_step sk asn sp br bd rt ev (EA_Assume b) = br b True"
| "local_spec_step sk asn sp br bd rt ev (EA_AssumeNot b) = br b False"
| "local_spec_step sk asn sp br bd rt ev (EA_Body p) = bd p"
| "local_spec_step sk asn sp br bd rt ev (EA_Ret e p) = rt e p"
| "local_spec_step sk asn sp br bd rt ev (EA_Check l cnd) = ev (Check_Event l cnd)"

fun event_action :: "analysis_event ⇒ edge_action" where
  "event_action (Check_Event l cnd) = EA_Check l cnd"

end
```

6.6 DG_Spec_Sound

```
theory DG_Spec_Sound
  imports DG_Spec "Voblint_CFG.CFG_Transfer"
begin
```

What a manager-native specification owes a concretization

Soundness of a manager-native `dg_spec` relates its returned local value to an environment of analysis-visible globals. The manager maps each global name to a solver key; `genv` reads the global component at those keys.

A transfer starts in the input environment and establishes its result in that environment joined pointwise with its publications. Unchanged globals therefore retain their meaning without being republished. A specification may read and publish several names, all interpreted by the same environment. Concretizations that ignore globals discharge the global obligations vacuously. The unit-name compatibility lemma retains the older sufficient condition that publications alone describe the resulting global state.

What a call's alternatives must establish. The coverage contract is the part that does not mention a strategy program: `enter_runs` and `enter_deps` below do describe an entry program and the continuations it is handed, but what an alternative *means* is settled here, before any of that. `pairs` covers a concrete caller when *some* alternative accounts for it: the caller store lies in the continuation half and the store the callee starts from lies in the entry half, in the same pair. The quantifier is existential because alternatives are a disjunction — a call that splits into two of them has each covering the concrete runs it was computed for, and demanding that every alternative cover every caller would defeat the splitting the list exists to express.

Keeping it separate from any particular program is what lets an entry obligation be discharged once and reused: a proof about how alternatives are folded never has to restate what an alternative means.

```
definition entry_pairs_cover ::
  "('D ⇒ 's set) ⇒ 's ⇒ 's ⇒ 'D enter_result list ⇒ bool"
where
  "entry_pairs_cover gammaD caller entered pairs ⟷
     (∃cont entry. (cont, entry) ∈ set pairs
        ∧ caller ∈ gammaD cont ∧ entered ∈ gammaD entry)"

lemma entry_pairs_coverI [intro]:
  assumes "(cont, entry) ∈ set pairs"
    and "caller ∈ gammaD cont" and "entered ∈ gammaD entry"
  shows "entry_pairs_cover gammaD caller entered pairs"
  using assms unfolding entry_pairs_cover_def by blast

lemma entry_pairs_coverE [elim]:
  assumes "entry_pairs_cover gammaD caller entered pairs"
  obtains cont entry
    where "(cont, entry) ∈ set pairs"
      and "caller ∈ gammaD cont" and "entered ∈ gammaD entry"
  using assms unfolding entry_pairs_cover_def by blast

lemma entry_pairs_cover_Nil [simp]:
  "¬ entry_pairs_cover gammaD caller entered []"
  by (simp add: entry_pairs_cover_def)
```

An entry transfer is a program, so its alternatives cannot be read back out of it: the tree it builds answers into the solver's carrier, and a list of pairs does not live there. What *can* be said is how it behaves under a fixed solution. `enter_runs T m sigma pairs pub` says that under `sigma` the program hands its continuation exactly `pairs` and publishes exactly `pub` on the way, whatever the continuation then does — the reads resolve against `sigma`, so both are determined before any continuation is chosen.

This is what lets a proof name an alternative. Rather than extracting the list after the fact, a caller fixes the `pairs` the continuation receives and reasons under that, which is the ordinary way to reason about a continuation-passing program. `pub` is carried separately because the prefix's own publications are part of what the call establishes: an entry that reads a global and republishes it has those contributions in `pub`, not in anything the continuation returns.

```
definition enter_runs ::
  "('x,'k,'v,'D::bounded_semilattice_sup_bot,'G::bounded_semilattice_sup_bot) man_enter_transfer
   ⇒ ('x,'k,'v,'D,'G) man ⇒ ('x + 'k ⇒ ('D,'G) dg_state)
   ⇒ 'D enter_result list ⇒ ('x + 'k ⇒ ('D,'G) dg_state) ⇒ bool"
where
  "enter_runs T m sigma pairs pub ⟷
     (∀K. traverse_rhs (T m K) sigma = traverse_rhs (K pairs) sigma
          ∧ sides_of_rhs (T m K) sigma = pub ⊔ sides_of_rhs (K pairs) sigma)"
```

```
lemma enter_runsD_traverse:
  assumes "enter_runs T m sigma pairs pub"
  shows "traverse_rhs (T m K) sigma = traverse_rhs (K pairs) sigma"
  using assms unfolding enter_runs_def by blast

lemma enter_runsD_sides:
  assumes "enter_runs T m sigma pairs pub"
  shows "sides_of_rhs (T m K) sigma = pub ⊔ sides_of_rhs (K pairs) sigma"
  using assms unfolding enter_runs_def by blast
```

A pure entry hands over its alternatives directly and publishes nothing.

```
lemma enter_runs_local_enter_transfer [intro]:
  "enter_runs (local_enter_transfer f) m sigma (f (man_local m)) bot"
  by (simp add: enter_runs_def local_enter_transfer_def sp_return_def
      sup_fun_def bot_fun_def)
```

The same at a built manager, where `man_local` has already been read off. A caller whose alternatives are written out as a list — which is how every routed instance states its entry obligation — matches this one by ordinary unification, where the rule above would first have to reduce `man_local (mk_dg_man d key)` under a higher-order variable.

```
lemma enter_runs_local_enter_transfer_mk_dg_man [intro]:
  "enter_runs (local_enter_transfer f) (mk_dg_man d unknown_of) sigma (f d) bot"
  using enter_runs_local_enter_transfer[of f "mk_dg_man d unknown_of" sigma] by simp
```

The converse direction, which a consumer needs when it has an arbitrary run in hand rather than the one above: a pure entry's publication is pinned, so no case analysis on the run is possible.

```
lemma enter_runs_local_pub_bot:
  assumes "enter_runs (local_enter_transfer f) m sigma pairs pub"
  shows "pub = bot"
proof -
  have "sides_of_rhs (local_enter_transfer f m (λ_. Answer bot)) sigma
         = pub ⊔ sides_of_rhs (Answer bot) sigma"
    using assms by (rule enter_runsD_sides)
  then show ?thesis
    by (simp add: local_enter_transfer_def sp_return_def fun_eq_iff sup_fun_def bot_fun_def)
qed
```

Values and publications are not everything the generator needs to know about an entry program: what it *reads* matters too, and reading is invisible in both. An entry that consults a global depends on that unknown even when it publishes `bot` there, and the solver's admissibility arguments are stated over dependencies, not over answers.

`enter_deps` is deliberately a separate relation over the *same* `pairs`: a proof that obtained its alternatives from `enter_runs` and its dependencies from an independently quantified list would be talking about two different executions. Packaging the two together, with `entry_pairs_cover`, is what keeps one call's story single.

```
definition enter_deps ::
  "('x,'k,'v,'D::bounded_semilattice_sup_bot,'G::bounded_semilattice_sup_bot) man_enter_transfer
   ⇒ ('x,'k,'v,'D,'G) man ⇒ ('x + 'k ⇒ ('D,'G) dg_state)
   ⇒ 'D enter_result list ⇒ ('x + 'k) set ⇒ bool"
where
  "enter_deps T m sigma pairs deps ⟷
     (∀K. dep_aux sigma (T m K) = deps ∪ dep_aux sigma (K pairs))"

lemma enter_depsD:
  assumes "enter_deps T m sigma pairs deps"
  shows "dep_aux sigma (T m K) = deps ∪ dep_aux sigma (K pairs)"
  using assms unfolding enter_deps_def by blast
```

A pure entry reads nothing of its own.

```
lemma enter_deps_local_enter_transfer [intro]:
  "enter_deps (local_enter_transfer f) m sigma (f (man_local m)) {}"
  by (simp add: enter_deps_def local_enter_transfer_def sp_return_def)

lemma enter_deps_local_enter_transfer_mk_dg_man [intro]:
  "enter_deps (local_enter_transfer f) (mk_dg_man d unknown_of) sigma (f d) {}"
  using enter_deps_local_enter_transfer[of f "mk_dg_man d unknown_of" sigma] by simp
```

The analysis contract carries everything that is independent of how a call is compiled: the ordinary edge transfers and the return combine. Entry is not here. A $\texttt{enter}^\sharp$ answers a list, which is not an equation's answer, so what makes it sound is a property of the program the routed generator builds from it, stated through

`entry_pairs_cover` against the routed alternative the solved system actually selects — a property this locale alone cannot state, since it never mentions how a call compiles.

So `analysis_contract` is the entry-independent part and not a complete soundness statement: interpreting it alone leaves a call's entry entirely unconstrained. `routed_context` (`Routed_Context`, downstream in this session) is the complete statement, extending this contract with that entry obligation for the routed equation shape, and is what an analysis should be asked to establish.

Proof-level compiled combine form

A return combine compiled the way an edge transfer is: read the caller continuation and the callee exit from two unknowns, and run the composed combine against them. No generator builds this. The routed call program already holds the alternative's own continuation, so it runs `dg_spec_combine_transfer` against that value directly and never reads a caller unknown a second time.

It lives here, ahead of the locale that consumes it, because *stating* the return obligation at a pair of addresses is all it is for: a proof that quantifies over the pair needs a program to quantify over. Reading it in `Voblint_Framework.DG_Spec`, among the formers a generator really does build, invited exactly the wrong conclusion.

Both levels earn their place, which is why there are two of them. `combine_transfer_program` is used at an arbitrary `man_combine_transfer` — by the ownership-split observation equations in `DG_Ownership_Split_Spec`, and by the relational domain's — while `dg_spec_combine_program` is used at a selected specification, by `combine_sound_program` below, which restates the return obligation at two *solver addresses*. That is what the value form cannot express: a caller bounding the program from a post-solution reads it at two unknowns, so the obligation has to name the addresses, not the values they hold. `combine_program_at` is the one internal step, kept for symmetry with the edge driver rather than for a second consumer.

```
definition combine_program_at ::
  "('x,'k,'v,'dl,'dg) man_combine_transfer
   ⇒ 'x + 'k ⇒ 'x + 'k ⇒ ('v ⇒ 'k)
   ⇒ ('x,'k,('dl::bot,'dg) dg_state,'dl) strategy_program"
where
  "combine_program_at transfer src_cc src_ex unknown_of =
     do {
       dc ← dg_read_at src_cc;
       de ← dg_read_at src_ex;
       transfer (mk_dg_man dc unknown_of) de
     }"

definition combine_transfer_program ::
  "('x,'k,'v,'dl::bot,'dg::bot) man_combine_transfer ⇒ 'x + 'k ⇒ 'x + 'k ⇒ ('v ⇒ 'k)
   ⇒ ('x,'k,('dl,'dg) dg_state,('dl,'dg) dg_state) strategy_program"
where
  "combine_transfer_program T src_cc src_ex unknown_of =
     sp_map (λd. DG d ⊥) (combine_program_at T src_cc src_ex unknown_of)"

definition dg_spec_combine_program ::
  "('x,'k,'v,'dl::bot,'dg::bot) dg_spec ⇒ call_info ⇒ 'x + 'k ⇒ 'x + 'k ⇒ ('v ⇒ 'k)
   ⇒ ('x,'k,('dl,'dg) dg_state,('dl,'dg) dg_state) strategy_program"
where
  "dg_spec_combine_program S ci src_cc src_ex unknown_of =
     combine_transfer_program (dg_spec_combine_transfer S ci) src_cc src_ex unknown_of"
```

What compilation costs an observation here: the two source reads, and then the composed combine run at the values they produced. The counterpart of `Voblint_Framework.DG_Spec`'s edge bridges, and untagged for the same reason.

```
lemma traverse_combine_transfer_program:
  "traverse_program (combine_transfer_program T src_cc src_ex unknown_of) τ
     = traverse_rhs (sp_compile_with (λd. DG d ⊥)
         (T (mk_dg_man (dg_local (τ src_cc)) unknown_of) (dg_local (τ src_ex)))) τ"
  by (simp add: combine_transfer_program_def combine_program_at_def sp_compile_with_def
      sp_bind_def)

lemma sides_combine_transfer_program:
  "sides_of_program (combine_transfer_program T src_cc src_ex unknown_of) τ
     = sides_of_rhs (sp_compile_with (λd. DG d ⊥)
         (T (mk_dg_man (dg_local (τ src_cc)) unknown_of) (dg_local (τ src_ex)))) τ"
  by (simp add: combine_transfer_program_def combine_program_at_def sp_compile_with_def
      sp_bind_def)

lemma dep_combine_transfer_program:
```

```
  "dep_program τ (combine_transfer_program T src_cc src_ex unknown_of)
     = insert src_cc (insert src_ex
         (dep_aux τ (sp_compile_with (λd. DG d ⊥)
            (T (mk_dg_man (dg_local (τ src_cc)) unknown_of) (dg_local (τ src_ex))))))"
  by (simp add: combine_transfer_program_def combine_program_at_def sp_compile_with_def
      sp_bind_def)
```

What the compiled form observes when both stages are local — the combine counterparts of the edge-tree facts in `Voblint_Framework.DG_Spec`.

```
lemma traverse_local_combine_program [simp]:
  "traverse_program (combine_transfer_program (local_combine_transfer h) src_cc src_ex gk) τ
     = DG (h (dg_local (τ src_cc)) (dg_local (τ src_ex))) bot"
  by (simp add: traverse_combine_transfer_program local_combine_transfer_def
      sp_compile_with_def sp_return_def)

lemma sides_local_combine_program [simp]:
  "sides_of_program (combine_transfer_program (local_combine_transfer h) src_cc src_ex gk) τ k
     = bot"
  by (simp add: sides_combine_transfer_program local_combine_transfer_def
      sp_compile_with_def sp_return_def)

lemma dep_local_combine_program [simp]:
  "dep_program τ (combine_transfer_program (local_combine_transfer h) src_cc src_ex gk)
     = {src_cc, src_ex}"
  by (simp add: dep_combine_transfer_program local_combine_transfer_def
      sp_compile_with_def sp_return_def)
```

What a manager-native specification owes

A specification names its global facts by `'v`, and a key map places each name at a solver unknown. The global environment a valuation holds under a key map is what it holds at those unknowns; the same reading of a program's side-effects is the environment the program publishes.

```
definition genv :: "('v ⇒ 'k) ⇒ ('x + 'k ⇒ ('dl, 'dg) dg_state) ⇒ 'v ⇒ 'dg" where
  "genv key τ v = dg_global (τ (Inr (key v)))"
```

Applied to a name, the environment is the lookup it stands for.

```
declare genv_def [simp]
```

Every name at one key reads one value: the environment of a single slot.

```
lemma genv_const [simp]: "genv (λ_. k) τ = (λ_. dg_global (τ (Inr k)))"
  by (simp add: fun_eq_iff)
```

What an edge's compiled program answers under τ: its local result and the global environment its side-effects publish. The combine counterparts run the composed combine at a continuation `dc` and a callee exit `de`.

```
abbreviation edge_out where
  "edge_out S a src key τ ≡ dg_local (traverse_program (dg_spec_edge_program S a src key) τ)"

abbreviation edge_pub where
  "edge_pub S a src key τ ≡ genv key (sides_of_program (dg_spec_edge_program S a src key) τ)"

abbreviation combine_out where
  "combine_out S ci key dc de τ ≡
     dg_local (traverse_rhs (sp_compile_with (λd. DG d ⊥)
       (dg_spec_combine_transfer S ci (mk_dg_man dc key) de)) τ)"

abbreviation combine_pub where
  "combine_pub S ci key dc de τ ≡
     genv key (sides_of_rhs (sp_compile_with (λd. DG d ⊥)
       (dg_spec_combine_transfer S ci (mk_dg_man dc key) de)) τ)"
```

The contract a specification signs: its concretization is monotone, an edge's program over-approximates the concrete edge step, and its combine over-approximates the concrete return. Both obligations read the globals from the environment the valuation holds, and describe the result against that environment joined with what the program publishes: a global the step does not write is still what was read, so a step owes no publication for it.

```
locale analysis_contract =
  fixes S :: "('x,'k,'v,'D::bounded_semilattice_sup_bot,
               'G::bounded_semilattice_sup_bot) dg_spec"
    and γ_DG :: "'D ⇒ ('v ⇒ 'G) ⇒ store set"
    and 𝒢 :: "vname ⇒ bool"
```

```
  assumes spec_wf: "dg_spec_wf S"
    and gammaDG_mono:
      "⟦d ≤ d'; e ≤ e'⟧ ⟹ γDG d e ⊆ γDG d' e'"
    and step_sound:
      "edge_collect a (γDG (dg_local (τ src)) (genv key τ))
         ⊆ γDG (edge_out S a src key τ) (genv key τ ⊔ edge_pub S a src key τ)"
    and combine_sound:
      "⟦s ∈ γDG q (genv key τ); t ∈ γDG r (genv key τ)⟧ ⟹
        combine_collect G (ci_dst ci) s t
          ∈ γDG (combine_out S ci key q r τ) (genv key τ ⊔ combine_pub S ci key q r τ)"
```

The obligation is stated at *values* rather than at two unknown reads, because the routed call boundary applies combine at the continuation the entry produced, which is not in general what any unknown holds. Where the continuation *is* the caller's own solved value — the monovariant call shape — the tree form below is the instance at that value, so nothing is lost for a generator that reads both sides from unknowns.

```
lemma (in analysis_contract) combine_sound_program:
  assumes sc: "s ∈ γDG (dg_local (τ src_cc)) (genv key τ)"
    and se: "t ∈ γDG (dg_local (τ src_ex)) (genv key τ)"
  shows "combine_collect G (ci_dst ci) s t
          ∈ γDG (dg_local (traverse_program (dg_spec_combine_program S ci src_cc src_ex key) τ))
              (genv key τ ⊔ genv key (sides_of_program (dg_spec_combine_program S ci src_cc src_ex key) τ))"
  using combine_sound[where q = "dg_local (τ src_cc)" and r = "dg_local (τ src_ex)"
      and τ = τ and key = key and ci = ci, OF sc se]
  by (simp add: dg_spec_combine_program_def traverse_combine_transfer_program
      sides_combine_transfer_program)
```

A specification with a single global fact owes the contract at the environment that holds it. The stricter obligations below, which describe the result against what the program publishes alone, imply it, because the published value lies below its join with the one read.

```
lemma analysis_contract_unitI:
  fixes S :: "('x,'k,unit,'D::bounded_semilattice_sup_bot,
               'G::bounded_semilattice_sup_bot) dg_spec"
    and gm :: "'D ⇒ 'G ⇒ store set"
  assumes wf: "dg_spec_wf S"
    and mono: "⋀d d' g g'. ⟦d ≤ d'; g ≤ g'⟧ ⟹ gm d g ⊆ gm d' g'"
    and step: "⋀a τ src gk.
      edge_collect a (gm (dg_local (τ src)) (dg_global (τ (Inr gk))))
        ⊆ gm (edge_out S a src (λ_. gk) τ)
            (dg_global (sides_of_program (dg_spec_edge_program S a src (λ_. gk)) τ (Inr gk)))"
    and comb: "⋀s τ dc gk t de ci.
      ⟦s ∈ gm dc (dg_global (τ (Inr gk))); t ∈ gm de (dg_global (τ (Inr gk)))⟧ ⟹
        combine_collect G (ci_dst ci) s t
          ∈ gm (combine_out S ci (λ_. gk) dc de τ)
              (dg_global (sides_of_rhs (sp_compile_with (λd. DG d ⊥)
                 (dg_spec_combine_transfer S ci (mk_dg_man dc (λ_. gk)) de)) τ (Inr gk)))"
  shows "analysis_contract S (λd e. gm d (e ())) G"
proof
  show "dg_spec_wf S" by (rule wf)
next
  fix d d' :: 'D and e e' :: "unit ⇒ 'G"
  assume "d ≤ d'" "e ≤ e'"
  then show "gm d (e ()) ⊆ gm d' (e' ())" by (simp add: mono le_funD)
next
  fix a τ src and key :: "unit ⇒ 'k"
  have key: "key = (λ_. key ())" by (rule ext) simp
  have "edge_collect a (gm (dg_local (τ src)) (genv key τ ()))
          ⊆ gm (edge_out S a src key τ) (edge_pub S a src key τ ())"
    using step[of a τ src "key ()"] by (subst (1 2 3) key) (simp add: genv_def)
  also have "... ⊆ gm (edge_out S a src key τ) ((genv key τ ⊔ edge_pub S a src key τ) ())"
    by (rule mono) simp_all
  finally show "edge_collect a (gm (dg_local (τ src)) (genv key τ ()))
          ⊆ gm (edge_out S a src key τ) ((genv key τ ⊔ edge_pub S a src key τ) ())" .
next
  fix s dc t de ci and key :: "unit ⇒ 'k" and τ :: "'x + 'k ⇒ ('D, 'G) dg_state"
  assume s: "s ∈ gm dc (genv key τ ())" and t: "t ∈ gm de (genv key τ ())"
  have key: "key = (λ_. key ())" by (rule ext) simp
  have "combine_collect G (ci_dst ci) s t
          ∈ gm (combine_out S ci key dc de τ) (combine_pub S ci key dc de τ ())"
    using comb[of s dc τ "key ()" t de ci] s t by (subst (1 2 3) key) (simp add: genv_def)
  then show "combine_collect G (ci_dst ci) s t
          ∈ gm (combine_out S ci key dc de τ) ((genv key τ ⊔ combine_pub S ci key dc de τ) ())"
    by (rule subsetD[OF mono, rotated 2]) simp_all
qed
```

```
end
```

6.7 Transfer_Algebra

```
theory Transfer_Algebra
  imports "Voblint_CFG.CFG_Transfer" "Voblint_Domain.Nonrelational_State"
begin
```

The abstract-state algebra a whole-state transfer computes in

A Base-style analysis carries one pointwise abstract state per program point: a map from variable names to abstract values. This theory fixes the operations its transfer functions are assembled from – how a call resets the callee frame and binds its formals, how a return merges caller against callee and publishes the result value – together with their soundness and monotonicity against `gamma_state`.

It defines no interface. What an analysis supplies to the framework is a `dg_spec`: one manager-native transfer per edge action, taking the manager and returning a program. This theory is the algebra those transfers compute in, stated once so that no whole-state domain re-proves it.

Most of the operations themselves live lower still, in the language session, because they are generic in the state's codomain and the concrete semantics runs the very same constants: `combine_env`, `enter_frame`, `bind_formals`, `combine_assign` and `enter_binding` are all shared. What is genuinely new here is their meaning against a concretization, which is the first point at which the domain session and the CFG session are both in scope.

The structural environment combine

```
lemma combine_env_mono:
  fixes sc1 sc2 se1 se2 :: "'a::order abs_state"
  assumes "sc1 ≤ sc2" and "se1 ≤ se2"
  shows "combine_env 𝒢 sc1 se1 ≤ combine_env 𝒢 sc2 se2"
  using assms by (auto simp: combine_env_def le_fun_def)
```

Soundness of the abstract combine: combining a caller store (sound for sc) with a callee-exit store (sound for se) yields a store sound for `combine_env 𝒢 sc se`. A pure `numeric_domain` fact – independent of any transfer function – reused by both the interprocedural constraint-system soundness and the effectful pipeline. `combine_env` is the fixed structural merge the Base call boundary uses; an analysis that wants a different one overrides its specification's own environment stage instead.

```
lemma combine_env_sound [intro]:
  fixes dc de :: "'a::numeric_domain abs_state"
  assumes sc: "s ∈ γ dc" and se: "t ∈ γ de"
  shows "combine_env 𝒢 s t ∈ γ (combine_env 𝒢 dc de)"
  using assms by (auto simp: gamma_state_def le_fun_def)
```

Generic soundness and monotonicity helpers

The two facts a call boundary keeps re-deriving: writing one variable, and binding a list of formals, each preserve soundness and monotonicity. They are stated for an arbitrary domain and no transfer function, so every analysis picks them up rather than proving its own copy.

```
lemma gamma_state_upd [intro]:
  fixes d :: "'a::numeric_domain abs_state"
  assumes s: "s ∈ γ d" and v: "v ∈ γ a"
  shows "s(x := v) ∈ γ (d(x := a))"
  using s v unfolding gamma_state_def by auto
```

Binding formals preserves soundness: pointwise-sound actual values bound to the same formals yield a sound entry state.

```
lemma bind_formals_sound [intro]:
  fixes d :: "'a::numeric_domain abs_state"
  assumes s: "s ∈ γ d"
    and vals: "list_all2 (λv a. v ∈ γ a) vs avs"
  shows "bind_formals xs vs s ∈ γ (bind_formals xs avs d)"
  using assms apply (induction xs arbitrary: vs avs s d)
  subgoal
    by simp
  subgoal for _ _ vs avs
    by (cases vs; cases avs) (auto simp: gamma_state_upd)
  done

lemma bind_formals_mono:
  fixes d1 d2 :: "'a::order abs_state"
  assumes base: "d1 ≤ d2"
    and vals: "list_all2 (≤) avs1 avs2"
```

```
  shows "bind_formals xs avs1 d1 ≤ bind_formals xs avs2 d2"
  using assms apply (induction xs arbitrary: avs1 avs2 d1 d2)
  subgoal
    by simp
  subgoal for _ _ avs1 avs2
    by (cases avs1; cases avs2) (auto simp add: le_fun_def)
  done
```

Procedure entry: binding formals and resetting the frame

Procedure entry itself is not defined here. `enter_frame` and `enter_binding` (`Voblint_VIMP.VIMP_Proc`) are already generic in the state's codomain, and the concrete semantics is `enter_binding` at the reset value `0` and the evaluator `aval` (`call_enter`). An abstract domain runs the same constant at its own `top` and its own evaluator, so the two can never drift apart: there is one definition and two instances of it.

What this theory adds is the pair of facts that relate those two instances – the concrete entered store lands in the concretization of the abstract one, and the abstract one is monotone. Neither is expressible below here, because γ lives in the domain session and `enter_binding` in the language session.

```
lemma enter_frame_sound [intro]:
  fixes reset_val :: "'a::numeric_domain"
  assumes sv: "s ∈ γ d" and reset_full: "γ reset_val = UNIV"
  shows "enter_state 𝒢 s ∈ γ (enter_frame 𝒢 reset_val d)"
  unfolding gamma_state_def enter_state_def
  using gamma_stateD[OF sv] reset_full by auto

lemma enter_frame_mono:
  fixes reset_val :: "'a::order"
  assumes "d1 ≤ d2"
  shows "enter_frame 𝒢 reset_val d1 ≤ enter_frame 𝒢 reset_val d2"
  by (simp add: assms le_funD le_funI)
```

`bind_formals`'s own soundness and monotonicity are relational in the actual values, but that is not the shape a domain owns: an evaluator's soundness and monotonicity are pointwise in one actual. This turns the one into the other, so the entry lemmas below can take the fact a caller already has and nobody builds a `list_all2` by hand.

```
lemma list_all2_map_mapI [intro]:
  assumes "⋀x. x ∈ set xs ⟹ R (f x) (g x)"
  shows "list_all2 R (map f xs) (map g xs)"
  using assms by (induction xs) auto

lemma enter_binding_sound_list_all2 [intro]:
  fixes reset_val :: "'a::numeric_domain"
  assumes sv: "s ∈ γ d" and reset_full: "γ reset_val = UNIV"
    and vals: "list_all2 (λv a. v ∈ γ a)
                 (map (λe. ⟦e⟧ₑ s) es) (map (λe. aval_abs e d) es)"
  shows "enter_binding 𝒢 0 aval xs es s
           ∈ γ (enter_binding 𝒢 reset_val aval_abs xs es d)"
  unfolding enter_binding_def enter_state_def[symmetric]
  by (rule bind_formals_sound[OF enter_frame_sound[OF sv reset_full] vals])

lemma enter_binding_sound [intro]:
  fixes reset_val :: "'a::numeric_domain"
  assumes sv: "s ∈ γ d" and reset_full: "γ reset_val = UNIV"
    and eval: "⋀e. e ∈ set es ⟹ ⟦e⟧ₑ s ∈ γ (aval_abs e d)"
  shows "enter_binding 𝒢 0 aval xs es s
           ∈ γ (enter_binding 𝒢 reset_val aval_abs xs es d)"
proof (rule enter_binding_sound_list_all2[OF sv reset_full])
  show "list_all2 (λv a. v ∈ γ a)
          (map (λe. ⟦e⟧ₑ s) es) (map (λe. aval_abs e d) es)"
    by (rule list_all2_map_mapI) (rule eval)
qed

lemma enter_binding_mono:
  fixes reset_val :: "'a::order"
  assumes state_le: "d1 ≤ d2"
    and eval_mono: "⋀e. e ∈ set es ⟹ aval_abs e d1 ≤ aval_abs e d2"
  shows "enter_binding 𝒢 reset_val aval_abs xs es d1
           ≤ enter_binding 𝒢 reset_val aval_abs xs es d2"
  unfolding enter_binding_def
proof (rule bind_formals_mono[OF enter_frame_mono[OF state_le]])
  show "list_all2 (≤) (map (λe. aval_abs e d1) es) (map (λe. aval_abs e d2) es)"
    by (rule list_all2_map_mapI) (rule eval_mono)
qed
```

A call site reads the formals and actuals off `call_info` rather than passing them separately, but that projection stays at the call site: a domain writes `enter_binding 𝒢 top ev (ci_formals ci) (ci_args ci)`, exactly as the return side writes `combine♯ 𝒢 (ci_dst ci)`. No constant here wraps it.

The caller's own continuation is not computed here either. Goblint's `Spec.enter` answers caller/callee pairs and `constraints.ml` hands the caller half on to `combine_env`; a specification's entry field answers the same pairs, so a domain that wants to drop caller facts a callee could invalidate — Goblint's `varEq` filters its caller state by taint — returns the filtered state as the continuation half of its own alternatives. Nothing in this algebra chooses that half: the operations below are what a whole-state domain computes each half *with*.

The structural return combine

The return-value write is a single-slot update, hence monotone in both the written value and the state it writes into. Any combine built over it inherits monotonicity from this one fact.

```
lemma combine_assign_mono:
  fixes s1 s2 :: "'a::order abs_state"
  assumes v: "v1 ≤ v2" and s: "s1 ≤ s2"
  shows "combine_assign dst v1 s1 ≤ combine_assign dst v2 s2"
  using assms by (cases dst) (auto simp: le_fun_def)
```

Return combination joins caller locals with callee globals and then assigns the callee's `ret_var` to the optional destination. The ordinary abstract state update publishes the result without domain-specific return machinery.

```
definition combine_collect_abs ::
    "(vname ⇒ bool) ⇒ vname option ⇒ 'a abs_state ⇒ 'a abs_state ⇒ 'a abs_state"
    ("combine♯") where
  "combine_collect_abs 𝒢 dst dc de =
     combine_assign dst (de ret_var) (combine_env 𝒢 dc de)"

lemma combine_collect_abs_mono:
  fixes dc1 dc2 de1 de2 :: "'a::order abs_state"
  assumes c: "dc1 ≤ dc2" and e: "de1 ≤ de2"
  shows "combine♯ 𝒢 dst dc1 de1 ≤ combine♯ 𝒢 dst dc2 de2"
  unfolding combine_collect_abs_def
  by (rule combine_assign_mono[OF le_funD[OF e] combine_env_mono[OF c e]])
```

The binary env-combine is the destination-free instance of the return-threaded combine: with no destination the return slot is not written. Together with the writing case these reduce the abstract combine under a `cases dst`, so no caller unfolds its definition by hand.

```
lemma combine_collect_abs_None[simp]:
  "combine♯ 𝒢 None a b = combine_env 𝒢 a b"
  by (simp add: combine_collect_abs_def)

lemma combine_collect_abs_Some[simp]:
  "combine♯ 𝒢 (Some x) a b = (combine_env 𝒢 a b)(x := b ret_var)"
  by (simp add: combine_collect_abs_def)
```

Soundness of the structural return combine

Soundness of the abstract combine including result publication. A pure `numeric_domain` fact: the destination slot is sound because the callee's `ret_var` slot is, and every other slot is handled by ⟦`?s ∈ γ ?dc; ?t ∈ γ ?de`⟧ ⟹ `combine_env ?𝒢 ?s ?t ∈ γ (combine_env ?𝒢 ?dc ?de)`.

```
lemma combine_collect_sound [intro]:
  fixes dc de :: "'a::numeric_domain abs_state"
  assumes sc: "s ∈ γ dc" and se: "t ∈ γ de"
  shows "combine_collect 𝒢 dst s t ∈ γ (combine♯ 𝒢 dst dc de)"
  unfolding combine_collect_def combine_collect_abs_def
  using combine_env_sound[OF sc se] gamma_stateD[OF se]
  by (cases dst) (auto simp add: gamma_state_upd)
```

The C-faithful initial store set

`cinit_stores` (`Voblint_VIMP.VIMP_Globals`) is the C-faithful initial store set. Any analysis that uses a domain-specific abstract seed `s0` satisfying `cinit_stores 𝒢 ⊆ γ s0` may state its soundness theorem against `cinit_stores 𝒢` rather than `UNIV`, matching VIMP's C-like initialization semantics.

end

6.8 State_Restriction

```
theory State_Restriction
  imports "Voblint_VIMP.VIMP_Globals" "Voblint_Domain.Nonrelational_State"
    "Voblint_Domain.Reachability_Lift"
begin
```

Local/global restriction of abstract states

A variable classifier selects one half of an abstract state and replaces the other with $\bot$. Both restrictions specialize the generic `combine_env` selector. Their join recovers the original state; they preserve joins, are idempotent, and annihilate each other.

```
definition restrict_local_for ::
  "(vname => bool) => 'a::bounded_semilattice_sup_bot abs_state => 'a abs_state" where
  "restrict_local_for 𝒢 sigma = combine_env 𝒢 sigma bot"

definition restrict_global_for ::
  "(vname => bool) => 'a::bounded_semilattice_sup_bot abs_state => 'a abs_state" where
  "restrict_global_for 𝒢 sigma = combine_env 𝒢 bot sigma"

lemma restrict_local_for_join [simp]:
  "restrict_local_for 𝒢 (A ⊔ B) = restrict_local_for 𝒢 A ⊔ restrict_local_for 𝒢 B"
  unfolding restrict_local_for_def sup_fun_def by (rule ext) simp

lemma restrict_global_for_join [simp]:
  "restrict_global_for 𝒢 (A ⊔ B) = restrict_global_for 𝒢 A ⊔ restrict_global_for 𝒢 B"
  unfolding restrict_global_for_def sup_fun_def by (rule ext) simp

lemma restrict_local_for_idem [simp]:
  "restrict_local_for 𝒢 (restrict_local_for 𝒢 A) = restrict_local_for 𝒢 A"
  unfolding restrict_local_for_def by (rule ext) simp

lemma restrict_global_for_idem [simp]:
  "restrict_global_for 𝒢 (restrict_global_for 𝒢 A) = restrict_global_for 𝒢 A"
  unfolding restrict_global_for_def by (rule ext) simp

lemma map_lift_restrict_global_for_idem [simp]:
  fixes x :: "'a::bounded_semilattice_sup_bot abs_state lifted"
  shows "map_lift (restrict_global_for 𝒢) (map_lift (restrict_global_for 𝒢) x)
         = map_lift (restrict_global_for 𝒢) x"
  unfolding map_lift_comp o_def by simp

lemma restrict_local_for_restrict_global_for_bot [simp]:
  "restrict_local_for 𝒢 (restrict_global_for 𝒢 A) = bot"
  unfolding restrict_local_for_def restrict_global_for_def by (rule ext) simp

lemma restrict_global_for_restrict_local_for_bot [simp]:
  "restrict_global_for 𝒢 (restrict_local_for 𝒢 A) = bot"
  unfolding restrict_local_for_def restrict_global_for_def by (rule ext) simp

lemma restrict_local_for_global_join [simp]:
  "restrict_local_for 𝒢 d ⊔ restrict_global_for 𝒢 d = d"
  unfolding restrict_local_for_def restrict_global_for_def sup_fun_def
  by (rule ext) simp

lemma restrict_global_for_local_join [simp]:
  "restrict_global_for 𝒢 d ⊔ restrict_local_for 𝒢 d = d"
  unfolding restrict_local_for_def restrict_global_for_def sup_fun_def
  by (rule ext) simp
```

The selector equals the join of its disjoint local and global projections. This bridge lets state proofs use the restriction algebra without unfolding `combine_env`.

```
lemma combine_env_for_eq_restrictions:
  "combine_env 𝒢 sc se =
     restrict_local_for 𝒢 sc ⊔ restrict_global_for 𝒢 se"
  unfolding combine_env_def restrict_local_for_def restrict_global_for_def
    sup_fun_def
  by (rule ext) simp
```

Lifted restrictions

The lifted rules transport the same join homomorphisms to reachable states.

```
lemma map_lift_restrict_local_for_join [simp]:
```

```
  "map_lift (restrict_local_for 𝒢) (a ⊔ b)
     = map_lift (restrict_local_for 𝒢) a ⊔ map_lift (restrict_local_for 𝒢) b"
  by (rule map_lift_sup) simp

lemma map_lift_restrict_global_for_join [simp]:
  "map_lift (restrict_global_for 𝒢) (a ⊔ b)
     = map_lift (restrict_global_for 𝒢) a ⊔ map_lift (restrict_global_for 𝒢) b"
  by (rule map_lift_sup) simp
```

Splitting and rejoining

Routing a state's two halves back through `combine_env` recovers it exactly: each half is already bottom outside the names it owns.

```
lemma combine_env_restrict_id [simp]:
  "combine_env 𝒢 (restrict_local_for 𝒢 sigma) (restrict_global_for 𝒢 sigma) = sigma"
  by (simp add: combine_env_for_eq_restrictions)
```

Reading one selected name

The two pointwise equations, so a proof about a single variable never has to unfold `combine_env`. The nesting law is what lets a combine step read a returned value straight out of the callee exit: every call site owns `ret_var` as its own compiler-internal name and never a user-declared global (`reserved_ret_var`), so routing it through `combine_env` a second time would change nothing.

```
lemma combine_env_local_eq [simp]:
  "¬ 𝒢 x ⟹ combine_env 𝒢 sc se x = sc x"
  by (simp add: combine_env_def)

lemma combine_env_global_eq [simp]:
  "𝒢 x ⟹ combine_env 𝒢 sc se x = se x"
  by (simp add: combine_env_def)

lemma combine_env_combine_env_left [simp]:
  "combine_env 𝒢 (combine_env 𝒢 dc g) (combine_env 𝒢 de g) = combine_env 𝒢 dc g"
  by (auto simp: combine_env_def)

end
```

6.9 DG_Ownership_Split_Spec

```
theory DG_Ownership_Split_Spec
  imports DG_Local_State_Spec State_Restriction "Voblint_Solver.Strategy_Program_Fold"
begin
```

Lifting a whole-state analysis onto the global channel

Every other specification in this session is local-only: it answers from the local value and never touches `man_global` or `man_sideg`. This theory takes such a specification and returns one that does the opposite, and is the reason those capabilities exist.

The lifted specification uses one carrier for `D` and `G`, splitting each state by variable ownership: `combine_env`'s predicate is the source language's own declaration of which names are global, so the global names live on the shared channel and the rest on the program point's own unknown. The split is not a representation choice: a return recombines two concrete stores by that same ownership rule, keeping the caller's locals and taking the globals from the callee.

Each lifted transfer therefore reads the shared fact, rejoins it with the local half to recover the whole state, runs the argument's transfer on a manager holding that whole state, and splits the result back – publishing the global half with `man_sideg` and answering with the local half. Its compiled tree reads the source, queries the global key, publishes a `Side`, and answers, which is what makes it the only specification here whose equations carry a genuine `QueryG` and `Side`.

Because it consumes and produces a specification, it is a lifter in Goblint's sense – a `Spec2Spec` functor, like the privatizations that give a Base analysis its shared-state discipline – not a second way of writing an analysis. The argument keeps its own transfers; only what a transfer's local value means, and what happens to its result, change.

The construction is generic in the carrier – the same three operations exist on function-valued states and on the solver's association lists – but not in the ownership rule, which is always `combine_env` and the two restrictions to the global and the local half at that carrier.

The ownership-splitting wrapper

The pattern is stated once, over any carrier that can merge a local and a global half and project each back out. Which carrier is a separate question from what the wrapper does: the same three operations exist on function-valued states and on the solver's association lists, so the executable mirror is this definition at other arguments rather than a second development.

It wraps a transfer rather than a pure function. What the wrapped transfer sees is a manager whose local value is the merged whole state; its own capabilities are the outer ones, which a whole-state transfer never calls. That is what makes the wrapper a specification-to-specification lifter – Goblint's `Spec2Spec` – instead of a second way of writing a transfer.

```
definition ownership_split_transfer_gen ::
  "('d ⇒ 'd ⇒ 'd) ⇒ ('d ⇒ 'd) ⇒ ('d ⇒ 'd)
   ⇒ ('x,'k,unit,'d::bounded_semilattice_sup_bot,'d) man_transfer
   ⇒ ('x,'k,unit,'d,'d) man_transfer"
where
  "ownership_split_transfer_gen cmb rg rl T m =
     do {
       g ← man_global m ();
       res ← T (m(|man_local := cmb (man_local m) g|));
       _ ← man_sideg m () (rg res);
       sp_return (rl res)
     }"
```

A return combine is the same pattern with two values to merge: the caller continuation becomes the wrapped transfer's local value, and the callee exit is merged against the same shared fact before it is handed over. Merging both sides means the wrapped transfer sees two whole states and needs to know nothing about the split – in particular the callee's globals come from the shared channel, not from its own locally-restricted copy.

```
definition ownership_split_combine_transfer_gen ::
  "('d ⇒ 'd ⇒ 'd) ⇒ ('d ⇒ 'd) ⇒ ('d ⇒ 'd)
   ⇒ ('x,'k,unit,'d::bounded_semilattice_sup_bot,'d) man_combine_transfer
   ⇒ ('x,'k,unit,'d,'d) man_combine_transfer"
where
  "ownership_split_combine_transfer_gen cmb rg rl T m de =
     do {
       g ← man_global m ();
       res ← T (m(|man_local := cmb (man_local m) g|)) (cmb de g);
```

```
      _ ← man_sideg m () (rg res);
      sp_return (rl res)
    }"
```

Entry is the one boundary the pattern cannot be reused at verbatim: a call answers a list of alternatives rather than one successor value. The shared fact is read once and the wrapped entry runs once against the merged caller state, so every alternative comes out of the same whole state — an actual may mention a global, so the alternatives cannot be computed from the local half alone. Each alternative then has both of its halves split, and the answer is the list of split pairs.

What reaches the shared channel is the join, over every alternative, of both halves' global projections. A continuation is as much a product of the merged state as its callee entry is, and the split discards its global half; publishing only one of the two would need a proof that the other carries no global information the environment does not already hold.

```
definition ownership_split_enter_sides ::
  "('d ⇒ 'd) ⇒ ('d::bounded_semilattice_sup_bot) enter_result list ⇒ 'd"
where
  "ownership_split_enter_sides rg pairs =
     (⨆(cont, entry)←pairs. rg cont ⊔ rg entry)"

lemma ownership_split_enter_sides_Nil [simp]:
  "ownership_split_enter_sides rg [] = bot"
  by (simp add: ownership_split_enter_sides_def)

lemma ownership_split_enter_sides_Cons [simp]:
  "ownership_split_enter_sides rg ((cont, entry) # pairs)
     = rg cont ⊔ rg entry ⊔ ownership_split_enter_sides rg pairs"
  by (simp add: ownership_split_enter_sides_def)
```

What the joint publication bounds, and what it is bounded by. The two member rules are what a soundness proof reaches for once it has selected a covering alternative: that pair's two halves are each below what the call published, so a concretization taken against the published global still contains them. They stay untagged — their conclusions are bare inequalities in `rg` and would broaden proof search wherever a $\leq$ goal appears.

```
lemma ownership_split_enter_sides_le_iff:
  "ownership_split_enter_sides rg pairs ≤ g
     ⟷ (∀(cont, entry) ∈ set pairs. rg cont ≤ g ∧ rg entry ≤ g)"
  by (induction pairs) auto

definition ownership_split_enter_transfer_gen ::
  "('d ⇒ 'd ⇒ 'd) ⇒ ('d ⇒ 'd) ⇒ ('d ⇒ 'd)
   ⇒ ('x,'k,unit,'d::bounded_semilattice_sup_bot,'d) man_enter_transfer
   ⇒ ('x,'k,unit,'d,'d) man_enter_transfer"
where
  "ownership_split_enter_transfer_gen cmb rg rl T m =
     do {
       g ← man_global m ();
       pairs ← T (m⦇man_local := cmb (man_local m) g⦈);
       _ ← man_sideg m () (ownership_split_enter_sides rg pairs);
       sp_return (map (λ(cont, entry). (rl cont, rl entry)) pairs)
     }"
```

What the compiled tree reads and publishes

Soundness judges a transfer by these two observations, so they are proved once here, on the pattern at a whole-state transfer, rather than per carrier: the answer is the local projection of the merged whole state, and the contribution at the routed key is its global projection.

```
lemma dg_local_traverse_ownership_split_transfer_gen [simp]:
  "dg_local (traverse_program
             (transfer_program (ownership_split_transfer_gen cmb rg rl (local_transfer f))
                src (λ_. gk)) τ)
     = rl (f (cmb (dg_local (τ src)) (dg_global (τ (Inr gk)))))"
  by (simp add: traverse_transfer_program ownership_split_transfer_gen_def
      local_transfer_def sp_compile_with_def sp_bind_def sp_return_def)

lemma dg_global_sides_ownership_split_transfer_gen [simp]:
  "dg_global (sides_of_program
            (transfer_program (ownership_split_transfer_gen cmb rg rl (local_transfer f))
               src (λ_. gk)) τ (Inr gk))
     = rg (f (cmb (dg_local (τ src)) (dg_global (τ (Inr gk)))))"
  by (simp add: sides_transfer_program ownership_split_transfer_gen_def
      local_transfer_def sp_compile_with_def sp_bind_def sp_return_def)
```

```
lemma dg_local_traverse_ownership_split_combine_transfer_gen [simp]:
  "dg_local (traverse_program
             (combine_transfer_program
                (ownership_split_combine_transfer_gen cmb rg rl (local_combine_transfer h))
                src_cc src_ex (λ_. gk)) τ)
     = rl (h (cmb (dg_local (τ src_cc)) (dg_global (τ (Inr gk))))
             (cmb (dg_local (τ src_ex)) (dg_global (τ (Inr gk)))))"
  by (simp add: traverse_combine_transfer_program ownership_split_combine_transfer_gen_def
      local_combine_transfer_def sp_compile_with_def sp_bind_def sp_return_def)

lemma dg_global_sides_ownership_split_combine_transfer_gen [simp]:
  "dg_global (sides_of_program
            (combine_transfer_program
               (ownership_split_combine_transfer_gen cmb rg rl (local_combine_transfer h))
               src_cc src_ex (λ_. gk)) τ (Inr gk))
     = rg (h (cmb (dg_local (τ src_cc)) (dg_global (τ (Inr gk))))
             (cmb (dg_local (τ src_ex)) (dg_global (τ (Inr gk)))))"
  by (simp add: sides_combine_transfer_program ownership_split_combine_transfer_gen_def
      local_combine_transfer_def sp_compile_with_def sp_bind_def sp_return_def)
```

A whole-state transfer never asks, so the query channel the generator installs around the wrapper leaves it unchanged.

```
lemma ownership_split_transfer_gen_local_outer_man [simp]:
  "ownership_split_transfer_gen cmb rg rl (local_transfer f) (outer_man Q m)
     = ownership_split_transfer_gen cmb rg rl (local_transfer f) m"
  by (simp add: ownership_split_transfer_gen_def local_transfer_def)
```

What the lifter reads, in the same terms. The shared slot is read before the wrapped transfer runs, so it is a dependency of the call whatever the wrapped transfer does — and it stays one even when nothing is published there, which is exactly why dependencies are tracked apart from `enter_runs`'s publications. The trailing `man_sideg` writes rather than reads, so it contributes no dependency of its own.

```
lemma enter_deps_ownership_split_enter_transfer_gen [intro]:
  fixes sigma :: "'x + 'k ⇒ ('d::bounded_semilattice_sup_bot,'d) dg_state"
  assumes T: "enter_deps T (mk_dg_man (cmb d (dg_global (sigma (Inr (unknown_of ())))) unknown_of)
                sigma pairs deps"
  shows "enter_deps (ownership_split_enter_transfer_gen cmb rg rl T) (mk_dg_man d unknown_of) sigma
           (map (λ(cont, entry). (rl cont, rl entry)) pairs)
           ({Inr (unknown_of ())} ∪ deps)"
  unfolding enter_deps_def
proof (intro allI)
  fix K
  show "dep_aux sigma (ownership_split_enter_transfer_gen cmb rg rl T (mk_dg_man d unknown_of) K)
          = ({Inr (unknown_of ())} ∪ deps)
            ∪ dep_aux sigma (K (map (λ(cont, entry). (rl cont, rl entry)) pairs))"
    by (simp add: ownership_split_enter_transfer_gen_def sp_bind_def sp_return_def
        enter_depsD[OF T] Un_assoc)
qed
```

How the lifter behaves under a solution, in the same terms as the transfer it wraps. The manager has to be a concrete `mk_dg_man`: for an arbitrary one `man_sideg m ()` is opaque and the slot it publishes at cannot be named, so there would be nothing to state.

The wrapped transfer is run at the reconstructed caller state, so its alternatives are the ones computed against the whole state rather than the local half — that is the step a caller-local-only entry could not perform. Its own publications survive in `pub`, and the lifter adds exactly one contribution of its own, at the routed slot.

```
lemma enter_runs_ownership_split_enter_transfer_gen [intro]:
  fixes sigma :: "'x + 'k ⇒ ('d::bounded_semilattice_sup_bot,'d) dg_state"
  assumes T: "enter_runs T (mk_dg_man (cmb d (dg_global (sigma (Inr (unknown_of ())))) unknown_of)
                sigma pairs pub"
  shows "enter_runs (ownership_split_enter_transfer_gen cmb rg rl T) (mk_dg_man d unknown_of) sigma
           (map (λ(cont, entry). (rl cont, rl entry)) pairs)
           (pub ⊔ (bot(Inr (unknown_of ()) := DG bot (ownership_split_enter_sides rg pairs))))"
  unfolding enter_runs_def
proof (intro allI conjI)
  fix K
  show "traverse_rhs (ownership_split_enter_transfer_gen cmb rg rl T (mk_dg_man d unknown_of) K) sigma
          = traverse_rhs (K (map (λ(cont, entry). (rl cont, rl entry)) pairs)) sigma"
    by (simp add: ownership_split_enter_transfer_gen_def sp_bind_def sp_return_def
        enter_runsD_traverse[OF T])
next
  fix K
  show "sides_of_rhs (ownership_split_enter_transfer_gen cmb rg rl T (mk_dg_man d unknown_of) K) sigma
          = (pub ⊔ (bot(Inr (unknown_of ()) := DG bot (ownership_split_enter_sides rg pairs))))
```

```
          ⊔ sides_of_rhs (K (map (λ(cont, entry). (rl cont, rl entry)) pairs)) sigma"
    by (simp add: ownership_split_enter_transfer_gen_def sp_bind_def sp_return_def
        enter_runsD_sides[OF T] sup_fun_def fun_upd_def)
       (auto simp: fun_eq_iff ac_simps split: if_splits)
qed
```

The specification-to-specification lifter

Every edge transfer of the argument specification is wrapped, entry through its own list-shaped wrapper, and the whole return pipeline is wrapped once rather than stage by stage: wrapping each stage separately would split and re-merge between them, and the second merge would read the shared fact again instead of the first stage's own result. So `combine_env`$^\sharp$ stays at its identity default and `combine_assign`$^\sharp$ carries the wrapped `dg_spec_combine_transfer` of the argument.

```
definition ownership_split_lift_gen ::
  "('d ⇒ 'd ⇒ 'd) ⇒ ('d ⇒ 'd) ⇒ ('d ⇒ 'd)
   ⇒ ('x,'k,unit,'d::bounded_semilattice_sup_bot,'d) dg_spec
   ⇒ ('x,'k,unit,'d,'d) dg_spec"
where
  "ownership_split_lift_gen cmb rg rl S = local_dg_spec_template⦇
     dgs_skip := ownership_split_transfer_gen cmb rg rl (skip♯ S),
     dgs_assign := (λx e. ownership_split_transfer_gen cmb rg rl (assign♯ S x e)),
     dgs_special := (λsc x. ownership_split_transfer_gen cmb rg rl (special♯ S sc x)),
     dgs_branch := (λb pol. ownership_split_transfer_gen cmb rg rl (branch♯ S b pol)),
     dgs_body := (λp. ownership_split_transfer_gen cmb rg rl (body♯ S p)),
     dgs_return := (λe p. ownership_split_transfer_gen cmb rg rl (return♯ S e p)),
     dgs_enter := (λci. ownership_split_enter_transfer_gen cmb rg rl (enter♯ S ci)),
     dgs_event := (λevt. ownership_split_transfer_gen cmb rg rl (event♯ S evt)),
     dgs_combine_assign :=
       (λci. ownership_split_combine_transfer_gen cmb rg rl (dg_spec_combine_transfer S ci)) ⦈"
```

Both eliminate a constructed wrapper specification and expose the corresponding transfer wrapper, in a terminating direction, so they fire wherever a lifted specification meets the edge or combine dispatch.

```
lemma dg_spec_step_ownership_split_lift_gen [simp]:
  "dg_spec_step (ownership_split_lift_gen cmb rg rl S) a
     = ownership_split_transfer_gen cmb rg rl (dg_spec_step S a)"
  unfolding ownership_split_lift_gen_def by (cases a) simp_all

lemma dgs_enter_ownership_split_lift_gen [simp]:
  "enter♯ (ownership_split_lift_gen cmb rg rl S) ci
     = ownership_split_enter_transfer_gen cmb rg rl (enter♯ S ci)"
  unfolding ownership_split_lift_gen_def by simp

lemma dgs_query_ownership_split_lift_gen [simp]:
  "dgs_query (ownership_split_lift_gen cmb rg rl S) m q = sp_return ⊤"
  unfolding ownership_split_lift_gen_def by simp

lemma dg_spec_combine_transfer_ownership_split_lift_gen [simp]:
  "dg_spec_combine_transfer (ownership_split_lift_gen cmb rg rl S) ci
     = ownership_split_combine_transfer_gen cmb rg rl (dg_spec_combine_transfer S ci)"
  unfolding dg_spec_combine_transfer_def ownership_split_lift_gen_def
  by (simp add: local_transfer_def local_combine_transfer_def)
```

The lifter preserves well-formedness: it reads the shared slot, runs the wrapped transfer at a manager built the same way, publishes once and answers. Every step of that is a closure lemma.

```
lemma sp_wf_dgs_combine_assign_ownership_split_lift_gen [intro]:
  assumes "dg_spec_wf S"
  shows
    "sp_wf (combine_assign♯ (ownership_split_lift_gen cmb rg rl S) ci (mk_dg_man d unknown_of) ex)"
  unfolding ownership_split_lift_gen_def
  by (auto simp: ownership_split_combine_transfer_gen_def
      intro!: sp_wf_bind dg_spec_wf_combine[OF assms])

lemma dg_spec_wf_ownership_split_lift_gen [intro]:
  assumes "dg_spec_wf S"
  shows "dg_spec_wf (ownership_split_lift_gen cmb rg rl S)"
  unfolding dg_spec_wf_def
  by (auto simp: ownership_split_transfer_gen_def ownership_split_enter_transfer_gen_def
      ownership_split_combine_transfer_gen_def
      intro!: sp_wf_bind dg_spec_wf_step_ask[OF assms] dg_spec_wf_enter[OF assms]
        dg_spec_wf_combine[OF assms])

end
```

6.10 DG_Indexed_Generator

theory DG_Indexed_Generator
  imports DG_Constraint_Programs CFG_Enumeration "TD.TD_side_Interface"
    "Voblint_Solver.Strategy_Tree_Side_Buffering"
begin

context
begin

`sp_compile` stays folded here. Every fact this theory needs about a compiled program — what it answers, publishes and reads — is available as a named observation equation stated against `sp_compile` itself, so a proof meets those rather than the compiler's representation.

Keyed equation generators

Builds a strategy-tree equation system over a CFG from supplied edge/combine/extra program hooks, where every unknown carries an extra `'k` key alongside its node – context, activation, or any other routing tag a caller wants the generator to thread through. Specifications, routing conventions, and analysis-specific representations are instantiated above this layer; the generator itself never sees one. The ten parameters, once:

- `pred_sel` — which intra predecessors a node folds, each at its address.
- `site_sel` — which call sites return at a node.
- `route` — the callee context a call site selects, threaded to `cmb` and `extra` and never inspected here.
- `it` — the program for one intra edge.
- `cmb` — the program for one call returning at this node.
- `extra` — the routing protocol's own per-node programs, such as an activation-seed read.
- `buffer_key_at` — the node-owned global buffer address for a context.
- `bot0` — the local value a node starts its fold from.
- d_0 — the local value injected at the CFG entry.
- g_0 — the global value published at the CFG entry.

  Two encodings, and they are not peers. `routed_node_rhs` is the direct, reference one: each publishing hook emits its own `Side`, keyed and folded independently. `routed_node_rhs_buffered` is what production runs — every executable analysis in this development reaches the solver through it. It differs deliberately: contributions return their global part in the answer, so the node's own `buffer_key_at c` is published once, after the whole fold has been evaluated, and the enclosing `buffer_sides` then consolidates *every* key it names — activation seeds included — into one flush apiece. The flush comes before the next local read at a node where at most one call returns, and at the answer elsewhere.

  `routed_node_rhs_buffered_correspondence` relates the two declaratively: equal answer, publications and dependencies at a fixed valuation, under explicit per-hook hypotheses. That is all it says. Nothing here claims the two run the same: equal observations at each valuation leave the iteration order free, and where the first occurrence of a key falls decides which update the solver widens against. Deferring the node-global publication past the contributions is therefore a scheduling decision, not an encoding detail, and swapping the encodings is a question for the executable regression corpus rather than for these theorems. `routed_node_rhs_mono_eq`, `routed_node_rhs_mono_sides` and `routed_node_rhs_mono_deps` close the file by discharging the vendored solver's own `TD_side_mono` preconditions from per-hook properties — well-formedness, traverse/sides monotonicity and dependency monotonicity — for an arbitrary generator instance. They are the hypotheses of the vendored least-solution theorem for the solver without widening and narrowing, not a property of the result. The warrowing solver `TD_side_upd_rule` that the shipped analyses run needs none of them, and no shipped analysis discharges the per-hook properties.

The heterogeneous seeded keyed generator

The one context-generic generator. Every program it folds comes from a hook, so a monovariant and a context-sensitive analysis are both instances, and the generator itself never sees a specification:

- `pred_sel g v ctx` selects the intra predecessors folded as Answers into a node, each paired with the *address* in the solver's valuation space that carries its value. `intra_predecessor_addr_list` over `intra` addresses every predecessor at its own `(pp, 'c)` local unknown. Callee entries are not predecessors: every routed instance reaches them through published seeds.
- `site_sel g v` selects the call sites whose calls return at a node. `call_site_list` reads them off `calls`; an executable instance may answer from an index built once per graph, since the generator only folds the list.
- `it c src a` is the intra edge program itself – an instantiator supplies the compiled right-hand side for one edge action at one source address, e.g. a spec's compiled `dg_spec_edge_program` at the routed key. Which unknowns it reads and which keys it publishes are its own business; the generator makes no assumption about either.
- `cmb` is the procedure-return combine program (already fully abstract: the context-sensitive instance reads the callee exit under the routed context).
- `extra c v` supplies additional per-node programs folded after the intra and combine programs. Their Answers add to the node's local accumulator and their `Side` effects are collected — this is where a frame-entry seed *read* (a `QueryG` of the incoming seed slot) and the caller-side call-entry *publication* (a routed `Side` to the callee seed slot) live. Every routed instance supplies one, the context-insensitive one included: seeding is what activates a callee at all, so no analysis reaching a call leaves this hook empty.

The standard predecessor selection: every intra predecessor is carried by its own `(pp, 'c)` local unknown, so its address is `Inl` of that pair.

```
definition intra_predecessor_addr_list ::
  "cfg ⇒ pp ⇒ 'c ⇒ ((pp × 'c + 'k) × edge_action) list"
where
  "intra_predecessor_addr_list g v ctx =
     map (λ(u, a). (Inl (u, ctx), a)) (intra_predecessor_list g v)"
```

Everything that flows into one unknown, in one list: the intra edges entering `v`, the calls returning at it, and whatever the routing protocol adds there. Both generators fold exactly this list — the buffered one at its own hooks — and every bound, correspondence and monotonicity statement below is about it, so it is named once rather than spelled out at each.

The three membership rules are how a per-hook hypothesis reaches a selected contribution. They stay untagged: the list is a concatenation of two maps and a hook's own list, so a membership goal is decided by `auto` from the definition, and tagging them would put three rules in front of every `set` membership goal in scope.

```
definition routed_contribution_programs ::
  "(cfg ⇒ pp ⇒ 'c ⇒ ((pp × 'c + 'k) × edge_action) list)
   ⇒ (cfg ⇒ pp ⇒ (pp × call_action) list)
   ⇒ (pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c)
   ⇒ ('c ⇒ pp × 'c + 'k ⇒ edge_action
        ⇒ (pp × 'c, 'k, ('d::bounded_semilattice_sup_bot,
                         'h::bounded_semilattice_sup_bot) dg_state,
                        ('d, 'h) dg_state) strategy_program)
   ⇒ ((pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ call_action ⇒ pp ⇒ pp
        ⇒ (pp × 'c, 'k, ('d, 'h) dg_state, ('d, 'h) dg_state) strategy_program)
   ⇒ ((pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ pp
        ⇒ (pp × 'c, 'k, ('d, 'h) dg_state, ('d, 'h) dg_state) strategy_program list)
   ⇒ cfg ⇒ 'c ⇒ pp
   ⇒ (pp × 'c, 'k, ('d, 'h) dg_state, ('d, 'h) dg_state) strategy_program list"
where
  "routed_contribution_programs pred_sel site_sel route it cmb extra g ctx v =
     map (λ(src, a). it ctx src a) (pred_sel g v ctx)
     @ map (λ(cc, ca). cmb route ctx ca cc v) (site_sel g v)
     @ extra route ctx v"
```

Every contribution runs its continuation once, given that each hook does. The fold's observation lemmas all carry that side condition, and this is where a generator instance discharges it: the hooks a context policy supplies are built from `sp_return`, the reads and `sp_publish`, whose closure lemmas fire by `simp`.

```
lemma routed_contribution_programs_wf:
```

```
  assumes "⋀c' src a. sp_wf (it c' src a)"
    and "⋀c' ca cc ex. sp_wf (cmb route c' ca cc ex)"
    and "⋀c' w p. p ∈ set (extra route c' w) ⟹ sp_wf p"
  shows "∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb extra g ctx v).
          sp_wf p"
  using assms by (auto simp: routed_contribution_programs_def)

lemma routed_contribution_programs_combineI:
  "(cc, ca) ∈ set (site_sel g v)
   ⟹ cmb route ctx ca cc v
        ∈ set (routed_contribution_programs pred_sel site_sel route it cmb extra g ctx v)"
  by (force simp: routed_contribution_programs_def)

lemma routed_contribution_programs_extraI:
  "t ∈ set (extra route ctx v)
   ⟹ t ∈ set (routed_contribution_programs pred_sel site_sel route it cmb extra g ctx v)"
  by (simp add: routed_contribution_programs_def)

lemma routed_contribution_programsE:
  assumes "t ∈ set (routed_contribution_programs pred_sel site_sel route it cmb extra g ctx v)"
  obtains src a where "(src, a) ∈ set (pred_sel g v ctx)" "t = it ctx src a"
    | cc ca where "(cc, ca) ∈ set (site_sel g v)" "t = cmb route ctx ca cc v"
    | "t ∈ set (extra route ctx v)"
  using assms by (auto simp: routed_contribution_programs_def)
```

The local value a node's fold starts from: `bot0`, joined with the injected d_0 at the CFG entry. An abbreviation, so every statement that spells the conditional out prints as `rhs_init` and needs no unfolding. `rhs_init_global` is the global counterpart: the injected g_0 at the CFG entry, $\bot$ elsewhere.

```
abbreviation rhs_init :: "cfg ⇒ 'd::sup ⇒ 'd ⇒ pp ⇒ 'd" where
  "rhs_init g bot0 d₀ v ≡ (if v = cfg_entry g then bot0 ⊔ d₀ else bot0)"

abbreviation rhs_init_global :: "cfg ⇒ 'h::bot ⇒ pp ⇒ 'h" where
  "rhs_init_global g g₀ v ≡ (if v = cfg_entry g then g₀ else bot)"

definition routed_node_rhs ::
  "(cfg ⇒ pp ⇒ 'c ⇒ ((pp × 'c + 'k) × edge_action) list)
   ⇒ (cfg ⇒ pp ⇒ (pp × call_action) list)
   ⇒ ('c ⇒ 'k)
   ⇒ (pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c)
   ⇒ ('c ⇒ pp × 'c + 'k ⇒ edge_action
        ⇒ (pp × 'c, 'k, ('d::bounded_semilattice_sup_bot,
                         'h::bounded_semilattice_sup_bot) dg_state,
                        ('d, 'h) dg_state) strategy_program)
   ⇒ ((pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ call_action ⇒ pp ⇒ pp
        ⇒ (pp × 'c, 'k, ('d, 'h) dg_state, ('d, 'h) dg_state) strategy_program)
   ⇒ ((pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ pp
        ⇒ (pp × 'c, 'k, ('d, 'h) dg_state, ('d, 'h) dg_state) strategy_program list)
   ⇒ cfg ⇒ 'd ⇒ 'd ⇒ 'h
   ⇒ (pp × 'c, 'k, ('d, 'h) dg_state) eqsT"
where
  "routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ =
     (λ(v, c).
        let t = sp_compile (side_rhs_fold_dg (rhs_init g bot0 d₀ v)
                  (routed_contribution_programs pred_sel site_sel route it cmb extra g c v))
        in if v = cfg_entry g then Side (buffer_key_at c) (DG bot g₀) t else t)"


lemma eq_routed_node_rhs:
  assumes wf: "⋀w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                     extra g ctx w). sp_wf p"
  shows
  "eq (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀)
      (v, ctx) τ =
   DG (side_acc_dg (rhs_init g bot0 d₀ v) τ
     (routed_contribution_programs pred_sel site_sel route it cmb extra g ctx v)) bot"
  by (simp add: routed_node_rhs_def Let_def
        traverse_side_rhs_fold_dg[OF wf])

lemma sides_routed_node_rhs:
  assumes wf: "⋀w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                     extra g ctx w). sp_wf p"
  shows
  "sides_of_rhs (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀
      (v, ctx)) τ (Inr (buffer_key_at ctx)) =
   (if v = cfg_entry g then DG bot g₀ else bot)
   ⊔ (⨆t←routed_contribution_programs pred_sel site_sel route it cmb extra g ctx v.
```

```
       sides_of_program t τ (Inr (buffer_key_at ctx)))"
  by (simp add: routed_node_rhs_def Let_def
        sides_of_program_side_rhs_fold_dg_char[OF wf]
        bot_dg_state_def[symmetric] ac_simps)
```

The right-hand side as a join of its parts

At a fixed valuation the answer of `routed_node_rhs` is a join: the node's initial value, one value per intra edge entering it, one per call returning at it, and the values of the routing protocol's own programs, the seed read among them. `rhs_init` names the first part in the generator itself; the definitions below name the other three, and `routed_node_rhs_parts` states the answer as the join of all four. Every part is the local half of a contribution's answer at the valuation; publications do not enter the answer and are characterized by `sides_routed_node_rhs`.

```
definition rhs_edge where
  "rhs_edge it τ ctx u a = dg_local (traverse_program (it ctx u a) τ)"

definition rhs_call where
  "rhs_call cmb route τ ctx u ca v = dg_local (traverse_program (cmb route ctx ca u v) τ)"

definition rhs_seed where
  "rhs_seed extra route τ ctx v = (⨆p←extra route ctx v. dg_local (traverse_program p τ))"

lemma routed_node_rhs_parts:
  assumes wf: "⋀w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                    extra g c w). sp_wf p"
  shows
  "eq (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀)
      (v, c) τ =
   DG (rhs_init g bot0 d₀ v
       ⊔ (⨆(u, a)←pred_sel g v c. rhs_edge it τ c u a)
       ⊔ (⨆(u, ca)←site_sel g v. rhs_call cmb route τ c u ca v)
       ⊔ rhs_seed extra route τ c v) bot"
  by (simp add: eq_routed_node_rhs[OF wf] side_acc_dg_as_foldr routed_contribution_programs_def
        rhs_edge_def rhs_call_def rhs_seed_def split_def comp_def sup_assoc)
```

Buffered generator: fold Side-free contributions, publish once

`routed_node_rhs` writes a node's own key once per contribution that publishes there: several intra predecessors or several return call actions can each independently emit a `Side (buffer_key_at c) ...`, so several writes to the same `buffer_key_at c` land in one RHS evaluation. `routed_node_rhs_buffered` folds Side-free contribution programs (the caller's own `it_c` for `intra` and `cmb_c` for `comb`) with `fold_rhs_program` and publishes `buffer_key_at c` once, after every contribution has been read.

That fold shapes the node's *own* key, but each call returning at the node still publishes its callee's seed from inside its `cmb_c` program. Two call sites resuming at the same node, as the two branches of a conditional that both end in a call do, can therefore name one seed key twice. `buffer_sides` accumulates the writes per key and preserves the declarative `traverse_rhs`/`sides_of_rhs` value wherever it flushes, so the correspondence with the unbuffered generator below is unaffected by the choice of flush points.

The choice does decide when a seed reaches the solver. A call program publishes the seed and then reads the callee's exit, and that read is what makes the solver evaluate the callee. Held back to the answer, the seed arrives after the callee has been solved from ⊥, and the seed's change then sends the caller round again. So a node at which at most one call returns flushes before every local read, and the seed is published before the exit is read, in the order of Goblint's `sidel` and `getl`. A node at which several calls return flushes at the answer only: a flush before the first call's exit read would write a seed the second call may write again, and an update rule that keeps one slot per origin would see a partial contribution first in every evaluation. There `distinct_side_path_buffer_sides_answers_only` bounds the writes unconditionally; at the other nodes the bound `distinct_side_path_buffer_sides` holds as long as one call program does not write a seed again after reading an exit, which the shipped specifications' single-alternative entry transfers ensure.

```
definition routed_node_rhs_buffered ::
  "(cfg ⇒ pp ⇒ 'c ⇒ ((pp × 'c + 'k) × edge_action) list)
   ⇒ (cfg ⇒ pp ⇒ (pp × call_action) list)
   ⇒ ('c ⇒ 'k)
   ⇒ (pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c)
   ⇒ ('c ⇒ pp × 'c + 'k ⇒ edge_action
        ⇒ (pp × 'c, 'k, ('d::bounded_semilattice_sup_bot,
                         'h::bounded_semilattice_sup_bot) dg_state,
                        ('d, 'h) dg_state) strategy_program)
   ⇒ ((pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ call_action ⇒ pp ⇒ pp
        ⇒ (pp × 'c, 'k, ('d, 'h) dg_state, ('d, 'h) dg_state) strategy_program)
   ⇒ ((pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ pp
```

```
      ⇒ (pp × 'c, 'k, ('d, 'h) dg_state, ('d, 'h) dg_state) strategy_program list)
  ⇒ cfg ⇒ 'd ⇒ 'd ⇒ 'h
  ⇒ (pp × 'c, 'k, ('d, 'h) dg_state) eqsT"
where
  "routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀ =
     (λ(v, c).
        let acc0 = DG (rhs_init g bot0 d₀ v) (rhs_init_global g g₀ v);
            t = sp_compile (fold_rhs_program acc0
                  (routed_contribution_programs pred_sel site_sel route it_c cmb_c extra g c v))
        in buffer_sides (λ_. length (site_sel g v) ≤ 1) (sp_lift_tree t (λres.
          Side (buffer_key_at c) (DG bot (dg_global res)) (Answer (DG (dg_local res) bot)))))"
```

`fold_rhs_program` threads its accumulator in evaluation order (a left fold); the lemmas below need the equivalent right fold, since that is how each of their own goals is already phrased. `traverse_rhs` only ever joins, so the two forms agree by `foldr_sup_seed_swap`.

```
lemma eq_routed_node_rhs_buffered:
  assumes wf: "⋀w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it_c cmb_c
                    extra g ctx w). sp_wf p"
  shows
  "eq (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀)
      (v, ctx) τ =
   DG (dg_local (foldr (λt acc'. traverse_program t τ ⊔ acc')
     (routed_contribution_programs pred_sel site_sel route it_c cmb_c extra g ctx v)
     (DG (rhs_init g bot0 d₀ v) (rhs_init_global g g₀ v)))) bot"
  by (simp add: routed_node_rhs_buffered_def Let_def
       traverse_fold_rhs_program_char_foldr[OF wf])
```

The buffered generator answers the same join of parts as the reference generator, over its own contribution programs. Its answer carries no global half; the global part of the initial accumulator leaves through the publication at `buffer_key_at c`.

```
lemma routed_node_rhs_buffered_parts:
  assumes wf: "⋀w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                    extra g c w). sp_wf p"
  shows
  "eq (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀)
      (v, c) τ =
   DG (rhs_init g bot0 d₀ v
       ⊔ (⨆(u, a)←pred_sel g v c. rhs_edge it τ c u a)
       ⊔ (⨆(u, ca)←site_sel g v. rhs_call cmb route τ c u ca v)
       ⊔ rhs_seed extra route τ c v) bot"
proof -
  have "eq (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀)
      (v, c) τ
    = eq (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀) (v, c) τ"
    by (simp add: eq_routed_node_rhs_buffered[OF wf] eq_routed_node_rhs[OF wf]
        dg_local_foldr_generic side_acc_dg_as_foldr_seeded)
  then show ?thesis
    by (simp only: routed_node_rhs_parts[OF wf])
qed

lemma sides_routed_node_rhs_buffered:
  assumes wf: "⋀w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it_c cmb_c
                    extra g ctx w). sp_wf p"
    and intra_free_at_key:
      "⋀c' src a σ. sides_of_program (it_c c' src a) σ (Inr (buffer_key_at c')) = bot"
    and comb_free_at_key: "⋀c' ca cc ex σ.
                    sides_of_program (cmb_c route c' ca cc ex) σ (Inr (buffer_key_at c')) = bot"
    and extra_free: "⋀c' w σ z x. x ∈ set (extra route c' w) ⟹ sides_of_program x σ z = bot"
  shows "sides_of_rhs (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g
      bot0 d₀ g₀ (v, ctx)) τ (Inr (buffer_key_at ctx)) =
   DG bot (dg_global (foldr (λt acc'. traverse_program t τ ⊔ acc')
     (routed_contribution_programs pred_sel site_sel route it_c cmb_c extra g ctx v)
     (DG (rhs_init g bot0 d₀ v) (rhs_init_global g g₀ v))))"
proof -
  have free: "⋀w σ x.
      x ∈ set (routed_contribution_programs pred_sel site_sel route it_c cmb_c extra g ctx w)
      ⟹ sides_of_program x σ (Inr (buffer_key_at ctx)) = bot"
    using intra_free_at_key comb_free_at_key extra_free
    by (blast elim: routed_contribution_programsE)
  show ?thesis
  proof (cases "v = cfg_entry g")
    case True
    have z0: "(sides_of_program (fold_rhs_program (DG (bot0 ⊔ d₀) g₀)
        (routed_contribution_programs pred_sel site_sel route it_c cmb_c extra g ctx
```

```
          (cfg_entry g))) τ) (Inr (buffer_key_at ctx)) = bot"
      by (simp only: sides_of_program_fold_rhs_program_char[OF wf]
          foldr_sup_bot_of_all_bot[OF free[where w = "cfg_entry g"]])
    from True show ?thesis
      by (simp add: routed_node_rhs_buffered_def Let_def
          traverse_fold_rhs_program_char_foldr[OF wf]
          bot_dg_state_def sup_dg_state_def z0)
  next
    case False
    have z0: "(sides_of_program (fold_rhs_program (DG bot0 bot)
        (routed_contribution_programs pred_sel site_sel route it_c cmb_c extra g ctx v)) τ) (Inr (buffer_key_at ctx))
= bot"
      by (simp only: sides_of_program_fold_rhs_program_char[OF wf]
          foldr_sup_bot_of_all_bot[OF free[where w = v]])
    from False show ?thesis
      by (simp add: routed_node_rhs_buffered_def Let_def
          traverse_fold_rhs_program_char_foldr[OF wf]
          bot_dg_state_def sup_dg_state_def z0)
  qed
qed
```

Correspondence: the buffered generator against the direct one

A pointwise correspondence between two parallel contribution lists – one Side-free (feeding `fold_rhs_program`), one Side-emitting (feeding `side_rhs_fold_dg`) – lifts to their whole-list fold, in both the local-answer and global-side components. This is the list-level engine behind `routed_node_rhs_buffered_correspondence`: each of `intra`, `comb`, and `extra` instantiates it once, and `list_all2_appendI` combines the three.

`routed_node_rhs_buffered`'s declarative value matches `routed_node_rhs`'s, given that each buffered hook is the Side-free contribution analogue of its unbuffered counterpart at the SAME `buffer_key_at ctx` slot. `intra` and `comb` carry the same six hypotheses – pairwise local/global/dep agreement, buffered-side freeness at the key, side purity, and off-key agreement – and `extra` is Side-free and answers only its local slot (`extra_free`/`extra_local_only` – `routed_entry_seed_programs` discharges both by direct inspection, since it answers on its `dg_local` half and issues no `Side`). A local-only intra hook discharges its six trivially: its program publishes nothing, so the contribution analogue is the program itself and every hypothesis is reflexive or `bot`; an intra hook that genuinely publishes at `buffer_key_at ctx` supplies its reshaped contribution program, exactly as a reshaped `routed_call_program` does for `cmb`.

```
lemma routed_node_rhs_buffered_correspondence:
  fixes cmb cmb_c ::
    "(pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ call_action ⇒ pp ⇒ pp
       ⇒ (pp × 'c, 'k, ('d::bounded_semilattice_sup_bot,
                        'h::bounded_semilattice_sup_bot) dg_state,
           ('d, 'h) dg_state) strategy_program"
  assumes wf_c: "⋀w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it_c cmb_c
                      extra g ctx w). sp_wf p"
    and wf: "⋀w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                 extra g ctx w). sp_wf p"
    and intra_t: "⋀c' src a τ. dg_local (traverse_program (it_c c' src a) τ)
                    = dg_local (traverse_program (it c' src a) τ)"
    and intra_side_pure: "⋀c' src a τ.
                          dg_local (sides_of_program (it c' src a) τ (Inr (buffer_key_at c'))) = bot"
    and intra_s: "⋀c' src a τ. dg_global (traverse_program (it_c c' src a) τ)
                    = dg_global (sides_of_program (it c' src a) τ (Inr (buffer_key_at c')))"
    and intra_free_at_key:
      "⋀c' src a τ. sides_of_program (it_c c' src a) τ (Inr (buffer_key_at c')) = bot"
    and intra_sides_off_key: "⋀c' src a τ z. z ≠ Inr (buffer_key_at c')
                    ⟹ sides_of_program (it_c c' src a) τ z = sides_of_program (it c' src a) τ z"
    and intra_dep: "⋀c' src a τ. dep_program τ (it_c c' src a) = dep_program τ (it c' src a)"
    and comb_t: "⋀c' ca cc ex τ. dg_local (traverse_program (cmb_c route c' ca cc ex) τ)
                    = dg_local (traverse_program (cmb route c' ca cc ex) τ)"
    and comb_side_pure: "⋀c' ca cc ex τ.
                    dg_local (sides_of_program (cmb route c' ca cc ex) τ (Inr (buffer_key_at c'))) = bot"
    and comb_s: "⋀c' ca cc ex τ. dg_global (traverse_program (cmb_c route c' ca cc ex) τ)
                    = dg_global (sides_of_program (cmb route c' ca cc ex) τ (Inr (buffer_key_at c')))"
    and comb_free_at_key: "⋀c' ca cc ex τ.
                    sides_of_program (cmb_c route c' ca cc ex) τ (Inr (buffer_key_at c')) = bot"
    and comb_sides_off_key: "⋀c' ca cc ex τ z. z ≠ Inr (buffer_key_at c')
                    ⟹ sides_of_program (cmb_c route c' ca cc ex) τ z
                        = sides_of_program (cmb route c' ca cc ex) τ z"
    and comb_dep: "⋀c' ca cc ex τ. dep_program τ (cmb_c route c' ca cc ex)
                    = dep_program τ (cmb route c' ca cc ex)"
    and extra_free: "⋀c' w τ z x. x ∈ set (extra route c' w) ⟹ sides_of_program x τ z = bot"
    and extra_local_only: "⋀c' w τ x.
```

```
                          x ∈ set (extra route c' w) ⟹ dg_global (traverse_program x τ) = bot"
  shows "traverse_rhs
           (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀
             (v, ctx)) τ
         = traverse_rhs
           (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ"
    (is ?T)
    and "dep_aux τ
           (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀
             (v, ctx))
         = dep_aux τ
           (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx))"
    (is ?D)
    and "sides_of_rhs
           (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀
             (v, ctx)) τ
         = sides_of_rhs
           (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ"
    (is ?S)
proof -
  let ?intra_new = "map (λ(src, a). it_c ctx src a) (pred_sel g v ctx)"
  let ?intra_old = "map (λ(src, a). it ctx src a) (pred_sel g v ctx)"
  let ?comb_new = "map (λ(cc, ca). cmb_c route ctx ca cc v) (site_sel g v)"
  let ?comb_old = "map (λ(cc, ca). cmb route ctx ca cc v) (site_sel g v)"
  let ?extra = "extra route ctx v"
  let ?acc0 = "DG (rhs_init g bot0 d₀ v) (rhs_init_global g g₀ v)"
  let ?new = "routed_contribution_programs pred_sel site_sel route it_c cmb_c extra g ctx v"
  let ?old = "routed_contribution_programs pred_sel site_sel route it cmb extra g ctx v"
  have new_split: "?new = ?intra_new @ ?comb_new @ ?extra"
    by (simp add: routed_contribution_programs_def)
  have old_split: "?old = ?intra_old @ ?comb_old @ ?extra"
    by (simp add: routed_contribution_programs_def)
  have intra_local: "list_all2
      (λt_new t_old. dg_local (traverse_program t_new τ) = dg_local (traverse_program t_old τ))
      ?intra_new ?intra_old"
    by (rule list_all2_map_diag) (auto simp: intra_t)
  have intra_global: "list_all2
      (λt_new t_old. dg_global (traverse_program t_new τ)
                       = dg_global (sides_of_program t_old τ (Inr (buffer_key_at ctx))))
      ?intra_new ?intra_old"
    by (rule list_all2_map_diag) (auto simp: intra_s)
  have comb_local: "list_all2
      (λt_new t_old. dg_local (traverse_program t_new τ) = dg_local (traverse_program t_old τ))
      ?comb_new ?comb_old"
    by (rule list_all2_map_diag) (auto simp: comb_t)
  have comb_global: "list_all2
      (λt_new t_old. dg_global (traverse_program t_new τ)
                       = dg_global (sides_of_program t_old τ (Inr (buffer_key_at ctx))))
      ?comb_new ?comb_old"
    by (rule list_all2_map_diag) (auto simp: comb_s)
  have extra_local: "list_all2
      (λt_new t_old. dg_local (traverse_program t_new τ) = dg_local (traverse_program t_old τ))
      ?extra ?extra"
    by (rule list.rel_refl_strong) simp
  have extra_global: "list_all2
      (λt_new t_old. dg_global (traverse_program t_new τ)
                       = dg_global (sides_of_program t_old τ (Inr (buffer_key_at ctx))))
      ?extra ?extra"
    by (rule list.rel_refl_strong) (simp add: extra_free extra_local_only bot_dg_state_def)
  have local_list: "list_all2
      (λt_new t_old. dg_local (traverse_program t_new τ) = dg_local (traverse_program t_old τ))
      ?new ?old"
    unfolding new_split old_split
    using intra_local comb_local extra_local by (intro list_all2_appendI)
  have global_list: "list_all2 (λt_new t_old. dg_global (traverse_program t_new τ)
                                  = dg_global (sides_of_program t_old τ (Inr (buffer_key_at ctx))))
      ?new ?old"
    unfolding new_split old_split
    using intra_global comb_global extra_global by (intro list_all2_appendI)
  show ?T
  proof -
    have "traverse_rhs
        (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀
          (v, ctx)) τ
       = DG (dg_local (foldr (λt acc'. traverse_program t τ ⊔ acc') ?new ?acc0)) bot"
      by (simp add: eq_routed_node_rhs_buffered[OF wf_c] routed_contribution_programs_def)
```

```
  also have "... = DG (dg_local ?acc0 ⊔ (⨆t←?new. dg_local (traverse_program t τ))) bot"
    by (simp add: dg_local_foldr_generic foldr_join_seed_out dg_local_join_list)
  also have "... = DG (dg_local ?acc0 ⊔ (⨆t←?old. dg_local (traverse_program t τ))) bot"
    by (simp only: foldr_sup_list_all2_cong[OF local_list])
  also have "... = DG (side_acc_dg (dg_local ?acc0) τ ?old) bot"
    by (simp add: side_acc_dg_as_foldr)
  also have "... = traverse_rhs
      (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ"
    by (simp add: eq_routed_node_rhs[OF wf] routed_contribution_programs_def)
  finally show ?thesis .
qed
have global_new: "sides_of_rhs
    (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀
      (v, ctx)) τ (Inr (buffer_key_at ctx))
    = DG bot (dg_global (foldr (λt acc'. traverse_program t τ ⊔ acc') ?new ?acc0))"
  unfolding routed_contribution_programs_def[symmetric]
  by (intro sides_routed_node_rhs_buffered[OF wf_c])
     (use intra_free_at_key comb_free_at_key extra_free in blast)+
have dg_global_new_char: "dg_global (foldr (λt acc'. traverse_program t τ ⊔ acc') ?new ?acc0)
    = dg_global ?acc0
      ⊔ (⨆t←?old. dg_global (sides_of_program t τ (Inr (buffer_key_at ctx))))"
proof -
  have "dg_global (foldr (λt acc'. traverse_program t τ ⊔ acc') ?new ?acc0)
      = dg_global ?acc0 ⊔ (⨆t←?new. dg_global (traverse_program t τ))"
    by (simp add: dg_global_foldr_generic foldr_join_seed_out dg_global_join_list)
  also have "... = dg_global ?acc0
      ⊔ (⨆t←?old. dg_global (sides_of_program t τ (Inr (buffer_key_at ctx))))"
    by (simp only: foldr_sup_list_all2_cong[OF global_list])
  finally show ?thesis .
qed
have old_elem_side_pure: "⋀t. t ∈ set ?old
    ⟹ dg_local (sides_of_program t τ (Inr (buffer_key_at ctx))) = bot"
  using comb_side_pure extra_free
  by (auto simp: intra_side_pure bot_dg_state_def routed_contribution_programs_def
        split: prod.splits)
have old_fold_side_pure:
  "dg_local (⨆t←?old. sides_of_program t τ (Inr (buffer_key_at ctx))) = bot"
  by (rule foldr_sides_dg_local_bot) (rule old_elem_side_pure)
have old_fold_collapse: "(⨆t←?old. sides_of_program t τ (Inr (buffer_key_at ctx)))
    = DG bot (dg_global (⨆t←?old. sides_of_program t τ (Inr (buffer_key_at ctx))))"
  using old_fold_side_pure
  by (cases "⨆t←?old. sides_of_program t τ (Inr (buffer_key_at ctx))") simp
have old_sides: "sides_of_rhs
    (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ
    (Inr (buffer_key_at ctx))
    = DG bot (dg_global ?acc0
              ⊔ dg_global (⨆t←?old. sides_of_program t τ (Inr (buffer_key_at ctx))))"
proof -
  have "sides_of_rhs
      (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ
      (Inr (buffer_key_at ctx))
      = (if v = cfg_entry g then DG bot g₀ else bot)
        ⊔ (⨆t←?old. sides_of_program t τ (Inr (buffer_key_at ctx)))"
    by (rule sides_routed_node_rhs[OF wf])
  also have "... = DG bot (dg_global ?acc0
        ⊔ dg_global (⨆t←?old. sides_of_program t τ (Inr (buffer_key_at ctx))))"
    by (subst old_fold_collapse)
       (cases "v = cfg_entry g", simp_all add: DG_sup_bot_left bot_dg_state_def)
  finally show ?thesis .
qed
have S_at_key: "sides_of_rhs
    (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀
      (v, ctx)) τ (Inr (buffer_key_at ctx))
    = sides_of_rhs
      (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ
      (Inr (buffer_key_at ctx))"
proof -
  have "sides_of_rhs
      (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀
        (v, ctx)) τ (Inr (buffer_key_at ctx))
      = DG bot (dg_global (foldr (λt acc'. traverse_program t τ ⊔ acc') ?new ?acc0))"
    by (rule global_new)
  also have "... = DG bot (dg_global ?acc0
      ⊔ dg_global (⨆t←?old. sides_of_program t τ (Inr (buffer_key_at ctx))))"
    by (simp only: dg_global_new_char foldr_dg_global_sides_char)
  also have "... = sides_of_rhs
```

```
      (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ
      (Inr (buffer_key_at ctx))"
    by (rule old_sides[symmetric])
  finally show ?thesis .
qed
have intra_dep_list: "list_all2 (λt_new t_old. dep_program τ t_new = dep_program τ t_old)
    ?intra_new ?intra_old"
  by (rule list_all2_map_diag) (auto simp: intra_dep)
have comb_dep_list: "list_all2 (λt_new t_old. dep_program τ t_new = dep_program τ t_old)
    ?comb_new ?comb_old"
  by (rule list_all2_map_diag) (auto simp: comb_dep)
have extra_dep: "list_all2 (λt_new t_old. dep_program τ t_new = dep_program τ t_old)
    ?extra ?extra"
  by (rule list.rel_refl_strong) simp
have dep_list: "list_all2 (λt_new t_old. dep_program τ t_new = dep_program τ t_old)
    ?new ?old"
  unfolding new_split old_split
  using intra_dep_list comb_dep_list extra_dep by (intro list_all2_appendI)
show ?D
proof -
  have "dep_aux τ
      (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀
        (v, ctx))
      = (⋃t∈set ?new. dep_program τ t)"
    by (simp add: routed_node_rhs_buffered_def Let_def
        dep_program_fold_rhs_program_char[OF wf_c])
  also have "... = (⋃t∈set ?old. dep_program τ t)"
    by (rule list_all2_Union_eq[OF dep_list])
  also have "... = dep_aux τ
      (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx))"
    by (simp add: routed_node_rhs_def Let_def
        dep_program_side_rhs_fold_dg_char[OF wf])
  finally show ?thesis .
qed
have intra_sides_off: "⋀z. z ≠ Inr (buffer_key_at ctx)
    ⟹ list_all2 (λt_new t_old. sides_of_program t_new τ z = sides_of_program t_old τ z)
          ?intra_new ?intra_old"
  by (rule list_all2_map_diag) (auto simp: intra_sides_off_key)
have comb_sides_off: "⋀z. z ≠ Inr (buffer_key_at ctx)
    ⟹ list_all2 (λt_new t_old. sides_of_program t_new τ z = sides_of_program t_old τ z)
          ?comb_new ?comb_old"
  by (rule list_all2_map_diag) (auto simp: comb_sides_off_key)
have extra_sides_off: "⋀z.
    list_all2 (λt_new t_old. sides_of_program t_new τ z = sides_of_program t_old τ z)
      ?extra ?extra"
  by (rule list.rel_refl_strong) simp
have sides_list_off: "⋀z. z ≠ Inr (buffer_key_at ctx)
    ⟹ list_all2 (λt_new t_old. sides_of_program t_new τ z = sides_of_program t_old τ z)
        ?new ?old"
  unfolding new_split old_split
  using intra_sides_off comb_sides_off extra_sides_off by (intro list_all2_appendI)
have fold_sides_off: "⋀z. z ≠ Inr (buffer_key_at ctx) ⟹
    (⨆t←?new. sides_of_program t τ z) = (⨆t←?old. sides_of_program t τ z)"
  by (rule foldr_sup_list_all2_cong) (rule sides_list_off)
show ?S
proof (rule ext)
  fix z
  show "sides_of_rhs
      (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀
        (v, ctx)) τ z
    = sides_of_rhs
      (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ z"
  proof (cases "z = Inr (buffer_key_at ctx)")
    case True
    with S_at_key show ?thesis by simp
  next
    case False
    have new_off: "sides_of_rhs
        (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀
          (v, ctx)) τ z
      = (⨆t←?new. sides_of_program t τ z)"
      using False
      by (simp add: routed_node_rhs_buffered_def Let_def
          sides_of_program_fold_rhs_program_char[OF wf_c] bot_dg_state_def[symmetric])
    have old_off: "sides_of_rhs
        (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ z
```

```
      = (⨆t←?old. sides_of_program t τ z)"
      using False
      by (simp add: routed_node_rhs_def Let_def
          sides_of_program_side_rhs_fold_dg_char[OF wf])
    from new_off old_off fold_sides_off[OF False] show ?thesis by simp
  qed
 qed
qed
```

The observational interface `part_post_solution` actually consumes: given the correspondence, a buffered post-solution is also an unbuffered one. Callers only need this fact, never the tree-shape argument behind `routed_node_rhs_buffered` itself – `part_post_solution`'s three conjuncts ($\mathtt{dep}_L$, `eq`, `sides_of_rhs`) are exactly `?D`/`?T`/`?S`, read pointwise at each `u` ∈ `vars`.

```
lemma part_post_solution_routed_node_rhs_buffered:
 fixes cmb cmb_c ::
   "(pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ call_action ⇒ pp ⇒ pp
      ⇒ (pp × 'c, 'k, ('d::bounded_semilattice_sup_bot,
                       'h::bounded_semilattice_sup_bot) dg_state,
          ('d, 'h) dg_state) strategy_program"
 assumes wf_c: "⋀c' w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it_c
                     cmb_c extra g c' w). sp_wf p"
   and wf: "⋀c' w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                   extra g c' w). sp_wf p"
   and intra_t: "⋀c' src a τ. dg_local (traverse_program (it_c c' src a) τ)
                   = dg_local (traverse_program (it c' src a) τ)"
   and intra_side_pure: "⋀c' src a τ.
                     dg_local (sides_of_program (it c' src a) τ (Inr (buffer_key_at c'))) = bot"
   and intra_s: "⋀c' src a τ. dg_global (traverse_program (it_c c' src a) τ)
                   = dg_global (sides_of_program (it c' src a) τ (Inr (buffer_key_at c')))"
   and intra_free_at_key:
     "⋀c' src a τ. sides_of_program (it_c c' src a) τ (Inr (buffer_key_at c')) = bot"
   and intra_sides_off_key: "⋀c' src a τ z. z ≠ Inr (buffer_key_at c')
                   ⟹ sides_of_program (it_c c' src a) τ z = sides_of_program (it c' src a) τ z"
   and intra_dep: "⋀c' src a τ. dep_program τ (it_c c' src a) = dep_program τ (it c' src a)"
   and comb_t: "⋀c' ca cc ex τ. dg_local (traverse_program (cmb_c route c' ca cc ex) τ)
                   = dg_local (traverse_program (cmb route c' ca cc ex) τ)"
   and comb_side_pure: "⋀c' ca cc ex τ.
                   dg_local (sides_of_program (cmb route c' ca cc ex) τ (Inr (buffer_key_at c'))) = bot"
   and comb_s: "⋀c' ca cc ex τ. dg_global (traverse_program (cmb_c route c' ca cc ex) τ)
                   = dg_global (sides_of_program (cmb route c' ca cc ex) τ (Inr (buffer_key_at c')))"
   and comb_free_at_key: "⋀c' ca cc ex τ.
                   sides_of_program (cmb_c route c' ca cc ex) τ (Inr (buffer_key_at c')) = bot"
   and comb_sides_off_key: "⋀c' ca cc ex τ z. z ≠ Inr (buffer_key_at c')
                   ⟹ sides_of_program (cmb_c route c' ca cc ex) τ z
                      = sides_of_program (cmb route c' ca cc ex) τ z"
   and comb_dep: "⋀c' ca cc ex τ. dep_program τ (cmb_c route c' ca cc ex)
                   = dep_program τ (cmb route c' ca cc ex)"
   and extra_free: "⋀c' w τ z x. x ∈ set (extra route c' w) ⟹ sides_of_program x τ z = bot"
   and extra_local_only: "⋀c' w τ x.
                     x ∈ set (extra route c' w) ⟹ dg_global (traverse_program x τ) = bot"
   and pp_buf: "part_post_solution
       (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀)
       x sigma vars"
 shows "part_post_solution
     (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀) x sigma vars"
proof -
 let ?Tbuf =
   "routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it_c cmb_c extra g bot0 d₀ g₀"
 let ?Told =
   "routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀"
 have corr: "⋀u. traverse_rhs (?Tbuf u) sigma = traverse_rhs (?Told u) sigma
     ∧ dep_aux sigma (?Tbuf u) = dep_aux sigma (?Told u)
     ∧ sides_of_rhs (?Tbuf u) sigma = sides_of_rhs (?Told u) sigma"
 proof -
   fix u :: "pp × 'c"
   obtain v ctx where u: "u = (v, ctx)" by (cases u) auto
   show "traverse_rhs (?Tbuf u) sigma = traverse_rhs (?Told u) sigma
       ∧ dep_aux sigma (?Tbuf u) = dep_aux sigma (?Told u)
       ∧ sides_of_rhs (?Tbuf u) sigma = sides_of_rhs (?Told u) sigma"
     unfolding u
     using routed_node_rhs_buffered_correspondence
           [where cmb = cmb and cmb_c = cmb_c and it = it and it_c = it_c
              and route = route and buffer_key_at = buffer_key_at,
            OF wf_c wf
               intra_t intra_side_pure intra_s intra_free_at_key intra_sides_off_key intra_dep
```

```
                comb_t comb_side_pure comb_s comb_free_at_key comb_sides_off_key comb_dep
                extra_free extra_local_only]
    by simp
  qed
  have "part_post_solution ?Tbuf x sigma vars = part_post_solution ?Told x sigma vars"
    by (rule part_post_solution_cong) (use corr in blast)+
  with pp_buf show ?thesis by simp
qed
```

The same correspondence without reshaped hooks, for contribution programs that answer nothing on their global half. Such a program may still publish at `buffer_key_at ctx`: the buffered generator keeps that publication as it is and adds only the seed it hoists, which is the unbuffered generator's entry publication. A specification that splits its state and publishes the global half through its manager is of this kind.

```
lemma routed_node_rhs_buffered_correspondence_answers_local:
  fixes cmb ::
    "(pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ call_action ⇒ pp ⇒ pp
       ⇒ (pp × 'c, 'k, ('d::bounded_semilattice_sup_bot,
                        'h::bounded_semilattice_sup_bot) dg_state,
           ('d, 'h) dg_state) strategy_program"
  assumes wf: "⋀w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                 extra g ctx w). sp_wf p"
    and answers_local: "⋀w p τ. p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                 extra g ctx w) ⟹ dg_global (traverse_program p τ) = bot"
  shows "traverse_rhs
           (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀
             (v, ctx)) τ
         = traverse_rhs
           (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ"
    (is ?T)
    and "dep_aux τ
           (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀
             (v, ctx))
         = dep_aux τ
           (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx))"
    (is ?D)
    and "sides_of_rhs
           (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀
             (v, ctx)) τ
         = sides_of_rhs
           (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ"
    (is ?S)
proof -
  let ?ps = "routed_contribution_programs pred_sel site_sel route it cmb extra g ctx v"
  let ?acc0 = "DG (rhs_init g bot0 d₀ v) (rhs_init_global g g₀ v)"
  have glob: "dg_global (foldr (λt acc'. traverse_program t τ ⊔ acc') ?ps ?acc0)
      = dg_global ?acc0"
  proof -
    have "(⨆t←?ps. dg_global (traverse_program t τ)) = bot"
      by (rule foldr_sup_bot_of_all_bot) (rule answers_local)
    then show ?thesis
      by (simp only: dg_global_foldr_generic foldr_join_seed_out[where a = "dg_global ?acc0"]
          sup_bot_right)
  qed
  show ?T
  proof -
    have "traverse_rhs
        (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀
          (v, ctx)) τ
        = DG (foldr (λt acc'. dg_local (traverse_program t τ) ⊔ acc') ?ps (dg_local ?acc0)) bot"
      by (simp add: eq_routed_node_rhs_buffered[OF wf] dg_local_foldr_generic)
    also have "... = DG (side_acc_dg (dg_local ?acc0) τ ?ps) bot"
      by (simp add: side_acc_dg_as_foldr_seeded)
    also have "... = traverse_rhs
        (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ"
      by (simp add: eq_routed_node_rhs[OF wf])
    finally show ?thesis .
  qed
  show ?D
    by (simp add: routed_node_rhs_buffered_def routed_node_rhs_def Let_def
        dep_program_fold_rhs_program_char[OF wf] dep_program_side_rhs_fold_dg_char[OF wf])
  show ?S
  proof (rule ext)
    fix z
    show "sides_of_rhs
        (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀
```

```
        (v, ctx)) τ z
    = sides_of_rhs
      (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ (v, ctx)) τ z"
    using glob
    by (cases "v = cfg_entry g")
       (auto simp: routed_node_rhs_buffered_def routed_node_rhs_def Let_def
          sides_of_program_fold_rhs_program_char[OF wf]
          sides_of_program_side_rhs_fold_dg_char[OF wf]
          traverse_fold_rhs_program_char_foldr[OF wf] sup_commute bot_dg_state_def[symmetric])
  qed
qed

lemma part_post_solution_routed_node_rhs_buffered_answers_local:
  fixes cmb ::
    "(pp ⇒ 'c ⇒ 'd ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ call_action ⇒ pp ⇒ pp
      ⇒ (pp × 'c, 'k, ('d::bounded_semilattice_sup_bot,
                       'h::bounded_semilattice_sup_bot) dg_state,
          ('d, 'h) dg_state) strategy_program"
  assumes wf: "⋀c' w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                    extra g c' w). sp_wf p"
    and answers_local: "⋀c' w p τ. p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                  extra g c' w) ⟹ dg_global (traverse_program p τ) = bot"
    and pp_buf: "part_post_solution
        (routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀)
        x sigma vars"
  shows "part_post_solution
      (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀) x sigma vars"
proof -
  let ?Tbuf =
    "routed_node_rhs_buffered pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀"
  let ?Told =
    "routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀"
  have corr: "⋀u. traverse_rhs (?Tbuf u) sigma = traverse_rhs (?Told u) sigma
      ∧ dep_aux sigma (?Tbuf u) = dep_aux sigma (?Told u)
      ∧ sides_of_rhs (?Tbuf u) sigma = sides_of_rhs (?Told u) sigma"
  proof -
    fix u :: "pp × 'c"
    obtain v ctx where u: "u = (v, ctx)" by (cases u) auto
    show "traverse_rhs (?Tbuf u) sigma = traverse_rhs (?Told u) sigma
        ∧ dep_aux sigma (?Tbuf u) = dep_aux sigma (?Told u)
        ∧ sides_of_rhs (?Tbuf u) sigma = sides_of_rhs (?Told u) sigma"
      unfolding u
      by (intro conjI routed_node_rhs_buffered_correspondence_answers_local[OF wf])
         (erule answers_local)+
  qed
  have "part_post_solution ?Tbuf x sigma vars = part_post_solution ?Told x sigma vars"
    by (rule part_post_solution_cong) (use corr in blast)+
  with pp_buf show ?thesis by simp
qed
```

Threefold monotonicity for an arbitrary generator instance

The three `TD_side_mono` preconditions reduce to a per-program contract on the intra, combine, and extra hooks, discharged once here and reusable at every context policy — a routed context policy is then a second interpretation of this reduction, not a second monotonicity proof. The outer `Side` wrapper at `cfg_entry g` is invisible to `traverse_rhs` and `dep_aux` (a `Side` node only ever repackages, never queries); it only has to be threaded through the `sides_of_rhs` case via `fun_upd_sup_mono`, below.

```
lemma routed_node_rhs_mono_eq:
  fixes g :: cfg
  assumes wf: "⋀c w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                    extra g c w). sp_wf p"
  assumes intra_mono: "⋀v c src a s1 s2. (src, a) ∈ set (pred_sel g v c) ⟹ s1 ≤ s2 ⟹
      traverse_program (it c src a) s1 ≤ traverse_program (it c src a) s2"
  assumes comb_mono: "⋀v c cc ca s1 s2. (cc, ca) ∈ set (site_sel g v) ⟹ s1 ≤ s2 ⟹
      traverse_program (cmb route c ca cc v) s1 ≤ traverse_program (cmb route c ca cc v) s2"
  assumes extra_mono: "⋀v c t s1 s2. t ∈ set (extra route c v) ⟹ s1 ≤ s2 ⟹
      traverse_program t s1 ≤ traverse_program t s2"
  shows
    "is_mono_eq (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀)"
proof (unfold is_mono_eq_def, intro allI)
  fix x s1 s2
  show "s1 ≤ s2 ⟶
      eq (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀) x s1
        ≤ eq (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀) x s2"
  proof (intro impI)
```

```
    assume le: "s1 ≤ s2"
    obtain v c where x: "x = (v, c)" by (cases x)
    have tree_mono:
        "∀t ∈ set (routed_contribution_programs pred_sel site_sel route it cmb extra g c v).
                        ∀s1 s2. s1 ≤ s2 ⟶ traverse_program t s1 ≤ traverse_program t s2"
      using intra_mono comb_mono extra_mono
      by (auto simp: routed_contribution_programs_def)
    show "eq (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀) x s1
          ≤ eq (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀) x s2"
      unfolding x eq_routed_node_rhs[OF wf]
      using side_rhs_fold_dg_mono[OF wf tree_mono le]
      by (simp add: traverse_side_rhs_fold_dg[OF wf])
  qed
qed
```

A generic fact about pointwise-ordered functions, function update, and join — nothing here mentions a strategy tree. It has exactly one use, in the publication case below, so it stays beside that call site rather than in generic infrastructure.

```
lemma fun_upd_sup_mono:
  fixes m1 m2 :: "'b ⇒ 'a::bounded_semilattice_sup_bot"
  assumes "m1 ≤ m2"
  shows "m1(y := m1 y ⊔ cd) ≤ m2(y := m2 y ⊔ cd)"
proof -
  have eq: "⋀m. m(y := m y ⊔ cd) = m ⊔ ((λ_. bot)(y := cd))"
    unfolding fun_upd_def sup_fun_def by (rule ext) simp
  show ?thesis unfolding eq by (rule sup_mono[OF assms order_refl])
qed

lemma routed_node_rhs_mono_sides:
  fixes g :: cfg
  assumes wf: "⋀c w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                    extra g c w). sp_wf p"
  assumes intra_sides_mono: "⋀v c src a s1 s2. (src, a) ∈ set (pred_sel g v c) ⟹ s1 ≤ s2 ⟹
      sides_of_program (it c src a) s1 ≤ sides_of_program (it c src a) s2"
  assumes comb_sides_mono: "⋀v c cc ca s1 s2. (cc, ca) ∈ set (site_sel g v) ⟹ s1 ≤ s2 ⟹
      sides_of_program (cmb route c ca cc v) s1 ≤ sides_of_program (cmb route c ca cc v) s2"
  assumes extra_sides_mono: "⋀v c t s1 s2. t ∈ set (extra route c v) ⟹ s1 ≤ s2 ⟹
      sides_of_program t s1 ≤ sides_of_program t s2"
  shows
    "mono_sides (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀)"
proof (unfold mono_sides_def, intro allI)
  fix x s1 s2
  show "s1 ≤ s2 ⟶
      sides_of_rhs (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ x) s1
        ≤ sides_of_rhs
            (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ x) s2"
  proof (intro impI)
    assume le: "s1 ≤ s2"
    obtain v c where x: "x = (v, c)" by (cases x)
    have tree_sides_mono: "⋀w. ∀t ∈ set
                      (routed_contribution_programs pred_sel site_sel route it cmb extra g c w).
                      ∀s1 s2. s1 ≤ s2 ⟶ sides_of_program t s1 ≤ sides_of_program t s2"
      using intra_sides_mono comb_sides_mono extra_sides_mono
      by (auto simp: routed_contribution_programs_def)
    have fold_le: "⋀acc w. sides_of_program (side_rhs_fold_dg acc
                      (routed_contribution_programs pred_sel site_sel route it cmb extra g c w)) s1
                  ≤ sides_of_program (side_rhs_fold_dg acc
                      (routed_contribution_programs pred_sel site_sel route it cmb extra g c w)) s2"
      by (rule side_rhs_fold_dg_sides_mono[OF wf tree_sides_mono le])
    show "sides_of_rhs
            (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ x) s1
          ≤ sides_of_rhs
            (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀ x) s2"
      unfolding x routed_node_rhs_def
      by (simp add: Let_def fold_le fun_upd_sup_mono[OF fold_le] split: if_splits)
  qed
qed

lemma routed_node_rhs_mono_deps:
  fixes g :: cfg
  assumes wf: "⋀c w. ∀p ∈ set (routed_contribution_programs pred_sel site_sel route it cmb
                    extra g c w). sp_wf p"
  assumes intra_deps_mono: "⋀v c src a. (src, a) ∈ set (pred_sel g v c) ⟹
      mono_tree_deps (sp_compile (it c src a))"
  assumes comb_deps_mono: "⋀v c cc ca. (cc, ca) ∈ set (site_sel g v) ⟹
      mono_tree_deps (sp_compile (cmb route c ca cc v))"
```

```
  assumes extra_deps_mono: "⋀v c t. t ∈ set (extra route c v)
      ⟹ mono_tree_deps (sp_compile t)"
  shows
    "mono_deps (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀)"
proof (unfold mono_deps_def, intro allI)
  fix x s1 s2
  show "s1 ≤ s2 ⟶
      dep (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀) s1 x
        ⊆ dep (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀) s2 x"
  proof (intro impI)
    assume ord: "s1 ≤ s2"
    obtain v c where x: "x = (v, c)" by (cases x)
    have tree_deps_mono: "⋀w t.
        t ∈ set (routed_contribution_programs pred_sel site_sel route it cmb extra g c w)
        ⟹ mono_tree_deps (sp_compile t)"
      using intra_deps_mono comb_deps_mono extra_deps_mono
      by (auto simp: routed_contribution_programs_def)
    have fold_mono: "⋀acc w. mono_tree_deps (sp_compile (side_rhs_fold_dg acc
                      (routed_contribution_programs pred_sel site_sel route it cmb extra g c w)))"
      by (rule side_rhs_fold_dg_mono_tree_deps[OF wf]) (rule tree_deps_mono)
    have dsub: "⋀acc w. dep_program s1 (side_rhs_fold_dg acc
                  (routed_contribution_programs pred_sel site_sel route it cmb extra g c w))
                ⊆ dep_program s2 (side_rhs_fold_dg acc
                  (routed_contribution_programs pred_sel site_sel route it cmb extra g c w))"
      using fold_mono[unfolded mono_tree_deps_def] ord by blast
    show "dep (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀) s1 x
          ⊆ dep (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀) s2 x"
      unfolding x dep_def routed_node_rhs_def
      by (simp add: Let_def dsub split: if_splits)
  qed
qed
```

The generator never sees a specification: the intra hook supplies each edge's program, exactly as `cmb` and `extra` already supply theirs, so which unknowns an equation reads and which keys it publishes are properties of the supplied programs, not of the generator. The one key the generator names itself is `buffer_key_at c`, at the CFG entry, where it publishes g_0; a routed activation seed is not its own but arrives through `extra`.

Initial global values as side effects

We represent an initial global value as an actual contribution to its global unknown, so the solved unknown itself contains that value. Reads do not need to join initialization into the stored value implicitly. `with_init init T` provides this for any equation system: the right-hand side of an unknown `x` first publishes every pair of `init x`, then continues as `T x`. An instance that publishes no initial values passes `λ_. []`, and the system is `T`.

```
definition init_sides :: "('g × 'd) list ⇒ ('x, 'g, 'd) strategy_tree ⇒ ('x, 'g, 'd) strategy_tree"
where
  "init_sides xs t = foldr (λ(k, d) t. Side k d t) xs t"

definition with_init :: "('x ⇒ ('g × 'd) list) ⇒ ('x, 'g, 'd) eqsT ⇒ ('x, 'g, 'd) eqsT"
where
  "with_init init T = (λx. init_sides (init x) (T x))"

lemma with_init_Nil [simp]: "with_init (λ_. []) T = T"
  by (simp add: with_init_def init_sides_def)

lemma init_sides_Cons [simp]: "init_sides ((k, d) # xs) t = Side k d (init_sides xs t)"
  by (simp add: init_sides_def)

lemma init_sides_Nil [simp]: "init_sides [] t = t"
  by (simp add: init_sides_def)

lemma traverse_rhs_init_sides [simp]: "traverse_rhs (init_sides xs t) σ = traverse_rhs t σ"
proof (induction xs)
  case (Cons p xs) then show ?case by (cases p) simp
qed simp

lemma dep_aux_init_sides [simp]: "dep_aux σ (init_sides xs t) = dep_aux σ t"
proof (induction xs)
  case (Cons p xs) then show ?case by (cases p) simp
qed simp

lemma sides_of_rhs_init_sides_ge:
  "sides_of_rhs t σ ≤ sides_of_rhs (init_sides xs (t :: ('x, 'g, 'd::bounded_semilattice_sup_bot) strategy_tree)) σ"
proof (induction xs)
```

```
  case (Cons p xs)
  obtain k d where p: "p = (k, d)" by (cases p)
  have "sides_of_rhs (init_sides xs t) σ
          ≤ sides_of_rhs (init_sides (p # xs) t) σ"
    unfolding p le_fun_def by (simp add: Let_def)
  with Cons.IH show ?case by (rule order_trans)
qed simp

lemma sides_of_rhs_init_sides_at:
  "(k, d) ∈ set xs
   ⟹ d ≤ sides_of_rhs (init_sides xs (t :: ('x, 'g, 'd::bounded_semilattice_sup_bot) strategy_tree)) σ (Inr k)"
proof (induction xs)
  case (Cons p xs)
  obtain k' d' where p: "p = (k', d')" by (cases p)
  show ?case
  proof (cases "(k, d) = p")
    case True
    with p show ?thesis by (simp add: Let_def)
  next
    case False
    with Cons.prems have "d ≤ sides_of_rhs (init_sides xs t) σ (Inr k)"
      by (intro Cons.IH) simp
    also have "... ≤ sides_of_rhs (init_sides (p # xs) t) σ (Inr k)"
      unfolding p by (simp add: Let_def)
    finally show ?thesis .
  qed
qed simp
```

A post-solution of the wrapped system bounds the wrapped system's own equations, since the wrapper only adds side effects, and covers every initial value at its key.

```
lemma tree_covered_at_init_sides:
  "tree_covered_at (init_sides xs (t :: ('x, 'g, 'd::bounded_semilattice_sup_bot) strategy_tree)) σ u
   ⟹ tree_covered_at t σ u"
  unfolding tree_covered_at_def
  by simp (meson order_trans sides_of_rhs_init_sides_ge)

lemma dep_with_init [simp]: "dep (with_init init T) σ x = dep T σ x"
  by (simp add: dep_def with_init_def)

lemma post_bounded_with_init_at:
  assumes
    "post_bounded (with_init init (T :: ('x, 'g, 'd::bounded_semilattice_sup_bot) eqsT)) x σ vars"
    and "u ∈ vars" and "(k, d) ∈ set (init u)"
  shows "d ≤ σ (Inr k)"
proof -
  have "sides_of_rhs (init_sides (init u) (T u)) σ ≤ σ"
    using post_boundedD[OF assms(1,2)] by (simp add: with_init_def tree_covered_at_def)
  then have "sides_of_rhs (init_sides (init u) (T u)) σ (Inr k) ≤ σ (Inr k)"
    by (rule le_funD)
  with sides_of_rhs_init_sides_at[OF assms(3)] show ?thesis by (rule order_trans)
qed

lemma part_post_solution_with_init:
  "part_post_solution (with_init init (T :: ('x, 'g, 'd::bounded_semilattice_sup_bot) eqsT)) x σ vars
   ⟹ part_post_solution T x σ vars"
  unfolding part_post_solution_iff_tree_covered_at
proof (intro conjI ballI; elim conjE)
  assume all: "∀u∈vars. dep_L (with_init init T) σ u ⊆ vars
                ∧ tree_covered_at (with_init init T u) σ u"
  fix u assume u: "u ∈ vars"
  show "dep_L T σ u ⊆ vars" using all u by (simp add: dep_L_def)
  show "tree_covered_at (T u) σ u"
    using all u by (simp add: with_init_def) (blast intro: tree_covered_at_init_sides)
qed
end

end
```

6.11 DG_Reader_Transport

```
theory DG_Reader_Transport
  imports Routed_Context_Unit
begin
```

Reading a whole equation system through a pair of readers

A `reader` here maps one carrier to another — typically an implementation value to the mathematical value it stands for. This theory proves, once, that reading commutes with everything an equation is built from: a node's value, the side effects it emits, and the unknowns it queries. The only properties any proof in the chain uses are that a reader preserves `bot` and `(⊔)`, so they are the locale's two assumptions and the development is generic in both carriers. A concrete reader becomes a one-line interpretation.

Dependency commutation for the generator

The generator trees are non-branching (`QueryL` then `QueryG` then `Answer`/`Side`), so the queried-unknown set is structural: independent of the analysis step values and the valuation. Hence dependencies transport verbatim.

Carrier-generic whole-CFG commute

The commute facts below only ever use that a reader preserves `bot` and `(⊔)`; no proof in the chain inspects `default_st_to_fun` or `abs_state` itself. `dg_reader_commute_gen` factors that out: a pair of local/global readers `Floc`/`Fglob` satisfying those two laws, from which every whole-tree and whole-equation-system commute fact in this chain is proved once. The pair of represented functions `dg_state_to_fun` and its reachability-lifted version are both thin instances of the same engine.

Reading a pair componentwise is the datatype's own map, `map_dg_state`; its selector equations join the simplifier.

```
declare dg_state.map_sel [simp]
```

The locale assumes only that both readers preserve `bot` and joins; monotonicity and the commutation with `map_dg_state` follow.

```
locale dg_reader_commute_gen =
  fixes Floc :: "'a::bounded_semilattice_sup_bot ⇒ 'a2::bounded_semilattice_sup_bot"
    and Fglob :: "'b::bounded_semilattice_sup_bot ⇒ 'b2::bounded_semilattice_sup_bot"
  assumes Floc_bot: "Floc bot = bot"
      and Floc_sup: "⋀x y. Floc (x ⊔ y) = Floc x ⊔ Floc y"
      and Fglob_bot: "Fglob bot = bot"
      and Fglob_sup: "⋀x y. Fglob (x ⊔ y) = Fglob x ⊔ Fglob y"
begin

lemma Floc_mono: "x ≤ y ⟹ Floc x ≤ Floc y"
  by (metis Floc_sup le_iff_sup)

lemma Fglob_mono: "x ≤ y ⟹ Fglob x ≤ Fglob y"
  by (metis Fglob_sup le_iff_sup)

lemma map_dg_state_bot [simp]:
  "map_dg_state Floc Fglob (bot :: ('a,'b) dg_state) = bot"
  by (simp add: bot_dg_state_def Floc_bot Fglob_bot)

lemma map_dg_state_sup:
  "map_dg_state Floc Fglob (a ⊔ b :: ('a,'b) dg_state)
     = map_dg_state Floc Fglob a ⊔ map_dg_state Floc Fglob b"
  by (simp add: sup_dg_state_def Floc_sup Fglob_sup)
```

Bundled per-tree transport relation `dg_tree_st_commute` bundles the three facts one strategy tree must transport: its value, its side effects per key, and its dependencies.

```
definition dg_tree_st_commute ::
  "('u + 'k ⇒ ('a,'b) dg_state) ⇒ ('u, 'k, ('a,'b) dg_state) strategy_tree
    ⇒ ('u, 'k, ('a2,'b2) dg_state) strategy_tree ⇒ bool"
where
  "dg_tree_st_commute σ_st t_st t_abs ⟷
     map_dg_state Floc Fglob (traverse_rhs t_st σ_st) =
       traverse_rhs t_abs (map_dg_state Floc Fglob ∘ σ_st) ∧
     (∀k. map_dg_state Floc Fglob (sides_of_rhs t_st σ_st k) =
       sides_of_rhs t_abs (map_dg_state Floc Fglob ∘ σ_st) k) ∧
     dep_aux σ_st t_st =
```

```
      dep_aux (map_dg_state Floc Fglob ∘ σ_st) t_abs"

lemma dg_tree_st_commute_sides:
  "dg_tree_st_commute σ_st t_st t_abs
   ⟹ map_dg_state Floc Fglob (sides_of_rhs t_st σ_st k) =
      sides_of_rhs t_abs (map_dg_state Floc Fglob ∘ σ_st) k"
  by (simp add: dg_tree_st_commute_def)
```

The same relation between two *programs*, bundled with the well-formedness each side needs for the fold's observations to be well posed. Carrying `sp_wf` inside the relation is what lets a list induction over two contribution lists keep it: a `list_all2` of this relation says every pair transports and every element runs its continuation once.

```
definition dg_prog_st_commute ::
  "('u + 'k ⇒ ('a,'b) dg_state)
   ⇒ ('u, 'k, ('a,'b) dg_state, ('a,'b) dg_state) strategy_program
   ⇒ ('u, 'k, ('a2,'b2) dg_state, ('a2,'b2) dg_state) strategy_program ⇒ bool"
where
  "dg_prog_st_commute σ_st p_st p_abs ⟷
     sp_wf p_st ∧ sp_wf p_abs
     ∧ dg_tree_st_commute σ_st (sp_compile p_st) (sp_compile p_abs)"

lemma dg_prog_st_commuteI:
  "⟦sp_wf p_st; sp_wf p_abs;
    dg_tree_st_commute σ_st (sp_compile p_st) (sp_compile p_abs)⟧
   ⟹ dg_prog_st_commute σ_st p_st p_abs"
  by (simp add: dg_prog_st_commute_def)

lemma dg_prog_list_wf_st:
  "list_all2 (dg_prog_st_commute σ_st) ps_st ps_abs
   ⟹ ∀p ∈ set ps_st. sp_wf p"
  by (induction rule: list_all2_induct) (auto simp: dg_prog_st_commute_def)

lemma dg_prog_list_wf_abs:
  "list_all2 (dg_prog_st_commute σ_st) ps_st ps_abs
   ⟹ ∀p ∈ set ps_abs. sp_wf p"
  by (induction rule: list_all2_induct) (auto simp: dg_prog_st_commute_def)

lemma dg_list_commute_trav:
  "list_all2 (dg_prog_st_commute σ_st) ps_st ps_abs
   ⟹ list_all2
      (λp_st p_abs. map_dg_state Floc Fglob (traverse_program p_st σ_st) =
        traverse_program p_abs (map_dg_state Floc Fglob ∘ σ_st))
      ps_st ps_abs"
  by (erule list_all2_mono) (simp add: dg_prog_st_commute_def dg_tree_st_commute_def)

lemma dg_list_commute_travsides:
  "list_all2 (dg_prog_st_commute σ_st) ps_st ps_abs
   ⟹ list_all2
      (λp_st p_abs.
        sp_wf p_st ∧ sp_wf p_abs ∧
        map_dg_state Floc Fglob (traverse_program p_st σ_st) =
          traverse_program p_abs (map_dg_state Floc Fglob ∘ σ_st) ∧
        (∀k. map_dg_state Floc Fglob (sides_of_program p_st σ_st k) =
          sides_of_program p_abs (map_dg_state Floc Fglob ∘ σ_st) k))
      ps_st ps_abs"
  by (erule list_all2_mono) (simp add: dg_prog_st_commute_def dg_tree_st_commute_def)

lemma dg_list_commute_dep:
  "list_all2 (dg_prog_st_commute σ_st) ps_st ps_abs
     ⟹ list_all2 (λp_st p_abs. dep_program σ_st p_st
                    = dep_program (map_dg_state Floc Fglob ∘ σ_st) p_abs) ps_st ps_abs"
  by (erule list_all2_mono) (simp add: dg_prog_st_commute_def dg_tree_st_commute_def)
```

Classifier-parametric fold transport The side-effect folds commute with the readers whenever the folded programs' traversals do, independent of how the fold classifies its contributions.

```
lemma side_acc_dg_commute:
  assumes
    "list_all2
       (λp_st p_abs.
         map_dg_state Floc Fglob (traverse_program p_st σ_st) =
           traverse_program p_abs (map_dg_state Floc Fglob ∘ σ_st))
       ps_st ps_abs"
  shows
    "Floc (side_acc_dg acc_st σ_st ps_st) =
     side_acc_dg (Floc acc_st)
```

```
      (map_dg_state Floc Fglob ∘ σ_st) ps_abs"
  using assms
proof (induction ps_st ps_abs arbitrary: acc_st rule: list_all2_induct)
  case Nil
  thus ?case by simp
next
  case (Cons p_st ps_st p_abs ps_abs)
  have hl: "Floc (dg_local (traverse_program p_st σ_st))
             = dg_local (traverse_program p_abs (map_dg_state Floc Fglob ∘ σ_st))"
    using Cons.hyps(1) by (metis dg_state.map_sel(1))
  have h: "Floc (acc_st ⊔ dg_local (traverse_program p_st σ_st))
         = Floc acc_st ⊔ dg_local (traverse_program p_abs (map_dg_state Floc Fglob ∘ σ_st))"
    by (simp add: Floc_sup hl)
  show ?case
    by (metis (no_types, lifting) Cons.IH h side_acc_dg_simps(2))
qed

lemma sides_side_rhs_fold_dg_commute:
  assumes
    "list_all2
       (λp_st p_abs.
         sp_wf p_st ∧ sp_wf p_abs ∧
         map_dg_state Floc Fglob (traverse_program p_st σ_st) =
           traverse_program p_abs (map_dg_state Floc Fglob ∘ σ_st) ∧
         (∀k. map_dg_state Floc Fglob (sides_of_program p_st σ_st k) =
           sides_of_program p_abs (map_dg_state Floc Fglob ∘ σ_st) k))
       ps_st ps_abs"
  shows
    "map_dg_state Floc Fglob
       (sides_of_program (side_rhs_fold_dg acc_st ps_st) σ_st k) =
     sides_of_program (side_rhs_fold_dg acc_abs ps_abs)
       (map_dg_state Floc Fglob ∘ σ_st) k"
  using assms
proof (induction ps_st ps_abs arbitrary: acc_st acc_abs rule: list_all2_induct)
  case Nil
  thus ?case by (simp add: bot_fun_def)
next
  case (Cons p_st ps_st p_abs ps_abs)
  have sd:
    "map_dg_state Floc Fglob (sides_of_program p_st σ_st k) =
     sides_of_program p_abs (map_dg_state Floc Fglob ∘ σ_st) k"
    using Cons.hyps(1) by simp
  have ih:
    "map_dg_state Floc Fglob
       (sides_of_program
           (side_rhs_fold_dg
             (acc_st ⊔ dg_local (traverse_program p_st σ_st)) ps_st)
         σ_st k)
     = sides_of_program
           (side_rhs_fold_dg
             (acc_abs ⊔
               dg_local (traverse_program p_abs
                 (map_dg_state Floc Fglob ∘ σ_st)))
             ps_abs)
         (map_dg_state Floc Fglob ∘ σ_st) k"
    by (rule Cons.IH)
  have wf: "sp_wf p_st" "sp_wf p_abs" using Cons.hyps(1) by simp_all
  show ?case
    by (simp add: sp_compile_bind sp_wfD[OF wf(1)] sp_wfD[OF wf(2)]
          map_dg_state_sup sd ih comp_def)
qed
```

Routed heterogeneous CALL/COMB transport `routed_call_program`/`routed_entry_seed_programs` (`Voblint_Framewo` are the canonical heterogeneous routing shape: parametric only in a routing function `route` and a seed-key injection `seed`, with the seed payload carried on the `dg_local` half so `'D`/`'G` stay independent. The lemmas below transport `routed_call_program`, so any context-sensitive analysis instantiating `cmb` at that constant discharges CALL/COMB transport once here rather than re-deriving its own tree-commute reasoning.

The specification's own enter and combine appear inside the routed tree as compiled sub-trees, so their transport hypotheses are themselves tree commutes. Sequencing them is what `dg_tree_st_commute_seqcomp` does: a bind commutes when its head commutes and its continuation commutes at the head's answer.

```
lemma dg_tree_st_commute_seqcomp:
  assumes head: "dg_tree_st_commute σ_st t_st t_abs"
    and tail: "dg_tree_st_commute σ_st (k_st (traverse_rhs t_st σ_st))
```

```
              (k_abs (traverse_rhs t_abs (map_dg_state Floc Fglob ∘ σ_st)))"
  shows "dg_tree_st_commute σ_st (sp_lift_tree t_st k_st) (sp_lift_tree t_abs k_abs)"
  using head tail
  unfolding dg_tree_st_commute_def
  by (simp add: map_dg_state_sup)

lemma dg_tree_st_commute_answer:
  "dg_tree_st_commute σ_st (Answer d) (Answer (map_dg_state Floc Fglob d))"
  by (simp add: dg_tree_st_commute_def)
```

A local-only transfer compiles to a single answer, so its commute is the reader equation on the pure function alone – no tree reasoning, and no hypothesis about the global half beyond the reader's own `bot` law.

```
lemma dg_tree_st_commute_local_transfer:
  assumes "Floc (f d) = F (Floc d)"
  shows "dg_tree_st_commute σ_st
           (sp_compile_with (λx. DG x bot) (local_transfer f (mk_dg_man d gk)))
           (sp_compile_with (λx. DG x bot) (local_transfer F (mk_dg_man (Floc d) gk)))"
  by (simp add: dg_tree_st_commute_def local_transfer_def mk_dg_man_def
      dg_state.map_sel bot_dg_state_def assms Floc_bot Fglob_bot)

lemma dg_tree_st_commute_local_combine_transfer:
  assumes "Floc (f d de) = F (Floc d) (Floc de)"
  shows "dg_tree_st_commute σ_st
           (sp_compile_with (λx. DG x bot) (local_combine_transfer f (mk_dg_man d gk) de))
           (sp_compile_with (λx. DG x bot)
              (local_combine_transfer F (mk_dg_man (Floc d) gk) (Floc de)))"
  by (simp add: dg_tree_st_commute_def local_combine_transfer_def mk_dg_man_def
      dg_state.map_sel bot_dg_state_def assms Floc_bot Fglob_bot)

lemma dg_tree_st_commute_QueryL:
  assumes cont: "dg_tree_st_commute σ_st (k_st (σ_st (Inl x)))
                   (k_abs (map_dg_state Floc Fglob (σ_st (Inl x))))"
  shows "dg_tree_st_commute σ_st (QueryL x k_st) (QueryL x k_abs)"
  using cont unfolding dg_tree_st_commute_def by (simp add: comp_def)

lemma dg_tree_st_commute_side_effect:
  assumes cont: "dg_tree_st_commute σ_st t_st t_abs"
  shows "dg_tree_st_commute σ_st (Side y d t_st)
           (Side y (map_dg_state Floc Fglob d) t_abs)"
proof -
  have sides: "⋀k. map_dg_state Floc Fglob (sides_of_rhs (Side y d t_st) σ_st k)
      = sides_of_rhs (Side y (map_dg_state Floc Fglob d) t_abs)
          (map_dg_state Floc Fglob ∘ σ_st) k"
  proof -
    fix k
    show "map_dg_state Floc Fglob (sides_of_rhs (Side y d t_st) σ_st k)
        = sides_of_rhs (Side y (map_dg_state Floc Fglob d) t_abs)
            (map_dg_state Floc Fglob ∘ σ_st) k"
      using dg_tree_st_commute_sides[OF cont]
      by (cases "k = Inr y") (simp_all add: Let_def map_dg_state_sup)
  qed
  show ?thesis
    using cont sides unfolding dg_tree_st_commute_def
    by simp
qed

lemma dg_prog_st_commute_side_rhs_fold_dg:
  assumes la: "list_all2 (dg_prog_st_commute σ_st) ps_st ps_abs"
  shows "dg_prog_st_commute σ_st
           (side_rhs_fold_dg acc_st ps_st)
           (side_rhs_fold_dg (Floc acc_st) ps_abs)"
proof -
  have dep: "(⋃p∈set ps_st. dep_program σ_st p)
      = (⋃p∈set ps_abs. dep_program (map_dg_state Floc Fglob ∘ σ_st) p)"
    using dg_list_commute_dep[OF la]
    by (induction rule: list_all2_induct) (auto simp: comp_def)
  show ?thesis
    unfolding dg_prog_st_commute_def dg_tree_st_commute_def
    by (simp add: sp_wf_side_rhs_fold_dg dg_prog_list_wf_st[OF la] dg_prog_list_wf_abs[OF la]
          traverse_side_rhs_fold_dg[OF dg_prog_list_wf_st[OF la]]
          traverse_side_rhs_fold_dg[OF dg_prog_list_wf_abs[OF la]] Fglob_bot
          dep_program_side_rhs_fold_dg_char[OF dg_prog_list_wf_st[OF la]]
          dep_program_side_rhs_fold_dg_char[OF dg_prog_list_wf_abs[OF la]] dep
          side_acc_dg_commute[OF dg_list_commute_trav[OF la]]
          sides_side_rhs_fold_dg_commute[OF dg_list_commute_travsides[OF la]])
qed
```

What a compiled entry must satisfy to transport. Entry answers a list of caller-continuation/callee-entry pairs, which is not the solver's carrier, so the reader equation cannot be stated on the answer of a compiled tree the way the edge and combine transports are. It is stated on the program instead: run both entries against continuations that already agree on pairs mapped componentwise by the readers, and the two trees agree. This observes the entry program exactly as `enter_runs` does, and for the same reason.

```
definition dg_enter_st_commute ::
  "('u + 'k ⇒ ('a,'b) dg_state)
   ⇒ ('u,'k,('a,'b) dg_state,'a enter_result list) strategy_program
   ⇒ ('u,'k,('a2,'b2) dg_state,'a2 enter_result list) strategy_program ⇒ bool"
where
  "dg_enter_st_commute σ_st T_st T_abs ⟷
     (∀K_st K_abs.
        (∀ps. dg_tree_st_commute σ_st (K_st ps) (K_abs (map (map_prod Floc Floc) ps)))
          ⟶ dg_tree_st_commute σ_st (T_st K_st) (T_abs K_abs))"

lemma dg_enter_st_commuteD:
  assumes "dg_enter_st_commute σ_st T_st T_abs"
    and "⋀ps. dg_tree_st_commute σ_st (K_st ps) (K_abs (map (map_prod Floc Floc) ps))"
  shows "dg_tree_st_commute σ_st (T_st K_st) (T_abs K_abs)"
  using assms unfolding dg_enter_st_commute_def by blast
```

A Base-style entry answers its list outright, so its transport is the reader equation on that list alone.

```
lemma dg_enter_st_commute_local_enter_transfer:
  assumes "map (map_prod Floc Floc) (f d) = F (Floc d)"
  shows "dg_enter_st_commute σ_st
           (local_enter_transfer f (mk_dg_man d gk))
           (local_enter_transfer F (mk_dg_man (Floc d) gk))"
  unfolding dg_enter_st_commute_def local_enter_transfer_def sp_return_def
proof (intro allI impI)
  fix K_st K_abs
  assume K: "∀ps. dg_tree_st_commute σ_st (K_st ps) (K_abs (map (map_prod Floc Floc) ps))"
  show "dg_tree_st_commute σ_st (K_st (f (man_local (mk_dg_man d gk))))
          (K_abs (F (man_local (mk_dg_man (Floc d) gk))))"
    using K[rule_format, of "f d"] assms by simp
qed
```

One call alternative. The bottom branch is a plain combine against ⊥, so it needs nothing beyond `Hcomb`; the other publishes the seed and reads the callee exit back, which is where `Hroute` is used. `Hbot` is what keeps the two carriers on the *same* branch: without it one could take the seed and the other not.

```
lemma dg_prog_st_commute_routed_call_alternative_program:
  assumes wf_st: "dg_spec_wf S_st"
    and wf_abs: "dg_spec_wf S_abs"
    and Hcomb: "⋀ci d de. dg_tree_st_commute σ_st
        (sp_compile_with (λx. DG x bot)
           (dg_spec_combine_transfer S_st ci (mk_dg_man d global_of) de))
        (sp_compile_with (λx. DG x bot)
           (dg_spec_combine_transfer S_abs ci (mk_dg_man (Floc d) global_of) (Floc de)))"
    and Hroute: "⋀u c' d ca'. route_st u c' d ca' = route_abs u c' (Floc d) ca'"
    and Hbot: "⋀d. is_bot_abs (Floc d) = is_bot_st d"
  shows "dg_prog_st_commute σ_st
           (routed_call_alternative_program S_st global_of seed route_st is_bot_st ctx ca cc p alt)
           (routed_call_alternative_program S_abs global_of seed route_abs is_bot_abs ctx ca cc p
              (map_prod Floc Floc alt))"
proof (rule dg_prog_st_commuteI)
  show "sp_wf (routed_call_alternative_program S_st global_of seed route_st is_bot_st
                 ctx ca cc p alt)"
    by (rule sp_wf_routed_call_alternative_program[OF wf_st])
  show "sp_wf (routed_call_alternative_program S_abs global_of seed route_abs is_bot_abs ctx ca cc p
                (map_prod Floc Floc alt))"
    by (rule sp_wf_routed_call_alternative_program[OF wf_abs])
  show "dg_tree_st_commute σ_st
          (sp_compile (routed_call_alternative_program S_st global_of seed route_st is_bot_st
             ctx ca cc p alt))
          (sp_compile (routed_call_alternative_program S_abs global_of seed route_abs is_bot_abs
             ctx ca cc p (map_prod Floc Floc alt)))"
  proof (cases alt)
    case (Pair cont entry)
    show ?thesis
    proof (cases "is_bot_st entry")
      case True
      then have abs: "is_bot_abs (Floc entry)" by (simp add: Hbot)
    have cb: "dg_tree_st_commute σ_st
        (sp_compile_with (λx. DG x bot)
```

```
          (dg_spec_combine_transfer S_st (call_info_of ca p) (mk_dg_man cont global_of) bot))
        (sp_compile_with (λx. DG x bot)
          (dg_spec_combine_transfer S_abs (call_info_of ca p)
            (mk_dg_man (Floc cont) global_of) bot))"
      using Hcomb[of "call_info_of ca p" cont bot] by (simp add: Floc_bot)
      show ?thesis
        unfolding Pair using cb by (simp add: True abs sp_compile_def)
    next
      case False
      then have abs: "¬ is_bot_abs (Floc entry)" by (simp add: Hbot)
      have r: "route_abs cc ctx (Floc entry) ca = route_st cc ctx entry ca"
        by (rule Hroute[symmetric])
    have eq_st: "sp_compile (routed_call_alternative_program
        S_st global_of seed route_st is_bot_st ctx ca cc p (cont, entry))
        = Side (seed (FunctionEntry p) (route_st cc ctx entry ca)) (DG entry bot)
            (QueryL (FunctionResult p, route_st cc ctx entry ca)
              (λcs. sp_compile_with (λx. DG x bot)
                (dg_spec_combine_transfer S_st (call_info_of ca p)
                  (mk_dg_man cont global_of) (dg_local cs))))"
      using False by (simp add: sp_compile_def)
    have eq_abs: "sp_compile (routed_call_alternative_program S_abs global_of seed route_abs
          is_bot_abs ctx ca cc p (Floc cont, Floc entry))
        = Side (seed (FunctionEntry p) (route_st cc ctx entry ca)) (DG (Floc entry) bot)
            (QueryL (FunctionResult p, route_st cc ctx entry ca)
              (λcs. sp_compile_with (λx. DG x bot)
                (dg_spec_combine_transfer S_abs (call_info_of ca p)
                  (mk_dg_man (Floc cont) global_of) (dg_local cs))))"
      using abs by (simp add: r sp_compile_def)
      show ?thesis
        unfolding Pair map_prod_simp fst_conv snd_conv eq_st eq_abs
        by (rule dg_tree_st_commute_side_effect
              [where d = "DG entry bot", simplified dg_state.map_sel dg_state.map Fglob_bot],
            rule dg_tree_st_commute_QueryL,
            simp only: dg_state.map_sel dg_state.map,
            rule Hcomb)
    qed
  qed
qed

lemma dg_prog_st_commute_routed_callee_call_program:
  assumes wf_st: "dg_spec_wf S_st"
    and wf_abs: "dg_spec_wf S_abs"
    and Henter: "⋀ci d. dg_enter_st_commute σ_st
        (enter♯ S_st ci (mk_dg_man d global_of))
        (enter♯ S_abs ci (mk_dg_man (Floc d) global_of))"
    and Hcomb: "⋀ci d de. dg_tree_st_commute σ_st
        (sp_compile_with (λx. DG x bot)
          (dg_spec_combine_transfer S_st ci (mk_dg_man d global_of) de))
        (sp_compile_with (λx. DG x bot)
          (dg_spec_combine_transfer S_abs ci (mk_dg_man (Floc d) global_of) (Floc de)))"
    and Hroute: "⋀u c' d ca'. route_st u c' d ca' = route_abs u c' (Floc d) ca'"
    and Hbot: "⋀d. is_bot_abs (Floc d) = is_bot_st d"
  shows "dg_prog_st_commute σ_st
          (routed_callee_call_program S_st global_of seed route_st is_bot_st ctx ca cc caller p)
          (routed_callee_call_program S_abs global_of seed route_abs is_bot_abs
            ctx ca cc (Floc caller) p)"
  unfolding dg_prog_st_commute_def
proof (intro conjI sp_wf_routed_callee_call_program wf_st wf_abs)
  have alt: "⋀x. dg_prog_st_commute σ_st
      (routed_call_alternative_program S_st global_of seed route_st is_bot_st ctx ca cc p x)
      (routed_call_alternative_program S_abs global_of seed route_abs is_bot_abs ctx ca cc p
        (map_prod Floc Floc x))"
    using wf_st wf_abs Hcomb Hroute Hbot
    by (rule dg_prog_st_commute_routed_call_alternative_program)
  show "dg_tree_st_commute σ_st
      (sp_compile (routed_callee_call_program S_st global_of seed route_st is_bot_st
        ctx ca cc caller p))
      (sp_compile (routed_callee_call_program S_abs global_of seed route_abs is_bot_abs
        ctx ca cc (Floc caller) p))"
    unfolding routed_callee_call_program_def sp_compile_bind
  proof (rule dg_enter_st_commuteD[OF Henter])
    fix ps :: "'a enter_result list"
    have la: "list_all2 (dg_prog_st_commute σ_st)
        (map (routed_call_alternative_program S_st global_of seed route_st is_bot_st ctx ca cc p) ps)
        (map (routed_call_alternative_program S_abs global_of seed route_abs is_bot_abs ctx ca cc p)
          (map (map_prod Floc Floc) ps))"
```

```
      by (simp add: list_all2_conv_all_nth alt)
    show "dg_tree_st_commute σ_st
        (sp_compile (side_rhs_fold_dg bot
          (map (routed_call_alternative_program S_st global_of seed route_st is_bot_st ctx ca cc p)
            ps)))
        (sp_compile (side_rhs_fold_dg bot
          (map (routed_call_alternative_program S_abs global_of seed route_abs is_bot_abs ctx ca cc p)
            (map (map_prod Floc Floc) ps))))"
      using dg_prog_st_commute_side_rhs_fold_dg[OF la, where acc_st = bot]
      by (simp add: Floc_bot dg_prog_st_commute_def)
  qed
qed
```

The call site itself: the resolver must answer the same targets on both carriers, which is the resolution-level twin of `Hroute`. At `static_resolve` that premise is free, since the answer never reads the state.

```
lemma dg_prog_st_commute_routed_call_program:
  assumes wf_st: "dg_spec_wf S_st"
    and wf_abs: "dg_spec_wf S_abs"
    and Henter: "⋀ci d. dg_enter_st_commute σ_st
        (enter♯ S_st ci (mk_dg_man d global_of))
        (enter♯ S_abs ci (mk_dg_man (Floc d) global_of))"
    and Hcomb: "⋀ci d de. dg_tree_st_commute σ_st
        (sp_compile_with (λx. DG x bot)
           (dg_spec_combine_transfer S_st ci (mk_dg_man d global_of) de))
        (sp_compile_with (λx. DG x bot)
           (dg_spec_combine_transfer S_abs ci (mk_dg_man (Floc d) global_of) (Floc de)))"
    and Hroute: "⋀u c' d ca'. route_st u c' d ca' = route_abs u c' (Floc d) ca'"
    and Hbot: "⋀d. is_bot_abs (Floc d) = is_bot_st d"
    and Hresolve: "⋀w cc' ca' d. resolve_st w cc' ca' d = resolve_abs w cc' ca' (Floc d)"
  shows "dg_prog_st_commute σ_st
          (routed_call_program S_st global_of seed resolve_st is_bot_st route_st ctx ca cc v)
          (routed_call_program S_abs global_of seed resolve_abs is_bot_abs route_abs ctx ca cc v)"
  unfolding dg_prog_st_commute_def
proof (intro conjI sp_wf_routed_call_program wf_st wf_abs)
  let ?caller = "dg_local (σ_st (Inl (cc, ctx)))"
  have at: "⋀p. dg_prog_st_commute σ_st
      (routed_callee_call_program S_st global_of seed route_st is_bot_st ctx ca cc ?caller p)
      (routed_callee_call_program S_abs global_of seed route_abs is_bot_abs ctx ca cc
         (Floc ?caller) p)"
    using wf_st wf_abs Henter Hcomb Hroute Hbot
    by (rule dg_prog_st_commute_routed_callee_call_program)
  have la: "list_all2 (dg_prog_st_commute σ_st)
      (map (routed_callee_call_program S_st global_of seed route_st is_bot_st ctx ca cc ?caller)
        (resolve_st v cc ca ?caller))
      (map
        (routed_callee_call_program S_abs global_of seed route_abs is_bot_abs
          ctx ca cc (Floc ?caller))
        (resolve_st v cc ca ?caller))"
    by (simp add: list_all2_conv_all_nth at)
  have body: "dg_tree_st_commute σ_st
      (sp_compile (side_rhs_fold_dg bot
        (map (routed_callee_call_program S_st global_of seed route_st is_bot_st ctx ca cc ?caller)
          (resolve_st v cc ca ?caller))))
      (sp_compile (side_rhs_fold_dg bot
        (map
          (routed_callee_call_program S_abs global_of seed route_abs is_bot_abs
            ctx ca cc (Floc ?caller))
          (resolve_st v cc ca ?caller))))"
    using dg_prog_st_commute_side_rhs_fold_dg[OF la, where acc_st = bot]
    by (simp add: Floc_bot dg_prog_st_commute_def)
  show "dg_tree_st_commute σ_st
      (sp_compile (routed_call_program S_st global_of seed resolve_st is_bot_st route_st ctx ca cc v))
      (sp_compile (routed_call_program S_abs global_of seed resolve_abs is_bot_abs route_abs
         ctx ca cc v))"
    unfolding routed_call_program_def sp_compile_bind
    using body
    by (simp add: dg_tree_st_commute_def sp_wf_observes Hresolve[symmetric] comp_def)
qed

end
end
```

6.12 Activation_Backbone

```
theory Activation_Backbone
  imports "Voblint_Domain.Nonrelational_State" "Voblint_CFG.Activation_Trace_Abstract"
begin
```

What one activation of a procedure may observe

Every concrete call creates one callee activation. Each step of that activation carries the contexts its creating call admits, a nested call derives its callee's contexts from those, and a return resumes the caller at the caller's own. `activation_context_rel` replays those choices along a trace, so a table indexed by `(node, context)` can be checked against concrete runs without carrying an auxiliary digest. That indexing is many-to-one and one-to-many at once: activations reaching the same context are collected together, and one activation may be collected under several contexts. The buckets of a node cover its stores; they need not partition them.

`cover v c` is the set of stores that table admits at node `v` in context `c`. Given the five local obligations of `activation_coverage` on it — `INIT` for the seed stores, `INTRA` per intra edge, `CALL` per call and admitted context, `RETURN` per return, `TOTAL` for at least one admitted context per covered call — `activation_collect_sound` bounds `activation_collect`, the set of stores some valid activation trace can leave at one `(node, context)`. It is the context-sensitive twin of `node_collect_semantic_postfix` and shares its proof shape: interpret `activation_coverage` at the supplied `cover`, then read off `valid_activation_trace_covered_at`.

`adm` says which contexts may describe a concrete call transition, and that is what indexes the collecting semantics. It is handed both stores, so a policy may inspect them, but need not: a call-string policy ignores them, and an entry-state policy is induced by the analyzer's own routing decisions on the abstract entry alternatives that cover them. Proving that a particular policy's `route` induces an `adm` total on the solved table is the routed context locale's job downstream, not this theorem's.

```
theorem activation_collect_sound:
  assumes "activation_coverage g S cover adm c₀ 𝒢"
  shows "𝒜𝒢,adm,c₀,g,S v c ⊆ cover v c"
proof -
  interpret G: activation_coverage g S cover adm c₀ 𝒢 by (fact assms)
  show ?thesis
  proof (rule subsetI)
    fix st assume "st ∈ 𝒜𝒢,adm,c₀,g,S v c"
    then obtain t where t: "t ∈ 𝒯𝒢,g,S"
      and sn: "sink_node t = v" and kc: "activation_context_rel 𝒢 adm c₀ g t c"
      and st: "sink_store t = st"
      by (rule activation_collect_E)
    have "sink_store t ∈ cover (sink_node t) c" using G.valid_activation_trace_covered_at[OF t kc]
      .
    then show "st ∈ cover v c" using sn st by simp
  qed
qed

end
```

6.13 DG_Ctx_Activation

```
theory DG_Ctx_Activation
  imports DG_Spec_Sound DG_Indexed_Generator
    "Voblint_Domain.Nonrelational_State"
begin
```

What a solved routed system says about one program point

A post-solution bounds a right-hand side by the value at its own unknown and every published side effect by the value at its target key. `dg_context_activation` fixes such a solution over the routed generator, a set `vars` of keys whose own equations it bounds, and a reader `sg` that answers the empty set off `vars`. The set may contain every key the solver visited or any subset. These facts derive the EDGE obligation the activation backbone asks for; `Routed_Context` derives COMB for the routed call shape. Entry and callee-seed coverage remain with the concrete analysis.

Both carriers are parameters: γ_{DG} interprets a D/G pair, while γ_M interprets whatever `sg` returns. An analysis whose reader is not an `abs_state` therefore instantiates this locale directly. The generator buffers its writes at `Inr buffer_key`; `global_of` names the key each global name is read from, and `sg_cov` ties the reader at a covered key to γ_{DG} of the local slot against the environment `genv global_of sigma`.

```
locale dg_context_activation = analysis_contract S γDG G
  for S :: "(pp × 'c, 'k, 'v, 'D::bounded_semilattice_sup_bot,
             'G::bounded_semilattice_sup_bot) dg_spec"
    and γDG :: "'D ⇒ ('v ⇒ 'G) ⇒ store set"
    and G :: "vname ⇒ bool" +
  fixes g :: cfg and buffer_key :: 'k and global_of :: "'v ⇒ 'k"
    and route :: "pp ⇒ 'c ⇒ 'D ⇒ call_action ⇒ 'c"
    and cmb :: "(pp ⇒ 'c ⇒ 'D ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ call_action ⇒ pp ⇒ pp
                ⇒ (pp × 'c, 'k, ('D, 'G) dg_state, ('D, 'G) dg_state) strategy_program"
      — one program per call site: the site's own resolver decides which callees it folds
    and extra :: "(pp ⇒ 'c ⇒ 'D ⇒ call_action ⇒ 'c) ⇒ 'c ⇒ pp
                ⇒ (pp × 'c, 'k, ('D, 'G) dg_state, ('D, 'G) dg_state) strategy_program list"
    and bot0 d0 :: 'D and g0 :: 'G
    and sigma :: "pp × 'c + 'k ⇒ ('D, 'G) dg_state"
    and vars :: "(pp × 'c) set" and x0 :: "pp × 'c"
    and sg :: "pp × 'c + 'k ⇒ 'M"
    and γM :: "'M ⇒ store set"
  assumes cmb_wf: "⋀c ca cc v. sp_wf (cmb route c ca cc v)"
    and extra_wf: "⋀c v q. q ∈ set (extra route c v) ⟹ sp_wf q"
    and finE: "finite (intra g)"
    and pp: "post_bounded
               (routed_node_rhs intra_predecessor_addr_list call_site_list (λ_. buffer_key)
                  route (λc src a. dg_spec_edge_program S a src global_of) cmb extra g bot0 d0 g0)
               x0 sigma vars"
    and sg_cov[simp]: "⋀v c. (v, c) ∈ vars
        ⟹ γM (sg (Inl (v, c))) =
          γDG (dg_local (sigma (Inl (v, c)))) (genv global_of sigma)"
    and sg_uncov[simp]: "⋀v c. (v, c) ∉ vars
        ⟹ γM (sg (Inl (v, c))) = {}"
    and fwd: "⋀u a v c. (u, c) ∈ vars
        ⟹ γDG (dg_local (sigma (Inl (u, c)))) (genv global_of sigma) ≠ {}
        ⟹ (u, a, v) ∈ intra g
        ⟹ (v, c) ∈ vars"
begin

abbreviation Gen :: "(pp × 'c, 'k, ('D, 'G) dg_state) eqsT" where
  "Gen ≡ routed_node_rhs intra_predecessor_addr_list call_site_list (λ_. buffer_key)
           route (λc src a. dg_spec_edge_program S a src global_of) cmb extra g bot0 d0 g0"

abbreviation acc0 :: "pp ⇒ 'D" where
  "acc0 v ≡ (if v = cfg_entry g then bot0 ⊔ d0 else bot0)"

abbreviation contribs :: "pp ⇒ 'c
    ⇒ (pp × 'c, 'k, ('D, 'G) dg_state, ('D, 'G) dg_state) strategy_program list" where
  "contribs v ctx ≡
     routed_contribution_programs intra_predecessor_addr_list call_site_list route
       (λc src a. dg_spec_edge_program S a src global_of) cmb extra g ctx v"
```

The reader's claim at a covered point (`sg_cov`) and the manager its call programs run under. Input-only, so facts of theory-level interpretations keep printing the explicit terms.

```
abbreviation (input) gamma_at :: "pp ⇒ 'c ⇒ store set" where
  "gamma_at v ctx ≡ γDG (dg_local (sigma (Inl (v, ctx)))) (genv global_of sigma)"
```

```
abbreviation (input) man_at where
  "man_at v ctx ≡ mk_dg_man (dg_local (sigma (Inl (v, ctx)))) global_of"
```

Every contribution the routed generator folds here runs its continuation once: the edge hook is a compiled specification step, and the call and extra hooks carry their own well-formedness as locale assumptions.

```
lemma contribs_wf [simp]: "∀q ∈ set (contribs v ctx). sp_wf q"
  by (rule routed_contribution_programs_wf)
     (auto intro: sp_wf_dg_spec_edge_program[OF spec_wf] cmb_wf extra_wf)
```

Post-solution elimination

The solver hypothesis, unpacked once into the two bounds every later proof uses: at a covered unknown the generated right-hand side is below the stored local, and its side writes are below the stored globals. Everything downstream cites these instead of `part_post_solution`.

```
lemma pp_eq_bound:
  "(v, ctx) ∈ vars ⟹ eq Gen (v, ctx) sigma ≤ sigma (Inl (v, ctx))"
  by (rule tree_covered_at_local[OF post_boundedD[OF pp]])

lemma pp_sides_bound:
  "(v, ctx) ∈ vars ⟹ sides_of_rhs (Gen (v, ctx)) sigma ≤ sigma"
  by (rule tree_covered_at_sides[OF post_boundedD[OF pp]])

lemma pp_entry_init_global_bound:
  assumes cov: "(cfg_entry g, ctx) ∈ vars"
  shows "g₀ ≤ dg_global (sigma (Inr buffer_key))"
proof -
  have "g₀ ≤ dg_global (sides_of_rhs (Gen (cfg_entry g, ctx)) sigma (Inr buffer_key))"
    unfolding routed_node_rhs_def Let_def
    by (simp add: Let_def sup_dg_state_def)
  also have "... ≤ dg_global (sigma (Inr buffer_key))"
    using pp_sides_bound[OF cov, THEN le_funD, of "Inr buffer_key"]
    by (simp add: less_eq_dg_state_def)
  finally show ?thesis .
qed
```

The local-carrier twin, proved the same way: `Gen`'s entry accumulator starts at `bot0` ⊔ d_0, `side_acc_dg_ge_acc` only grows it, and `pp_eq_bound` transports the bound across a covered point's own equation.

```
lemma pp_entry_init_local_bound:
  assumes cov: "(cfg_entry g, ctx) ∈ vars"
  shows "d₀ ≤ dg_local (sigma (Inl (cfg_entry g, ctx)))"
proof -
  have "d₀ ≤ dg_local (eq Gen (cfg_entry g, ctx) sigma)"
    by (simp add: eq_routed_node_rhs[OF contribs_wf])
       (rule order_trans[OF _ side_acc_dg_ge_acc], simp add: le_supI2)
  also have "... ≤ dg_local (sigma (Inl (cfg_entry g, ctx)))"
    using pp_eq_bound[OF cov] by (simp add: less_eq_dg_state_def)
  finally show ?thesis .
qed
```

The entry Side wrapper only grows the sides

At the entry the generator wraps its fold in one extra `Side` carrying the seed global, so a side read of the wrapped tree dominates the same read of the fold. Bounds proved against the fold therefore transport to `routed_node_rhs` without a separate entry case.

```
lemma sides_fold_le_Gen:
  "(sides_of_program (side_rhs_fold_dg (acc0 v) (contribs v ctx)) sigma) k
   ≤ sides_of_rhs (Gen (v, ctx)) sigma k"
  unfolding routed_node_rhs_def Let_def
  by (cases "v = cfg_entry g") (auto simp: Let_def intro: sup.cobounded1)
```

EDGE: the routed intra bounds and the guarded transport

Every intra predecessor of `v` contributes its edge program to `v`'s own fold, so both bounds below are the post-solution read off that one membership.

```
lemma edge_program_mem_contribs:
  assumes e: "(u, a, v) ∈ intra g"
  shows "dg_spec_edge_program S a (Inl (u, ctx)) global_of ∈ set (contribs v ctx)"
proof -
  have "(Inl (u, ctx), a) ∈ set (intra_predecessor_addr_list g v ctx)"
    using e by (force simp: intra_predecessor_addr_list_def
```

```
      set_intra_predecessor_list[OF finE] intra_predecessors_def)
  thus ?thesis by (force simp: routed_contribution_programs_def)
qed

lemma edge_bound_local:
  assumes cov_v: "(v, ctx) ∈ vars"
    and e: "(u, a, v) ∈ intra g"
  shows "edge_out S a (Inl (u, ctx)) global_of sigma ≤ dg_local (sigma (Inl (v, ctx)))"
proof -
  have "edge_out S a (Inl (u, ctx)) global_of sigma
      ≤ side_acc_dg (acc0 v) sigma (contribs v ctx)"
    using dg_local_traverse_le_side_acc_dg[OF edge_program_mem_contribs[OF e]] .
  also have "... = dg_local (eq Gen (v, ctx) sigma)"
    by (simp add: eq_routed_node_rhs[OF contribs_wf])
  also have "... ≤ dg_local (sigma (Inl (v, ctx)))"
    using pp_eq_bound[OF cov_v] by (simp add: less_eq_dg_state_def)
  finally show ?thesis .
qed

lemma edge_bound_global:
  assumes cov_v: "(v, ctx) ∈ vars"
    and e: "(u, a, v) ∈ intra g"
  shows "dg_global (sides_of_program (dg_spec_edge_program S a (Inl (u, ctx)) global_of) sigma (Inr k))
          ≤ dg_global (sigma (Inr k))"
proof -
  have "dg_global (sides_of_program (dg_spec_edge_program S a (Inl (u, ctx)) global_of) sigma (Inr k))
      ≤ dg_global ((sides_of_program (side_rhs_fold_dg (acc0 v) (contribs v ctx)) sigma) (Inr k))"
    using sides_le_side_rhs_fold_dg[OF contribs_wf edge_program_mem_contribs[OF e], where k =
      "Inr k"]
    by (simp add: less_eq_dg_state_def)
  also have "... ≤ dg_global (sides_of_rhs (Gen (v, ctx)) sigma (Inr k))"
    using sides_fold_le_Gen[where k = "Inr k"]
    by (simp add: less_eq_dg_state_def)
  also have "... ≤ dg_global (sigma (Inr k))"
    using pp_sides_bound[OF cov_v, THEN le_funD, of "Inr k"]
    by (simp add: less_eq_dg_state_def)
  finally show ?thesis .
qed

lemma edge_bound_genv:
  assumes cov_v: "(v, ctx) ∈ vars"
    and e: "(u, a, v) ∈ intra g"
  shows "edge_pub S a (Inl (u, ctx)) global_of sigma ≤ genv global_of sigma"
  unfolding le_fun_def genv_def using edge_bound_global[OF cov_v e] by blast

theorem dg_ctx_act_edge:
  assumes e: "(u, a, v) ∈ intra g"
    and sin: "s ∈ γ_M (sg (Inl (u, ctx)))" and st: "s' ∈ edge_step a s"
  shows "s' ∈ γ_M (sg (Inl (v, ctx)))"
proof (cases "(u, ctx) ∈ vars")
  case False
  hence "γ_M (sg (Inl (u, ctx))) = {}" by (rule sg_uncov)
  thus ?thesis using sin by simp
next
  case True
  let ?d = "dg_local (sigma (Inl (u, ctx)))"
  let ?g = "genv global_of sigma"
  have sin': "s ∈ γ_DG ?d ?g"
    using sin True by simp
  have cov_v: "(v, ctx) ∈ vars"
    using True _ e by (rule fwd) (use sin' in blast)
  have "{s} ⊆ γ_DG ?d ?g" using sin' by simp
  hence "edge_collect a {s} ⊆ edge_collect a (γ_DG ?d ?g)" by (rule edge_collect_mono)
  moreover have "s' ∈ edge_collect a {s}" using st by (simp add: edge_collect_single)
  ultimately have "s' ∈ edge_collect a (γ_DG ?d ?g)" by blast
  hence "s' ∈ γ_DG
      (edge_out S a (Inl (u, ctx)) global_of sigma)
      (?g ⊔ edge_pub S a (Inl (u, ctx)) global_of sigma)"
    using step_sound[of a sigma "Inl (u, ctx)" global_of] by blast
  also have "... ⊆ gamma_at v ctx"
    by (rule gammaDG_mono[OF edge_bound_local[OF cov_v e]
        sup_least[OF order_refl edge_bound_genv[OF cov_v e]]])
  also have "... = γ_M (sg (Inl (v, ctx)))"
    using cov_v by simp
  finally show ?thesis .
qed
```

end

end

6.14 Routed_Call_Programs

```
theory Routed_Call_Programs
  imports DG_Spec_Sound DG_Indexed_Generator
    "Voblint_Domain.Nonrelational_State"
begin
```

The equations one call action generates

What a routed call is, as solver input, before anything is claimed about its soundness. Each callee a call action resolves to becomes one strategy program: it runs the specification's entry transfer on the caller's local value, chooses the callee's context from the entry half of the alternative that produced, publishes that entered state at a global proxy key — an *activation seed*, the only channel by which the entered state reaches the callee's entry unknown — reads the callee's exit back at that same context, and runs the specification's combine against the alternative's continuation half. The publication only joins into the seed and destabilizes its readers; the exit read is the query that makes the solver evaluate the callee at that context. The other half of the protocol is the callee's own entry equation reading its seed slot back: that is `routed_entry_seed_programs`.

This theory is construction and observation only: which unknowns a program reads, which keys it publishes to, and lower bounds on what it contributes at each. Whether those contributions are sound is `Routed_Context`'s question, which fixes the shape built here as `cmb` and `extra` and discharges CALL and COMB against it once.

The routed key space

```
datatype ('v,'c) global_unknown =
  Analysis_Buffer
| Analysis_Global 'v
| Activation_Seed (seed_pp: pp) (seed_ctx: 'c)
```

The solver's global unknowns have three owners. The generator uses `Analysis_Buffer` for a node's buffered global answer. `Analysis_Global` wraps a name of the analysis's own type `'v` — Goblint's `V` — and is the only kind an analysis can reach, through `man_global`/`man_sideg`. `Activation_Seed` holds the entry state published at a callee's activation, keyed by the callee entry point and the routed context.

The remaining constructors separate analysis effects from activation routing. `Analysis_Global` embeds this development's analogue of Goblint's `Spec.V` into the solver's global-key space — the namespace itself is `'v`, and this constructor is the address around it. Goblint's own Base analysis supplies a non-trivial one (privatization globals, thread-return globals) rather than a single global cell. `Activation_Seed` is not Goblint's — their framework records reachable contexts on a separate global whose value is a set of contexts, keyed by function. It is a solver adapter, not the translation of any Goblint constraint constructor: the vendored solver publishes to global keys only, so the one way to hand a callee's entry unknown a value is to publish at a global proxy that the entry equation reads back. Publication does not demand the callee; the caller's read of the callee's exit does. It is named to say so, so that no later reader takes it for a Goblint correspondence.

The types keep the two apart. A transfer names a global as a `'v` and never builds a key, because `mk_dg_man` holds the embedding; only the routing and generator layers apply `Activation_Seed`. Every routed context instantiates `'c` (call strings at `cfg_node list`, entry states at a value list, context-insensitive at `unit` — which is exactly Goblint's own `C = Printable.Unit`), and an analysis with one global takes `'v = unit`.

```
lemma activation_seed_ne_analysis_global [simp]:
  "Activation_Seed u c ≠ Analysis_Global v"
  "Analysis_Global v ≠ Activation_Seed u c"
  by simp_all
```

The canonical routed entry-seed publication and return combine

The routing combine reads the caller under its own context, the callee exit under the context `route` selects from the callee's entered value — the alternative's own second component, after `enter`$^\sharp$ has run — and the exact matched `call_action` (from `return_call_action_list`, never re-derived from the call site's outgoing edges). The spec's own enter and combine transfers run as compiled manager programs over the caller value, so whether the global slot `buffer_key` is read or published at all is the specification's business: a Base-style spec's transfers touch no global and the routed program then carries no `buffer_key` effect either, while an effectful spec's own `man_global`/`man_sideg` calls ride inside the compiled subtrees.

Parameter order matches `dg_context_activation`'s `cmb` calling convention: the generator supplies `route` as `cmb`'s own first argument (`cmb route c ca cc ex` in `routed_node_rhs`), so `routed_call_program S global_of seed`, closing over the spec, the shared slot, and the seed-key injection, is the value that instantiates `cmb`.

This program owns the whole call lifecycle for one call action: it reads the caller once, routes the callee context once (`ctx'`), and shares that one `ctx'` between the entry-seed publication and the combine's callee-exit read.

The publication hands the callee's entry its value, and the exit read, a `QueryL` of the callee's `FunctionResult` node at `ctx'`, is what makes the solver evaluate the callee; both happen in the one equation that combines the result, so `enter#` runs exactly once per call action rather than once per hook. The caller continuation is not reconstructed here: it is handed to `dg_spec_combine_transfer` as the manager's own local value, and that pipeline runs `combine_env`$^\sharp$ and `combine_assign`$^\sharp$ over it, so this layer only chooses addresses and keys. Only the half that cannot move stays with `routed_entry_seed_programs`: a callee's own entry equation is the one place that can read its own seed slot back. `route` is kept as a parameter of that hook purely to match `dg_context_activation`'s `extra` calling convention, which always supplies it.

The seed channel is `'D`-typed, so `'D` and `'G` stay independent. The publication writes the callee's freshly entered local state — a `'D`-typed value — into the `dg_local` half of the `dg_state` published at the seed key. A `dg_state` already carries both fields at every key regardless of whether the key is reached through `Inl` (a `(pp, 'c)` local unknown) or `Inr` (a `'k` global unknown): `dg_local`/`dg_global` are call-site conventions, not something `QueryL`/`QueryG` themselves enforce. Since `seed p ctx` is always distinct from `buffer_key` (`seed_ne_buffer_key`) and no other equation ever publishes to a seed key, nothing else reads or writes that key's `dg_local` half.

One resolved callee: enter the frame through the spec's compiled enter transfer, key the entered state, publish it at the routed seed, read that activation's exit back and run the spec's composed combine. The caller's local state is passed in rather than read here, because the call site reads it once and every target it resolves to is entered from that same value.

What one call alternative contributes. The continuation and the callee-entry value arrive together, from a single run of the specification's entry, and this is where they are used together: the entry half chooses the context and is published at that context's seed, and the continuation half is the local value the return combine runs against. Neither is recoverable from the other, which is why the pair travels this far intact.

Naming it separately is what lets the bounds below speak about *one* alternative: the program the generator builds is a fold over these, so every obligation about a selected alternative is a statement about one element of that fold rather than about the fold itself.

```
definition routed_call_alternative_program ::
  "(pp × 'c, 'k, 'v, 'D::bounded_semilattice_sup_bot,
     'G::bounded_semilattice_sup_bot) dg_spec ⇒ ('v ⇒ 'k)
   ⇒ (pp ⇒ 'c ⇒ 'k)
   ⇒ (pp ⇒ 'c ⇒ 'D ⇒ call_action ⇒ 'c)
   ⇒ ('D ⇒ bool)
   ⇒ 'c ⇒ call_action ⇒ pp ⇒ pname ⇒ 'D enter_result
   ⇒ (pp × 'c, 'k, ('D, 'G) dg_state, ('D, 'G) dg_state) strategy_program"
where
  "routed_call_alternative_program S global_of seed route is_bot ctx ca cc p alt =
     (case alt of (cont, entry) ⇒
        (if is_bot entry
         then sp_map (λd. DG d bot)
                (dg_spec_combine_transfer S (call_info_of ca p)
                  (mk_dg_man cont global_of) bot)
         else
           (let ctx' = route cc ctx entry ca
            in do {
                 _ ← sp_publish (seed (FunctionEntry p) ctx') (DG entry bot);
                 callee_state ← sp_read_local (FunctionResult p, ctx');
                 sp_map (λd. DG d bot)
                   (dg_spec_combine_transfer S (call_info_of ca p)
                     (mk_dg_man cont global_of)
                     (dg_local callee_state))
               })))"
```

Deliberately not a `simp` rule: it expands into a `Side` over a `QueryL` over a compiled combine, and letting that fire everywhere would put a whole call boundary into every goal mentioning the constant. The observation lemmas below cite it where they need it.

```
lemma routed_call_alternative_program_bot [simp]:
  assumes "is_bot entry"
  shows "routed_call_alternative_program S global_of seed route is_bot ctx ca cc p (cont, entry)
           = sp_map (λd. DG d bot)
               (dg_spec_combine_transfer S (call_info_of ca p)
                 (mk_dg_man cont global_of) bot)"
  using assms by (simp add: routed_call_alternative_program_def)

lemma routed_call_alternative_program_nonbot [simp]:
  assumes "¬ is_bot entry"
  shows "routed_call_alternative_program S global_of seed route is_bot ctx ca cc p (cont, entry)
     = do {
         _ ← sp_publish (seed (FunctionEntry p) (route cc ctx entry ca)) (DG entry bot);
```

```
        callee_state ← sp_read_local (FunctionResult p, route cc ctx entry ca);
        sp_map (λd. DG d bot)
          (dg_spec_combine_transfer S (call_info_of ca p)
            (mk_dg_man cont global_of)
            (dg_local callee_state))
      }"
  unfolding routed_call_alternative_program_def using assms by (simp add: Let_def)

definition routed_callee_call_program ::
  "(pp × 'c, 'k, 'v, 'D::bounded_semilattice_sup_bot,
     'G::bounded_semilattice_sup_bot) dg_spec ⇒ ('v ⇒ 'k)
   ⇒ (pp ⇒ 'c ⇒ 'k)
   ⇒ (pp ⇒ 'c ⇒ 'D ⇒ call_action ⇒ 'c)
   ⇒ ('D ⇒ bool)
   ⇒ 'c ⇒ call_action ⇒ pp ⇒ 'D ⇒ pname
   ⇒ (pp × 'c, 'k, ('D, 'G) dg_state, ('D, 'G) dg_state) strategy_program"
where
  "routed_callee_call_program S global_of seed route is_bot ctx ca cc caller p =
     do {
       pairs ← enter♯ S (call_info_of ca p) (mk_dg_man caller global_of);
       side_rhs_fold_dg bot
         (map (routed_call_alternative_program S global_of seed route is_bot ctx ca cc p) pairs)
     }"
```

The call site's own contribution: read the caller state, ask `resolve` which procedures this site can enter given that state, and join what each resolved activation returns. The callee is the resolver's answer, not a component of the enumeration the equation system was built from, so a resolver reading the caller state selects the target at solve time. `static_targets` is the resolver that ignores the state and answers from the CFG, which reproduces the statically enumerated behaviour exactly.

An empty answer contributes ⊥. Under `static_targets` that happens only where the CFG has no call edge, i.e. where the enumeration produced no combine tree either.

```
definition routed_call_program ::
  "(pp × 'c, 'k, 'v, 'D::bounded_semilattice_sup_bot,
     'G::bounded_semilattice_sup_bot) dg_spec ⇒ ('v ⇒ 'k)
   ⇒ (pp ⇒ 'c ⇒ 'k)
   ⇒ (pp ⇒ pp ⇒ call_action ⇒ 'D ⇒ pname list)
   ⇒ ('D ⇒ bool)
   ⇒ (pp ⇒ 'c ⇒ 'D ⇒ call_action ⇒ 'c)
   ⇒ 'c ⇒ call_action ⇒ pp ⇒ pp
   ⇒ (pp × 'c, 'k, ('D, 'G) dg_state, ('D, 'G) dg_state) strategy_program"
where
  "routed_call_program S global_of seed resolve is_bot route ctx ca cc v =
     do {
       caller_state ← sp_read_local (cc, ctx);
       side_rhs_fold_dg bot
         (map (routed_callee_call_program S global_of seed route is_bot ctx ca cc
                (dg_local caller_state))
              (resolve v cc ca (dg_local caller_state)))
     }"
```

The seed read-back hook: at a callee entry it reads the seed out of the `dg_local` half, matching `routed_call_program`'s write.

```
definition routed_entry_seed_programs ::
  "(pp ⇒ 'c ⇒ 'k)
   ⇒ (pp ⇒ 'c ⇒ 'D::bounded_semilattice_sup_bot ⇒ call_action ⇒ 'c)
   ⇒ 'c ⇒ pp
   ⇒ (pp × 'c, 'k, ('D, 'G::bounded_semilattice_sup_bot) dg_state,
        ('D, 'G) dg_state) strategy_program list"
where
  "routed_entry_seed_programs seed route ctx v =
     (case v of FunctionEntry _ ⇒
        [do {
           seed_state ← sp_read_global (seed v ctx);
           sp_return (DG (dg_local seed_state) bot)
         }]
      | _ ⇒ [])"

lemma routed_entry_seed_programs_free:
  "x ∈ set (routed_entry_seed_programs seed route ctx v)
     ⟹ sides_of_program x tau z = bot"
  unfolding routed_entry_seed_programs_def
  by (cases v) (auto simp: sp_compile_def bot_fun_def)
```

```
lemma routed_entry_seed_programs_local_only:
  "x ∈ set (routed_entry_seed_programs seed route ctx v)
     ⟹ dg_global (traverse_program x tau) = bot"
  unfolding routed_entry_seed_programs_def by (cases v) (auto simp: sp_compile_def)
```

Every routed hook runs its continuation once, given that the specification's transfers do. The generator's fold asks for exactly that, and a context policy discharges it by naming its specification's `dg_spec_wf`.

```
lemma sp_wf_routed_entry_seed_programs [intro]:
  "x ∈ set (routed_entry_seed_programs seed route ctx v) ⟹ sp_wf x"
  unfolding routed_entry_seed_programs_def by (cases v) auto

lemma sp_wf_routed_call_alternative_program [intro]:
  assumes "dg_spec_wf S"
  shows
    "sp_wf (routed_call_alternative_program S global_of seed route is_bot ctx ca cc p alt)"
  unfolding routed_call_alternative_program_def Let_def
  by (auto intro!: sp_wf_map sp_wf_bind dg_spec_wf_combine[OF assms] split: prod.splits)

lemma sp_wf_routed_callee_call_program [intro]:
  assumes "dg_spec_wf S"
  shows
    "sp_wf (routed_callee_call_program S global_of seed route is_bot ctx ca cc caller p)"
  unfolding routed_callee_call_program_def
  by (intro sp_wf_bind dg_spec_wf_enter[OF assms] sp_wf_side_rhs_fold_dg)
     (auto intro: sp_wf_routed_call_alternative_program[OF assms])

lemma sp_wf_routed_call_program [intro]:
  assumes "dg_spec_wf S"
  shows
    "sp_wf (routed_call_program S global_of seed resolve is_bot route ctx ca cc v)"
  unfolding routed_call_program_def
  by (intro sp_wf_bind sp_wf_read_local sp_wf_side_rhs_fold_dg)
     (auto intro: sp_wf_routed_callee_call_program[OF assms])

lemma sp_wf_routed_call_alternative_programs [intro]:
  "dg_spec_wf S
     ⟹ ∀q ∈ set (map (routed_call_alternative_program S global_of seed route
             is_bot ctx ca cc p) pairs). sp_wf q"
  by (auto intro: sp_wf_routed_call_alternative_program)

lemma sp_wf_routed_callee_call_programs [intro]:
  "dg_spec_wf S
     ⟹ ∀q ∈ set (map (routed_callee_call_program S global_of seed route
             is_bot ctx ca cc caller) ps). sp_wf q"
  by (auto intro: sp_wf_routed_callee_call_program)
```

How a routed per-target program relates to its parts under a solution. Each bound is stated for *one* alternative of the list a single run of the entry produced, because the program is a fold over exactly those. These are the only facts CALL and COMB below need about its body.

```
lemma routed_callee_call_program_sides_ge_seed:
  assumes wfS: "dg_spec_wf S"
    and R: "enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
                (mk_dg_man caller global_of) σ pairs pub"
    and mem: "(cont, entry) ∈ set pairs"
    and nb: "¬ is_bot entry"
  shows "DG entry bot
           ≤ sides_of_program (routed_callee_call_program S global_of seed route is_bot ctx
                (CallEdge dst pars args) cc caller p) σ
                (Inr (seed (FunctionEntry p)
                   (route cc ctx entry (CallEdge dst pars args))))"
proof -
  let ?F = "routed_call_alternative_program S global_of seed route is_bot ctx
              (CallEdge dst pars args) cc p"
  let ?k = "Inr (seed (FunctionEntry p) (route cc ctx entry (CallEdge dst pars args)))"
  have "DG entry bot ≤ sides_of_program (?F (cont, entry)) σ ?k"
    unfolding routed_call_alternative_program_def using nb
    by (simp add: Let_def sp_compile_def)
  also have "... ≤ sides_of_program (side_rhs_fold_dg bot (map ?F pairs)) σ ?k"
    by (rule sides_le_side_rhs_fold_dg[OF sp_wf_routed_call_alternative_programs[OF wfS]])
       (use mem in force)
  also have "... ≤ sides_of_program (routed_callee_call_program S global_of seed route is_bot ctx
                     (CallEdge dst pars args) cc caller p) σ ?k"
    unfolding routed_callee_call_program_def
    by (simp add: sp_compile_bind sp_wf_observes enter_runsD_sides[OF R] sup_fun_def)
  finally show ?thesis .
```

```
qed
```

One rung further out: one resolved target's contribution survives the fold over the site's targets. Stated without a locale so a proof that must place a publication inside the solution — one whose concretization reads the global channel — can walk outwards from a call site it has not yet routed.

```
lemma routed_call_program_sides_ge_at:
  assumes wfS: "dg_spec_wf S"
    and mem: "p ∈ set (resolve v cc ca (dg_local (σ (Inl (cc, ctx)))))"
  shows "sides_of_program (routed_callee_call_program S global_of seed route is_bot ctx ca cc
            (dg_local (σ (Inl (cc, ctx)))) p) σ z
          ≤ sides_of_program
              (routed_call_program S global_of seed resolve is_bot route ctx ca cc v) σ z"
proof -
  have "routed_callee_call_program S global_of seed route is_bot ctx ca cc
          (dg_local (σ (Inl (cc, ctx)))) p
          ∈ set (map (routed_callee_call_program S global_of seed route is_bot ctx ca cc
                       (dg_local (σ (Inl (cc, ctx)))))
                  (resolve v cc ca (dg_local (σ (Inl (cc, ctx))))))"
    using mem by simp
  from sides_le_side_rhs_fold_dg[OF sp_wf_routed_callee_call_programs[OF wfS] this,
       where acc = bot and k = z]
  show ?thesis
    by (simp add: routed_call_program_def sp_compile_bind sp_wf_observes)
qed

lemma routed_callee_call_program_sides_ge_combine:
  assumes wfS: "dg_spec_wf S"
    and R: "enter_runs (enter♯ S (call_info_of ca p))
              (mk_dg_man caller global_of) σ pairs pub"
    and mem: "(cont, entry) ∈ set pairs"
  shows "sides_of_program
           (routed_call_alternative_program S global_of seed route is_bot ctx ca cc p (cont, entry))
             σ z
           ≤ sides_of_program
               (routed_callee_call_program S global_of seed route is_bot ctx ca cc caller p) σ z"
proof -
  let ?F = "routed_call_alternative_program S global_of seed route is_bot ctx ca cc p"
  have "sides_of_program (?F (cont, entry)) σ z
          ≤ sides_of_program (side_rhs_fold_dg bot (map ?F pairs)) σ z"
    by (rule sides_le_side_rhs_fold_dg[OF sp_wf_routed_call_alternative_programs[OF wfS]])
       (use mem in force)
  also have "... ≤ sides_of_program
                     (routed_callee_call_program S global_of seed route is_bot ctx ca cc caller p) σ z"
    unfolding routed_callee_call_program_def
    by (simp add: sp_compile_bind sp_wf_observes enter_runsD_sides[OF R] sup_fun_def)
  finally show ?thesis .
qed
```

And the answer: the value a selected alternative's return combine produces is below what the call answers, because the call answers the join over every alternative. This is the bound COMB needs — one alternative's combine is covered, not all of them jointly. It holds for a bottom alternative too, whose combine runs against $\bot$.

```
lemma routed_callee_call_program_traverse_ge:
  assumes wfS: "dg_spec_wf S"
    and R: "enter_runs (enter♯ S (call_info_of ca p))
              (mk_dg_man caller global_of) σ pairs pub"
    and mem: "(cont, entry) ∈ set pairs"
  shows "dg_local (traverse_program
           (routed_call_alternative_program S global_of seed route is_bot ctx ca cc p (cont, entry))
             σ)
           ≤ dg_local (traverse_program
               (routed_callee_call_program S global_of seed route is_bot ctx ca cc caller p) σ)"
proof -
  let ?F = "routed_call_alternative_program S global_of seed route is_bot ctx ca cc p"
  have "dg_local (traverse_program (?F (cont, entry)) σ)
          ≤ side_acc_dg bot σ (map ?F pairs)"
    by (rule dg_local_traverse_le_side_acc_dg) (use mem in force)
  also have "... = dg_local (traverse_program
                     (routed_callee_call_program S global_of seed route is_bot ctx ca cc caller p) σ)"
    unfolding routed_callee_call_program_def
    by (simp add: sp_compile_bind sp_wf_observes enter_runsD_traverse[OF R]
        traverse_side_rhs_fold_dg[OF sp_wf_routed_call_alternative_programs[OF wfS]])
  finally show ?thesis .
qed
```

When a routed combine publishes nothing at the analysis global

Whether a program needs reshaping before the buffered generator can consume it is a property of that program, not of the analysis that produced it. The two facts below say when a routed combine needs none.

- Its answer always carries `bot` on the globals half: the per-call-site fold accumulates only `dg_local`.
- It publishes nothing at `buffer_key` as soon as the spec's own enter and combine sub-programs publish nothing there — the only side the routed program adds is the seed, and a seed key is never `buffer_key`.

  A program with both properties is already its own contribution analogue, so the buffered generator's `cmb_c` hook can be the program itself. A spec whose transfers do publish at `buffer_key` fails the second and must supply a reshaped hook instead.

```
lemma sides_side_rhs_fold_dg_bot:
  assumes wf: "∀t ∈ set ts. sp_wf t"
    and "⋀t. t ∈ set ts ⟹ sides_of_program t σ z = bot"
  shows "sides_of_program (side_rhs_fold_dg acc ts) σ z = bot"
  using assms
  by (simp add: sides_of_program_side_rhs_fold_dg_char[OF wf] foldr_sup_bot_of_all_bot)

lemma routed_call_program_global_free:
  assumes wfS: "dg_spec_wf S"
  shows "dg_global (traverse_program
          (routed_call_program S global_of seed resolve is_bot route ctx ca cc v) σ) = bot"
  by (simp add: routed_call_program_def sp_compile_bind sp_wf_observes
      traverse_side_rhs_fold_dg[OF sp_wf_routed_callee_call_programs[OF wfS]])
```

Every hook the routed generator builds from a well-formed specification answers nothing on its global half: an edge answers its transfer's local value, a call answers the folded return values, and a seed answers its local slot. So the buffered generator a solver runs is the unbuffered one the framework is stated over, at every specification, whether or not it publishes at the analysis global.

```
lemma dg_global_traverse_dg_spec_edge_program:
  assumes "dg_spec_wf S"
  shows "dg_global (traverse_program (dg_spec_edge_program S a src unknown_of) σ) = bot"
proof -
  let ?P = "transfer_program_at (λm. dg_spec_step S a (outer_man (dgs_query S) m)) src unknown_of"
  have wf: "sp_wf ?P"
    unfolding transfer_program_at_def
    by (intro sp_wf_bind sp_wf_dg_read_at) (rule dg_spec_wf_step[OF assms])
  have "?P (λv. Answer (DG v bot))
     = sp_lift_tree (?P (λv. Answer (DG v bot))) (λd. Answer (DG (dg_local d) bot))"
    using sp_wf_graft[OF wf, where k = "λv. Answer (DG v bot)"
        and h = "λd. Answer (DG (dg_local d) bot)"]
    by simp
  then have "traverse_rhs (?P (λv. Answer (DG v bot))) σ
     = DG (dg_local (traverse_rhs (?P (λv. Answer (DG v bot))) σ)) bot"
    by (metis (no_types, lifting) traverse_rhs.simps(1) traverse_rhs_sp_lift_tree)
  then show ?thesis
    by (simp add: dg_spec_edge_program_def transfer_program_def sp_compile_def
        sp_compile_with_def) (metis dg_state.sel(2))
qed

theorem pp_routed_of_buffered:
  assumes wfS: "dg_spec_wf S"
    and ne: "⋀p ctx. seed p ctx ≠ buffer_key"
    and pp: "part_post_solution
     (routed_node_rhs_buffered intra_predecessor_addr_list call_site_list (λ_. buffer_key) route
        (λctx' src a. dg_spec_edge_program S a src global_of)
        (routed_call_program S global_of seed resolve is_bot)
        (routed_entry_seed_programs seed)
        g bot0 d₀ g₀)
      x0 sigma vars"
  shows "part_post_solution
     (routed_node_rhs intra_predecessor_addr_list call_site_list (λ_. buffer_key) route
        (λctx' src a. dg_spec_edge_program S a src global_of)
        (routed_call_program S global_of seed resolve is_bot)
        (routed_entry_seed_programs seed)
        g bot0 d₀ g₀)
     x0 sigma vars"
proof (rule part_post_solution_routed_node_rhs_buffered_answers_local[OF _ _ pp])
  show "⋀c' w. ∀p ∈ set (routed_contribution_programs intra_predecessor_addr_list
          call_site_list route (λctx' src a. dg_spec_edge_program S a src global_of)
```

```
          (routed_call_program S global_of seed resolve is_bot)
          (routed_entry_seed_programs seed) g c' w). sp_wf p"
    by (rule routed_contribution_programs_wf)
       (auto intro: sp_wf_dg_spec_edge_program[OF wfS] sp_wf_routed_call_program[OF wfS])
  show "⋀c' w p τ. p ∈ set (routed_contribution_programs intra_predecessor_addr_list
          call_site_list route (λctx' src a. dg_spec_edge_program S a src global_of)
          (routed_call_program S global_of seed resolve is_bot)
          (routed_entry_seed_programs seed) g c' w)
        ⟹ dg_global (traverse_program p τ) = bot"
    by (auto elim!: routed_contribution_programsE
        simp: routed_call_program_global_free[OF wfS] routed_entry_seed_programs_local_only
          dg_global_traverse_dg_spec_edge_program[OF wfS])
qed
```

The entry half of the condition is now stated through `enter_runs`: an entry program publishes nothing at `buffer_key` exactly when the `pub` it runs to is $\bot$ there. It has to be stated this way: an entry program answers a list of alternatives, not the solver's value type, so it has no compiled tree of its own to observe.

```
lemma routed_callee_call_program_side_free_at_buffer_key:
  assumes wfS: "dg_spec_wf S"
    and enter_free: "⋀ci d pairs pub.
        enter_runs (enter♯ S ci) (mk_dg_man d global_of) σ pairs pub ⟹ pub (Inr buffer_key) = bot"
    and enter_runs_ex: "⋀ci d. ∃pairs pub.
        enter_runs (enter♯ S ci) (mk_dg_man d global_of) σ pairs pub"
    and comb_free: "⋀ci d de z. sides_of_rhs (sp_compile_with (λx. DG x bot)
        (dg_spec_combine_transfer S ci (mk_dg_man d global_of) de)) σ z = bot"
    and ne: "⋀p ctx'. seed (FunctionEntry p) ctx' ≠ buffer_key"
  shows "sides_of_program
          (routed_callee_call_program S global_of seed route is_bot ctx ca cc caller p) σ (Inr buffer_key)
         = bot"
proof -
  obtain pairs pub
    where R: "enter_runs (enter♯ S (call_info_of ca p))
                (mk_dg_man caller global_of) σ pairs pub"
    using enter_runs_ex by blast
  have alts: "⋀t. t ∈ set (map (routed_call_alternative_program S global_of seed route is_bot ctx
                                 ca cc p) pairs)
                ⟹ sides_of_program t σ (Inr buffer_key) = bot"
    by (auto simp: routed_call_alternative_program_def Let_def sp_compile_def
        comb_free ne[THEN not_sym] split: if_splits)
  have "sides_of_program
          (routed_callee_call_program S global_of seed route is_bot ctx ca cc caller p) σ (Inr buffer_key)
        = pub (Inr buffer_key)
          ⊔ sides_of_program (side_rhs_fold_dg bot
                (map (routed_call_alternative_program S global_of seed route is_bot ctx ca cc p)
                  pairs))
              σ (Inr buffer_key)"
    unfolding routed_callee_call_program_def
    by (simp add: sp_compile_bind sp_wf_observes enter_runsD_sides[OF R] sup_fun_def)
  also have "... = bot"
  proof -
    have "sides_of_program (side_rhs_fold_dg bot
            (map (routed_call_alternative_program S global_of seed route is_bot ctx ca cc p) pairs))
            σ (Inr buffer_key) = bot"
      by (rule sides_side_rhs_fold_dg_bot[OF sp_wf_routed_call_alternative_programs[OF wfS]])
         (rule alts)
    then show ?thesis using enter_free[OF R] by simp
  qed
  finally show ?thesis .
qed

lemma routed_call_program_side_free_at_buffer_key:
  assumes wfS: "dg_spec_wf S"
    and enter_free: "⋀ci d pairs pub.
        enter_runs (enter♯ S ci) (mk_dg_man d global_of) σ pairs pub ⟹ pub (Inr buffer_key) = bot"
    and enter_runs_ex: "⋀ci d. ∃pairs pub.
        enter_runs (enter♯ S ci) (mk_dg_man d global_of) σ pairs pub"
    and comb_free: "⋀ci d de z. sides_of_rhs (sp_compile_with (λx. DG x bot)
        (dg_spec_combine_transfer S ci (mk_dg_man d global_of) de)) σ z = bot"
    and ne: "⋀p ctx'. seed (FunctionEntry p) ctx' ≠ buffer_key"
  shows "sides_of_program (routed_call_program S global_of seed resolve is_bot route ctx ca cc v)
          σ (Inr buffer_key) = bot"
proof -
  have at_free: "⋀caller p.
      sides_of_program (routed_callee_call_program S global_of seed route is_bot ctx ca cc caller p)
        σ (Inr buffer_key) = bot"
    by (rule routed_callee_call_program_side_free_at_buffer_key[OF wfS])
```

```
      (rule enter_free enter_runs_ex comb_free ne, assumption?)+
  have "sides_of_program (side_rhs_fold_dg bot
         (map (routed_callee_call_program S global_of seed route is_bot ctx ca cc
                 (dg_local (σ (Inl (cc, ctx)))))
            (resolve v cc ca (dg_local (σ (Inl (cc, ctx)))))) σ (Inr buffer_key) = bot"
    by (rule sides_side_rhs_fold_dg_bot[OF sp_wf_routed_callee_call_programs[OF wfS]])
      (auto simp: at_free)
  then show ?thesis by (simp add: routed_call_program_def sp_compile_bind sp_wf_observes)
qed


end
```

6.15 Routed_Context

theory Routed_Context
  imports Routed_Call_Programs DG_Ctx_Activation DG_Local_State_Spec
    Activation_Backbone
begin

One route, CALL and COMB discharged once

dg_context_activation already discharges EDGE (dg_ctx_act_edge) generically off the post-solution, independent of route/cmb/extra: intra edges never route. COMB cannot be discharged that generically, because cmb is an unconstrained parameter: a context-sensitive analysis whose entry-seed publication and return combine are hand-written has to rederive the routing argument itself. This theory fixes cmb and extra to the one canonical shape Routed_Call_Programs builds — parametric only in a routing function route and a seed-key injection seed — and discharges CALL and COMB as theorems of that shape: a k-call-string or a partial-tabulation context becomes an interpretation of this locale, not a second proof development.

The routed-context locale: D and G independently typed

routed_context instantiates dg_context_activation at routed_call_program/routed_entry_seed_programs, so S's own 'D/'G stay as independent as that locale already keeps them: no 'D = 'G constraint is threaded in by this locale's for clause. dg_context_activation itself carries no routing-specific content: every fact it supplies (pp_eq_bound, pp_sides_bound, sides_fold_le_Gen, edge_bound_local/_global, dg_ctx_act_edge) is already generic in cmb/extra, so instantiating it at routed_call_program/routed_entry_seed_programs reuses those proofs unchanged; only the seed-specific reasoning below, which dg_context_activation never has since seeding is routed_call_program's own addition, is carried out here.

Beyond dg_context_activation's parameters: seed injects a routed (pp, 'c) pair into the global-key space; adm is the trace-semantic context relation keying the activation-trace collecting semantics; and two obligations about it cannot be discharged generically. routing_adequate is adequacy: whenever adm admits a context for a real call edge and a covered caller store, some alternative of the specification's own entry run describes that call — its continuation half containing the caller store, its entry half containing the entered store, and its route being exactly the admitted context. The three come from one alternative rather than three, which is what keeps a multi-alternative entry from mixing one alternative's continuation with another's callee frame. routing_total is existence: every covered call admits some context. Restricting both to an edge of g (rather than quantifying over every value of the call_action type) matches how CALL and COMB below only ever invoke them at a matched edge, and keeps the obligations provable for an abstract domain that is exact on edges actually present in the program without being exact everywhere. These are per-instance proof obligations, not locale theorems.

calls_unique ties the edge a return reads to the edge its callee was entered through; the collecting semantics itself never needs it, and compiled programs have it for free.

```
locale routed_context =
  dg_context_activation S γ_DG 𝒢 g buffer_key global_of route
    "routed_call_program S global_of seed resolve is_bot"
      "routed_entry_seed_programs seed"
    bot0 d₀ g₀ sigma vars x0 sg γ_M
  for S :: "(pp × 'c, 'k, 'v, 'D::bounded_semilattice_sup_bot,
              'G::bounded_semilattice_sup_bot) dg_spec"
    and γ_DG :: "'D ⇒ ('v ⇒ 'G) ⇒ store set"
    and 𝒢 :: "vname ⇒ bool"
    and g buffer_key and global_of :: "'v ⇒ 'k"
    and route ("context♯")
    and bot0 d₀ g₀ sigma vars x0 sg
    and seed :: "pp ⇒ 'c ⇒ 'k"
    and resolve :: "pp ⇒ pp ⇒ call_action ⇒ 'D ⇒ pname list"
    and is_bot :: "'D ⇒ bool"
    and γ_M :: "'M ⇒ store set" +
  fixes adm :: "'c context_policy"
  assumes finC: "finite (calls g)"
    and calls_unique: "calls_source_unique g"
    and seed_ne_buffer_key[simp]: "⋀p ctx. seed p ctx ≠ buffer_key"
    and seed_ne_global_of[simp]: "⋀p ctx v. seed p ctx ≠ global_of v"
    and is_bot_bot[simp]: "is_bot bot"
    and is_bot_sound: "⋀d g. is_bot d ⟹ γ_DG d g = {}"
    and resolve_sound:
    "⋀u ctx dst pars args p cont s.
       (u, ctx) ∈ vars
       ⟹ (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
       ⟹ s ∈ γ_DG (dg_local (sigma (Inl (u, ctx)))) (genv global_of sigma)
```

```
      ⟹ p ∈ set (resolve cont u (CallEdge dst pars args)
                    (dg_local (sigma (Inl (u, ctx)))))"
  and routing_adequate:
  "⋀u ctx dst pars args p cont s ctx'.
     (u, ctx) ∈ vars
     ⟹ (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
     ⟹ s ∈ γ_DG (dg_local (sigma (Inl (u, ctx)))) (genv global_of sigma)
     ⟹ ctx' ∈ adm u ctx (call_info_of (CallEdge dst pars args) p) s
          (call_enter G (CallEdge dst pars args) s)
     ⟹ ∃pairs pub deps cont' entry.
          enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
            (mk_dg_man (dg_local (sigma (Inl (u, ctx)))) global_of) sigma pairs pub
        ∧ enter_deps (enter♯ S (call_info_of (CallEdge dst pars args) p))
            (mk_dg_man (dg_local (sigma (Inl (u, ctx)))) global_of) sigma pairs deps
        ∧ (cont', entry) ∈ set pairs
        ∧ s ∈ γ_DG cont' (genv global_of sigma)
        ∧ call_enter G (CallEdge dst pars args) s
            ∈ γ_DG entry (genv global_of sigma)
        ∧ context♯ u ctx entry (CallEdge dst pars args) = ctx'
        ∧ (FunctionEntry p, ctx') ∈ vars"
  and routing_total:
  "⋀u ctx dst pars args p cont s.
     (u, ctx) ∈ vars
     ⟹ (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
     ⟹ s ∈ γ_DG (dg_local (sigma (Inl (u, ctx)))) (genv global_of sigma)
     ⟹ adm u ctx (call_info_of (CallEdge dst pars args) p) s
                  (call_enter G (CallEdge dst pars args) s) ≠ {}"
  and comb_fwd:
  "⋀cl c1 dst pars args p cont.
     (cl, c1) ∈ vars ⟹ (cl, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
     ⟹ (cont, c1) ∈ vars"
begin
```

The reader's claim at a key, and the call contexts `adm` admits. Input-only, like `gamma_at`, so interpreted facts keep printing the explicit terms.

```
abbreviation (input) cover :: "pp ⇒ 'c ⇒ store set" where
  "cover v ctx ≡ γ_M (sg (Inl (v, ctx)))"

abbreviation (input) admits where
  "admits ≡ admits_call_context G g adm"
```

One place unpacks the call's entry protocol. `routing_adequate` is a nest of existentials over one entry run, and both CALL and COMB need the same witnesses: unpacking it separately in each would let them drift onto *different* alternatives of the same call, which is precisely the correlation the pair list exists to keep.

Invoke it with `obtain`; it is not an elimination rule for the classical reasoner.

```
lemma routed_enter_witness:
  assumes covV: "(u, ctx) ∈ vars"
    and ce: "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sin: "s ∈ gamma_at u ctx"
    and Rc: "ctx' ∈ adm u ctx (call_info_of (CallEdge dst pars args) p) s
               (call_enter G (CallEdge dst pars args) s)"
  obtains pairs pub deps cont' entry
    where "enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
             (man_at u ctx) sigma pairs pub"
      and "enter_deps (enter♯ S (call_info_of (CallEdge dst pars args) p))
             (man_at u ctx) sigma pairs deps"
      and "(cont', entry) ∈ set pairs"
      and "s ∈ γ_DG cont' (genv global_of sigma)"
      and "call_enter G (CallEdge dst pars args) s
             ∈ γ_DG entry (genv global_of sigma)"
      and "context♯ u ctx entry (CallEdge dst pars args) = ctx'"
      and "(FunctionEntry p, ctx') ∈ vars"
  using routing_adequate[OF covV ce sin Rc] that by blast
```

The alternative the eliminator hands over cannot be bottom, and no case distinction is needed to see it: a bottom alternative concretizes to nothing, while this one demonstrably contains the store the callee actually starts from. A bottom alternative therefore never describes a real callee execution — it stays in the fold only because its combine stage may still have effects.

```
lemma entry_cover_not_bot:
  assumes "s' ∈ γ_DG entry gv"
  shows "¬ is_bot entry"
proof
  assume "is_bot entry"
```

```
  then have "γ_DG entry gv = {}" by (rule is_bot_sound)
  with assms show False by simp
qed

lemma le_dg_state_localD: "d ≤ d' ⟹ dg_local d ≤ dg_local d'"
  by (simp add: less_eq_dg_state_def)

lemma le_dg_state_globalD: "d ≤ d' ⟹ dg_global d ≤ dg_global d'"
  by (simp add: less_eq_dg_state_def)
```

Reaching a resolved callee's contribution

A call edge reaches the generated equation in two hops: the call site's own program is one of the node's contribs, and the resolved callee's program is one summand of that site's fold. The second hop needs a concrete store at the call site, since a resolver may drop a target the abstract caller state rules out; where a store exists, the edge's own callee survives resolution.

```
lemma resolved_site_mem:
  assumes ce: "(cc, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
  shows "routed_call_program S global_of seed resolve is_bot context♯ ctx
           (CallEdge dst pars args) cc cont
         ∈ set (contribs cont ctx)"
proof -
  have "(cc, CallEdge dst pars args) ∈ set (call_site_list g cont)"
    using ce by (auto simp: set_call_site_list[OF finC])
  then show ?thesis by (rule routed_contribution_programs_combineI)
qed

lemma resolved_target_mem:
  assumes covV: "(cc, ctx) ∈ vars"
    and ce: "(cc, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sin: "s ∈ gamma_at cc ctx"
  shows "routed_callee_call_program S global_of seed context♯ is_bot ctx (CallEdge dst pars args) cc
           (dg_local (sigma (Inl (cc, ctx)))) p
         ∈ set (map (routed_callee_call_program S global_of seed context♯ is_bot ctx
                       (CallEdge dst pars args) cc
                     (dg_local (sigma (Inl (cc, ctx)))))
                    (resolve cont cc (CallEdge dst pars args)
                       (dg_local (sigma (Inl (cc, ctx))))))"
  using resolve_sound[OF covV ce sin] by simp

lemma resolved_at_le_site_acc:
  assumes covV: "(cc, ctx) ∈ vars"
    and ce: "(cc, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sin: "s ∈ gamma_at cc ctx"
  shows "dg_local (traverse_program
           (routed_callee_call_program S global_of seed context♯ is_bot ctx (CallEdge dst pars args) cc
              (dg_local (sigma (Inl (cc, ctx)))) p) sigma)
         ≤ side_acc_dg (acc0 cont) sigma (contribs cont ctx)"
proof -
  have "dg_local (traverse_program
           (routed_callee_call_program S global_of seed context♯ is_bot ctx (CallEdge dst pars args) cc
              (dg_local (sigma (Inl (cc, ctx)))) p) sigma)
       ≤ dg_local (traverse_program
           (routed_call_program S global_of seed resolve is_bot context♯ ctx
              (CallEdge dst pars args) cc cont)
           sigma)"
    using dg_local_traverse_le_side_acc_dg[OF resolved_target_mem[OF covV ce sin], where acc = bot]
    by (simp add: routed_call_program_def sp_compile_bind sp_wf_observes
        traverse_side_rhs_fold_dg[OF sp_wf_routed_callee_call_programs[OF spec_wf]])
  also have "... ≤ side_acc_dg (acc0 cont) sigma (contribs cont ctx)"
    by (rule dg_local_traverse_le_side_acc_dg[OF resolved_site_mem[OF ce]])
  finally show ?thesis .
qed

lemma resolved_at_le_site_sides:
  assumes covV: "(cc, ctx) ∈ vars"
    and ce: "(cc, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sin: "s ∈ gamma_at cc ctx"
  shows "sides_of_program
           (routed_callee_call_program S global_of seed context♯ is_bot ctx (CallEdge dst pars args) cc
              (dg_local (sigma (Inl (cc, ctx)))) p) sigma z
         ≤ sides_of_program (side_rhs_fold_dg (acc0 cont) (contribs cont ctx)) sigma z"
proof -
  have "sides_of_program
           (routed_callee_call_program S global_of seed context♯ is_bot ctx (CallEdge dst pars args) cc
```

```
            (dg_local (sigma (Inl (cc, ctx)))) p) sigma z
      ≤ sides_of_program
           (routed_call_program S global_of seed resolve is_bot context♯ ctx
              (CallEdge dst pars args) cc cont)
           sigma z"
    by (rule routed_call_program_sides_ge_at
          [where resolve = resolve and v = cont and cc = cc and ctx = ctx and σ = sigma,
           OF spec_wf resolve_sound[OF covV ce sin]])
  also have "... ≤ sides_of_program (side_rhs_fold_dg (acc0 cont) (contribs cont ctx)) sigma z"
    by (rule sides_le_side_rhs_fold_dg[OF contribs_wf resolved_site_mem[OF ce]])
  finally show ?thesis .
qed
```

CALL: the routed callee entry

The CALL obligation, ending in `routed_context_call`: the store a call hands the callee is covered by the callee entry read at the routed context. The bound does not go there directly. It goes through the seed unknown, which is the only place the caller's own program and the callee's entry equation meet.

```
lemma routed_seed_read_bound:
  assumes covV: "(FunctionEntry p, ctx') ∈ vars"
  shows "dg_local (sigma (Inr (seed (FunctionEntry p) ctx')))
         ≤ dg_local (sigma (Inl (FunctionEntry p, ctx')))"
proof -
  let ?t = "do { s ← sp_read_global (seed (FunctionEntry p) ctx');
                 sp_return (DG (dg_local s) bot) }"
  have mem: "?t ∈ set (contribs (FunctionEntry p) ctx')"
    by (rule routed_contribution_programs_extraI) (simp add: routed_entry_seed_programs_def)
  have "dg_local (sigma (Inr (seed (FunctionEntry p) ctx')))
      = dg_local (traverse_program ?t sigma)"
    by (simp add: sp_compile_def)
  also have "... ≤ side_acc_dg (acc0 (FunctionEntry p)) sigma (contribs (FunctionEntry p) ctx')"
    using dg_local_traverse_le_side_acc_dg[OF mem] .
  also have "... = dg_local (eq Gen (FunctionEntry p, ctx') sigma)"
    by (simp add: eq_routed_node_rhs[OF contribs_wf])
  also have "... ≤ dg_local (sigma (Inl (FunctionEntry p, ctx')))"
    using pp_eq_bound[OF covV] by (simp add: less_eq_dg_state_def)
  finally show ?thesis .
qed
```

The published callee-entry state is bounded by the seed unknown itself, and that bound needs only the continuation's coverage: the seed is written by the combine program living at the continuation, so nothing about the callee entry's own unknown enters the argument. `routed_seed_publish_bound_local` adds the one further hop from the seed to the callee-entry local, which is where the callee's own entry equation reads it back, and that hop is what needs the callee entry covered.

```
lemma routed_seed_publish_bound_seed:
  assumes covV: "(u, ctx) ∈ vars"
    and ce: "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sin: "s ∈ gamma_at u ctx"
    and covV_cont: "(cont, ctx) ∈ vars"
    and R: "enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
              (man_at u ctx) sigma pairs pub"
    and mem: "(cont', entry) ∈ set pairs"
    and nb: "¬ is_bot entry"
  shows "entry
         ≤ dg_local (sigma (Inr (seed (FunctionEntry p)
               (context♯ u ctx entry (CallEdge dst pars args)))))"
proof -
  let ?ctx' = "context♯ u ctx entry (CallEdge dst pars args)"
  let ?k = "Inr (seed (FunctionEntry p) ?ctx')"
  let ?t = "routed_callee_call_program S global_of seed context♯ is_bot ctx (CallEdge dst pars args) u
              (dg_local (sigma (Inl (u, ctx)))) p"
  have seedb: "DG entry bot ≤ sides_of_program ?t sigma ?k"
    using spec_wf R mem nb by (rule routed_callee_call_program_sides_ge_seed)
  have "entry ≤ dg_local (sides_of_program ?t sigma ?k)"
    using le_dg_state_localD[OF seedb] by simp
  also have "... ≤ dg_local (sides_of_program
      (side_rhs_fold_dg (acc0 cont) (contribs cont ctx)) sigma ?k)"
    by (rule le_dg_state_localD[OF resolved_at_le_site_sides[OF covV ce sin]])
  also have "... ≤ dg_local (sides_of_rhs (Gen (cont, ctx)) sigma ?k)"
    by (rule le_dg_state_localD[OF sides_fold_le_Gen])
  also have "... ≤ dg_local (sigma ?k)"
    using pp_sides_bound[OF covV_cont, THEN le_funD, of ?k]
    by (rule le_dg_state_localD)
```

```
  finally show ?thesis .
qed

lemma routed_seed_publish_bound_local:
  assumes covV_call: "(u, ctx) ∈ vars"
    and ce: "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sin: "s ∈ gamma_at u ctx"
    and covV_cont: "(cont, ctx) ∈ vars"
    and R: "enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
              (man_at u ctx) sigma pairs pub"
    and mem: "(cont', entry) ∈ set pairs"
    and nb: "¬ is_bot entry"
    and covV: "(FunctionEntry p, context♯ u ctx entry (CallEdge dst pars args)) ∈ vars"
  shows "entry
         ≤ dg_local (sigma (Inl (FunctionEntry p,
               context♯ u ctx entry (CallEdge dst pars args))))"
  by (rule order_trans
        [OF routed_seed_publish_bound_seed[OF covV_call ce sin covV_cont R mem nb]
           routed_seed_read_bound[OF covV]])

theorem routed_context_call:
  assumes ce: "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sin: "s ∈ cover u ctx"
    and Rc: "ctx' ∈ adm u ctx (call_info_of (CallEdge dst pars args) p) s
               (call_enter 𝒢 (CallEdge dst pars args) s)"
  shows "call_enter 𝒢 (CallEdge dst pars args) s ∈ cover (FunctionEntry p) ctx'"
proof (cases "(u, ctx) ∈ vars")
  case False
  hence "cover u ctx = {}" by (rule sg_uncov)
  thus ?thesis using sin by simp
next
  case True
  let ?g = "genv global_of sigma"
  have covV_cont: "(cont, ctx) ∈ vars"
    using comb_fwd[OF True ce] .
  have sin': "s ∈ γ_DG (dg_local (sigma (Inl (u, ctx)))) ?g"
    using sin True by simp
  obtain pairs pub deps cont' entry
    where Rr: "enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
                 (man_at u ctx) sigma pairs pub"
      and "enter_deps (enter♯ S (call_info_of (CallEdge dst pars args) p))
                 (man_at u ctx) sigma pairs deps"
      and mem: "(cont', entry) ∈ set pairs"
      and ecov: "call_enter 𝒢 (CallEdge dst pars args) s ∈ γ_DG entry ?g"
      and req: "context♯ u ctx entry (CallEdge dst pars args) = ctx'"
      and covV: "(FunctionEntry p, ctx') ∈ vars"
    by (rule routed_enter_witness[OF True ce sin' Rc])
  let ?ctx' = "context♯ u ctx entry (CallEdge dst pars args)"
  have nb: "¬ is_bot entry" using ecov by (rule entry_cover_not_bot)
  have "γ_DG entry ?g ⊆ γ_DG (dg_local (sigma (Inl (FunctionEntry p, ?ctx')))) ?g"
    by (rule gammaDG_mono
          [OF routed_seed_publish_bound_local[OF True ce sin' covV_cont Rr mem nb covV[folded req]]
              order_refl])
  also have "... = cover (FunctionEntry p) ?ctx'"
    using covV[folded req] by simp
  finally show ?thesis using ecov req by auto
qed
```

COMB: the routed return combine

The COMB obligation, ending in `routed_context_comb`: the state a return combines is covered by the continuation read at the caller's own context. The return takes one alternative among those the enter step produced, so the work is to bound that single alternative – once locally, once at the global key – against what the continuation's unknown already holds.

```
lemma routed_comb_bound_local:
  assumes covV: "(cl, c1) ∈ vars"
    and ce: "(cl, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sin: "s ∈ gamma_at cl c1"
    and covV_cont: "(cont, c1) ∈ vars"
    and R: "enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
              (man_at cl c1) sigma pairs pub"
    and mem: "(cont', entry) ∈ set pairs"
  shows "dg_local (traverse_program
           (routed_call_alternative_program S global_of seed context♯ is_bot c1
              (CallEdge dst pars args) cl p (cont', entry)) sigma)
```

```
        ≤ dg_local (sigma (Inl (cont, c1)))"
proof -
  have "dg_local (traverse_program
          (routed_call_alternative_program S global_of seed context♯ is_bot c1
             (CallEdge dst pars args) cl p (cont', entry)) sigma)
      ≤ dg_local (traverse_program
          (routed_callee_call_program S global_of seed context♯ is_bot c1 (CallEdge dst pars args) cl
             (dg_local (sigma (Inl (cl, c1)))) p) sigma)"
    by (rule routed_callee_call_program_traverse_ge[OF spec_wf R mem])
  also have "... ≤ side_acc_dg (acc0 cont) sigma (contribs cont c1)"
    by (rule resolved_at_le_site_acc[OF covV ce sin])
  also have "... = dg_local (eq Gen (cont, c1) sigma)"
    by (simp add: eq_routed_node_rhs[OF contribs_wf])
  also have "... ≤ dg_local (sigma (Inl (cont, c1)))"
    using pp_eq_bound[OF covV_cont] by (simp add: less_eq_dg_state_def)
  finally show ?thesis .
qed

lemma routed_comb_bound_global:
  assumes covV: "(cl, c1) ∈ vars"
    and ce: "(cl, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sin: "s ∈ gamma_at cl c1"
    and covV_cont: "(cont, c1) ∈ vars"
    and R: "enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
              (man_at cl c1) sigma pairs pub"
    and mem: "(cont', entry) ∈ set pairs"
  shows "dg_global (sides_of_program
          (routed_call_alternative_program S global_of seed context♯ is_bot c1
             (CallEdge dst pars args) cl p (cont', entry)) sigma (Inr k))
        ≤ dg_global (sigma (Inr k))"
proof -
  have "dg_global (sides_of_program
          (routed_call_alternative_program S global_of seed context♯ is_bot c1
             (CallEdge dst pars args) cl p (cont', entry)) sigma (Inr k))
      ≤ dg_global (sides_of_program
          (routed_callee_call_program S global_of seed context♯ is_bot c1 (CallEdge dst pars args) cl
             (dg_local (sigma (Inl (cl, c1)))) p) sigma (Inr k))"
    by (rule le_dg_state_globalD[OF routed_callee_call_program_sides_ge_combine[OF spec_wf R mem]])
  also have "... ≤ dg_global (sides_of_program
     (side_rhs_fold_dg (acc0 cont) (contribs cont c1)) sigma (Inr k))"
    by (rule le_dg_state_globalD[OF resolved_at_le_site_sides[OF covV ce sin]])
  also have "... ≤ dg_global (sides_of_rhs (Gen (cont, c1)) sigma (Inr k))"
    by (rule le_dg_state_globalD[OF sides_fold_le_Gen])
  also have "... ≤ dg_global (sigma (Inr k))"
    using pp_sides_bound[OF covV_cont, THEN le_funD, of "Inr k"]
    by (rule le_dg_state_globalD)
  finally show ?thesis .
qed

lemma routed_comb_bound_genv:
  assumes covV: "(cl, c1) ∈ vars"
    and ce: "(cl, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sin: "s ∈ gamma_at cl c1"
    and covV_cont: "(cont, c1) ∈ vars"
    and R: "enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
              (man_at cl c1) sigma pairs pub"
    and mem: "(cont', entry) ∈ set pairs"
  shows "genv global_of (sides_of_program
          (routed_call_alternative_program S global_of seed context♯ is_bot c1
             (CallEdge dst pars args) cl p (cont', entry)) sigma)
        ≤ genv global_of sigma"
  unfolding le_fun_def genv_def
  using routed_comb_bound_global[OF covV ce sin covV_cont R mem] by blast

theorem routed_context_comb:
  assumes ce: "(cl, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and s: "s ∈ cover cl c1"
    and adm: "admits cl c1 p' s es ctx'"
    and t: "t ∈ cover (FunctionResult p) ctx'"
  shows "combine_collect 𝒢 dst s t ∈ cover cont c1"
proof (cases "(cl, c1) ∈ vars")
  case False
  hence "cover cl c1 = {}" by (rule sg_uncov)
  thus ?thesis using s by simp
next
  case True
```

```
  let ?g = "genv global_of sigma"
  let ?ci = "call_info_of (CallEdge dst pars args) p"
  have sin: "s ∈ γ_DG (dg_local (sigma (Inl (cl, c1)))) ?g"
    using s True by simp
  — The callee was entered along some edge out of cl; call-source uniqueness makes it the edge the return reads.
  from adm obtain dst2 pars2 args2 cont2
    where e2: "(cl, CallEdge dst2 pars2 args2, FunctionEntry p', cont2) ∈ calls g"
      and es_eq: "es = call_enter 𝒢 (CallEdge dst2 pars2 args2) s"
      and Rc2: "ctx' ∈ adm cl c1 (call_info_of (CallEdge dst2 pars2 args2) p') s es"
    by (rule admits_call_contextE)
  have same: "CallEdge dst2 pars2 args2 = CallEdge dst pars args" and pp': "p' = p"
    using calls_source_unique_edgesD(1,2)[OF calls_unique e2 ce] by simp_all
  have Rc: "ctx' ∈ adm cl c1 ?ci s (call_enter 𝒢 (CallEdge dst pars args) s)"
    using Rc2 unfolding es_eq same pp' .
  obtain pairs pub deps cont' entry
    where Rr: "enter_runs (enter♯ S ?ci)
               (man_at cl c1) sigma pairs pub"
      and "enter_deps (enter♯ S ?ci)
               (man_at cl c1) sigma pairs deps"
      and mem: "(cont', entry) ∈ set pairs"
      and ccov: "s ∈ γ_DG cont' ?g"
      and ecov: "call_enter 𝒢 (CallEdge dst pars args) s ∈ γ_DG entry ?g"
      and req: "context♯ cl c1 entry (CallEdge dst pars args) = ctx'"
    by (rule routed_enter_witness[OF True ce sin Rc])
  let ?ex_ctx = "context♯ cl c1 entry (CallEdge dst pars args)"
  let ?alt = "routed_call_alternative_program S global_of seed context♯ is_bot c1
               (CallEdge dst pars args) cl p (cont', entry)"
  have nb: "¬ is_bot entry" using ecov by (rule entry_cover_not_bot)
  show ?thesis
  proof (cases "(FunctionResult p, ?ex_ctx) ∈ vars")
    case False
    hence "cover (FunctionResult p) ?ex_ctx = {}" by (rule sg_uncov)
    with req have "cover (FunctionResult p) ctx' = {}" by simp
    with t show ?thesis by simp
  next
    case True
    have covV_cont: "(cont, c1) ∈ vars"
      using comb_fwd[OF ‹(cl, c1) ∈ vars› ce] .
    have tin: "t ∈ γ_DG (dg_local (sigma (Inl (FunctionResult p, ?ex_ctx)))) ?g"
      using t req True by simp
    let ?sub = "sp_compile_with (λd. DG d bot)
                  (dg_spec_combine_transfer S ?ci (mk_dg_man cont' global_of)
                    (dg_local (sigma (Inl (FunctionResult p, ?ex_ctx)))))"
    have tr: "traverse_program ?alt sigma = traverse_rhs ?sub sigma"
      using nb by (simp add: sp_compile_def)
    have sd:
      "genv global_of (sides_of_program ?alt sigma) = genv global_of (sides_of_rhs ?sub sigma)"
    proof (rule ext)
      fix w
      have knk: "(Inr (global_of w) :: (pp × 'c) + 'k) ≠ Inr (seed (FunctionEntry p) ?ex_ctx)"
        by (rule not_sym) simp
      show "genv global_of (sides_of_program ?alt sigma) w = genv global_of (sides_of_rhs ?sub sigma) w"
        using nb
        by (simp add: genv_def Let_def sp_compile_def fun_upd_other[OF knk] del: fun_upd_apply)
    qed
    have "combine_collect 𝒢 dst s t
       ∈ γ_DG (dg_local (traverse_rhs ?sub sigma)) (?g ⊔ genv global_of (sides_of_rhs ?sub sigma))"
      using combine_sound[where q = cont'
          and r = "dg_local (sigma (Inl (FunctionResult p, ?ex_ctx)))"
          and τ = sigma and key = global_of and ci = ?ci, OF ccov tin]
      by simp
    also have "... = γ_DG (dg_local (traverse_program ?alt sigma))
                          (?g ⊔ genv global_of (sides_of_program ?alt sigma))"
      by (simp only: tr sd)
    also have "... ⊆ γ_DG (dg_local (sigma (Inl (cont, c1)))) ?g"
      by (rule gammaDG_mono[OF
            routed_comb_bound_local[OF ‹(cl, c1) ∈ vars› ce sin covV_cont Rr mem]
            sup_least[OF order_refl
              routed_comb_bound_genv[OF ‹(cl, c1) ∈ vars› ce sin covV_cont Rr mem]]])
    also have "... = cover cont c1"
      using covV_cont by simp
    finally show ?thesis .
  qed
qed
```

Activation-collect soundness against the routed local unknown

Every activation-collected store at any `(v, ctx)` pair is concretized by the routed local unknown's own γ_M reading. The five obligations of `activation_coverage` are this locale's own facts: the two entry bounds discharge `INIT` together, `dg_ctx_act_edge` is `INTRA`, the CALL and COMB theorems above are `CALL` and `RETURN`, and `routing_total` read through the reader is `TOTAL`. An instance therefore gets its activation-indexed soundness theorem by interpretation alone.

```
lemma routed_call_context_total:
  "call_context_total_on cover adm 𝒢 g"
  unfolding call_context_total_on_def
proof (intro allI impI)
  fix u dst pars args p cont ctx s
  assume ce: "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sin: "s ∈ cover u ctx"
  have covV: "(u, ctx) ∈ vars"
  proof (rule ccontr)
    assume "(u, ctx) ∉ vars"
    then have "cover u ctx = {}" by (rule sg_uncov)
    with sin show False by simp
  qed
  with sin have "s ∈ gamma_at u ctx" by simp
  from routing_total[OF covV ce this]
  show "adm u ctx (call_info_of (CallEdge dst pars args) p) s
                (call_enter 𝒢 (CallEdge dst pars args) s) ≠ {}" .
qed

theorem activation_collect_dg_sound:
  fixes S0 :: "store set" and c₀ :: 'c
  assumes entry_cov: "(cfg_entry g, c₀) ∈ vars"
    and init_sound: "S0 ⊆ γ_DG d₀ e₀"
    and init_env_le: "e₀ ≤ genv global_of sigma"
  shows "𝒜_{𝒢,adm,c₀,g,S0} v c ⊆ cover v c"
proof (rule activation_collect_sound[where cover = "cover"], unfold_locales)
  fix s0 assume s0mem: "s0 ∈ S0"
  have le_local: "d₀ ≤ dg_local (sigma (Inl (cfg_entry g, c₀)))"
    by (rule pp_entry_init_local_bound[OF entry_cov])
  have "γ_DG d₀ e₀
        ⊆ gamma_at (cfg_entry g) c₀"
    by (rule gammaDG_mono[OF le_local init_env_le])
  with s0mem init_sound have "s0 ∈ gamma_at (cfg_entry g) c₀" by blast
  thus "s0 ∈ cover (cfg_entry g) c₀"
    using entry_cov by simp
next
  fix u a v' c' s' s''
  assume "(u, a, v') ∈ intra g" "s' ∈ cover u c'" "s'' ∈ edge_step a s'"
  thus "s'' ∈ cover v' c'" by (rule dg_ctx_act_edge)
next
  fix u dst pars args p cont c' c'' s'
  assume ce: "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sm: "s' ∈ cover u c'"
    and Rc: "c'' ∈ adm u c' (call_info_of (CallEdge dst pars args) p) s'
               (call_enter 𝒢 (CallEdge dst pars args) s')"
  show "call_enter 𝒢 (CallEdge dst pars args) s' ∈ cover (FunctionEntry p) c''"
    using routed_context_call[OF ce sm Rc] .
next
  fix cl dst pars args p cont c1 c' p' s' t es
  assume ce: "(cl, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
    and sm: "s' ∈ cover cl c1"
    and adm: "admits cl c1 p' s' es c'"
    and tm: "t ∈ cover (FunctionResult p) c'"
  show "combine_collect 𝒢 dst s' t ∈ cover cont c1"
    using routed_context_comb[OF ce sm adm tm] .
next
  show "call_context_total_on cover adm 𝒢 g"
    by (rule routed_call_context_total)
qed
```

Every valid activation trace of a covered program carries some context under this instance's own `adm`: the same four EDGE/CALL/COMB/TOTAL facts that bound the buckets above also make `activation_coverage` total here, so an example such as `Example_Interval_Source_Ctx` can discharge the `has_ctx` premise `source_sound_from_collecting_ca` asks for without restating the interpretation itself.

```
lemma routed_valid_activation_trace_has_context:
  fixes S0 :: "store set" and c₀ :: 'c
```

```
  assumes entry_cov: "(cfg_entry g, c₀) ∈ vars"
    and init_sound: "S0 ⊆ γ_DG d₀ e₀"
    and init_env_le: "e₀ ≤ genv global_of sigma"
    and tv: "t ∈ 𝒯_{𝒢,g,S0}"
  shows "∃c. activation_context_rel 𝒢 adm c₀ g t c"
proof -
  interpret G: activation_coverage g S0 cover adm c₀ 𝒢
  proof unfold_locales
    fix s0 assume s0mem: "s0 ∈ S0"
    have le_local: "d₀ ≤ dg_local (sigma (Inl (cfg_entry g, c₀)))"
      by (rule pp_entry_init_local_bound[OF entry_cov])
    have "γ_DG d₀ e₀
          ⊆ gamma_at (cfg_entry g) c₀"
      by (rule gammaDG_mono[OF le_local init_env_le])
    with s0mem init_sound have "s0 ∈ gamma_at (cfg_entry g) c₀" by blast
    thus "s0 ∈ cover (cfg_entry g) c₀"
      using entry_cov by simp
  next
    fix u a v' c' s' s''
    assume "(u, a, v') ∈ intra g" "s' ∈ cover u c'" "s'' ∈ edge_step a s'"
    thus "s'' ∈ cover v' c'" by (rule dg_ctx_act_edge)
  next
    fix u dst pars args p cont c' c'' s'
    assume ce: "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
      and sm: "s' ∈ cover u c'"
      and Rc: "c'' ∈ adm u c' (call_info_of (CallEdge dst pars args) p) s'
                 (call_enter 𝒢 (CallEdge dst pars args) s')"
    show "call_enter 𝒢 (CallEdge dst pars args) s' ∈ cover (FunctionEntry p) c''"
      using routed_context_call[OF ce sm Rc] .
  next
    fix cl dst pars args p cont c1 c' p' s' t es
    assume ce: "(cl, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g"
      and sm: "s' ∈ cover cl c1"
      and adm: "admits cl c1 p' s' es c'"
      and tm: "t ∈ cover (FunctionResult p) c'"
    show "combine_collect 𝒢 dst s' t ∈ cover cont c1"
      using routed_context_comb[OF ce sm adm tm] .
  next
    show "call_context_total_on cover adm 𝒢 g"
      by (rule routed_call_context_total)
  qed
  show ?thesis using G.valid_activation_trace_has_context[OF tv] by blast
qed

end
```

The context relation a solved entry-state table induces

A state-dependent policy picks a context from an abstract entry state, and a pure entry answers a list of them, so one concrete call may be routed to several contexts — one per alternative whose continuation covers the caller store and whose entry covers the entered store. `routed_entry_context_rel alts` is exactly that set, for an entry operation whose alternatives are the pure function `alts` of the call information and the caller's solved local value: it admits `ctx'` whenever some alternative has those two coverings and its `route` is `ctx'`. Stated as an equivalence, both routed obligations come for free: adequacy is the definition read forwards, and totality is the alternatives covering the concrete call at all, which is what `entry_pairs_cover` says. No decoder of the concrete store is involved: the context is induced by the analyzer's own decision, as Goblint's `context` applied to each `enter` alternative induces it.

The relation is stated over `alts` rather than over an arbitrary entry run because a run does not determine its alternative list in general — over a one-point carrier every list traverses alike — while a pure entry's list is the function's own answer.

```
definition routed_entry_context_rel ::
  "(call_info ⇒ 'D ⇒ 'D enter_result list)
     ⇒ ('D ⇒ ('v ⇒ 'G) ⇒ store set) ⇒ (pp × 'c + 'k ⇒ ('D, 'G) dg_state) ⇒ ('v ⇒ 'k)
     ⇒ (pp ⇒ 'c ⇒ 'D ⇒ call_action ⇒ 'c) ⇒ 'c context_policy"
where
  "routed_entry_context_rel alts γ_DG sigma global_of route u ctx ci caller entered =
     {ctx'. ∃cont entry.
        (cont, entry) ∈ set (alts ci (dg_local (sigma (Inl (u, ctx)))))
      ∧ caller ∈ γ_DG cont (genv global_of sigma)
      ∧ entered ∈ γ_DG entry (genv global_of sigma)
      ∧ ctx' = route u ctx entry (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci))}"
```

```
lemma routed_entry_context_relI:
  assumes "(cont, entry) ∈ set (alts ci (dg_local (sigma (Inl (u, ctx)))))"
    and "caller ∈ γ_DG cont (genv global_of sigma)"
    and "entered ∈ γ_DG entry (genv global_of sigma)"
  shows "route u ctx entry (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci))
          ∈ routed_entry_context_rel alts γ_DG sigma global_of route u ctx ci caller entered"
  using assms unfolding routed_entry_context_rel_def by blast
```

Invoke with `obtain`: the alternative is a witness the classical reasoner should not be left to guess.

```
lemma routed_entry_context_relE:
  assumes
    "ctx' ∈ routed_entry_context_rel alts γ_DG sigma global_of route u ctx ci caller entered"
  obtains cont entry
    where "(cont, entry) ∈ set (alts ci (dg_local (sigma (Inl (u, ctx)))))"
      and "caller ∈ γ_DG cont (genv global_of sigma)"
      and "entered ∈ γ_DG entry (genv global_of sigma)"
      and "ctx' = route u ctx entry (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci))"
  using assms unfolding routed_entry_context_rel_def by blast
```

Totality is the alternatives covering the concrete call: exactly the `entry_pairs_cover` obligation every routed instance already carries.

```
lemma routed_entry_context_rel_total:
  assumes "entry_pairs_cover (λd. γ_DG d (genv global_of sigma)) caller entered
             (alts ci (dg_local (sigma (Inl (u, ctx)))))"
  shows
    "routed_entry_context_rel alts γ_DG sigma global_of route u ctx ci caller entered ≠ {}"
  using assms unfolding routed_entry_context_rel_def
  by blast
```

Formal-entry contexts: routing on the callee's declared formals

A context type derived from the callee's declared formals rather than call-site history: `'a list`, one abstract value per formal, in the order `CallEdge` already carries them — populated at compile time from the callee's own declaration, so no separate procedure-table lookup is needed here. `formals_context` is the plain per-variable projection (`'a abs_state` is `vname ⇒ 'a`, so this is just `map`), and the routing functions below apply it to the state they are handed. That state is already the *entered* callee frame: `routed_callee_call_program` runs the specification's own enter transfer first and passes its answer to `route`, so a routing function must not enter again. `formals_context_sem` is the trace-semantic counterpart, decoding the concrete entered store's formals the same way, given the point abstraction `decode` a domain provides for a concrete value and the CFG needed to look up a call site's own formal list. Neither definition mentions a domain-specific accessor beyond `decode` itself, nor a specification, so any domain reusing `routed_context` instantiates this pair once instead of hand-writing a per-formal projection.

```
definition formals_context :: "vname list ⇒ 'a abs_state ⇒ 'a list" where
  "formals_context pars d = map d pars"
```

The formals of the call originating at `u`: at most one, by the compiler's own invariant (`Voblint_Compile.VIMP_Proc_to_CFG` emits a single `CallEdge` per `activation_trace.Call`); `[]` if `u` has none.

```
definition formals_at_call_site :: "cfg ⇒ pp ⇒ vname list" where
  "formals_at_call_site g u =
     (case filter (λ(c, ca, ce, k). c = u) (cfg_calls_list g) of
        (_, CallEdge _ pars _, _, _) # _ ⇒ pars
      | _ ⇒ [])"

definition formals_context_sem ::
  "cfg ⇒ (int ⇒ 'a) ⇒ cfg_node ⇒ 'a list ⇒ store ⇒ 'a list"
where
  "formals_context_sem g decode u ctx s = formals_context (formals_at_call_site g u) (decode ∘ s)"
```

The whole matched `call_action` at a node, not only its formals: an `enterc` built purely from the caller's own solved abstract state (rather than by decoding the concrete entered store, as `formals_context_sem` does) needs the callee's actuals too, to recompute the same `enter♯` the route itself already ran. Same convention as `formals_at_call_site`: the head of the filtered call list, `CallEdge None [] []` if `u` has no outgoing call.

```
definition call_action_at_call_site :: "cfg ⇒ pp ⇒ call_action" where
  "call_action_at_call_site g u =
     (case filter (λ(c, ca, ce, k). c = u) (cfg_calls_list g) of
        (_, ca, _, _) # _ ⇒ ca
      | _ ⇒ CallEdge None [] [])"
```

Not a locale theorem, same as routing agreement itself: whether a node has at most one outgoing call is a per-instance fact about `g`, true for `compile_prog` output (`Voblint_Compile.VIMP_Proc_to_CFG`'s `compile_prog_calls_source_unique`) but not for an arbitrary hand-built CFG.

```
lemma calls_filter_singleton:
  assumes fin: "finite (calls g)"
    and uniq: "⋀ca1 ce1 af1 ca2 ce2 af2.
                (u, ca1, ce1, af1) ∈ calls g ⟹ (u, ca2, ce2, af2) ∈ calls g
                ⟹ ca1 = ca2 ∧ ce1 = ce2 ∧ af1 = af2"
    and ce: "(u, ca, cf, af) ∈ calls g"
  shows "filter (λ(c, ca, ce, k). c = u) (cfg_calls_list g) = [(u, ca, cf, af)]"
proof -
  let ?P = "λ(c, ca, ce, k). c = u"
  let ?L = "cfg_calls_list g"
  have mem: "(u, ca, cf, af) ∈ set ?L" using ce fin by simp
  have distinctL: "distinct ?L" unfolding cfg_calls_list_code by (rule distinct_sorted_list_of_set)
  have "set (filter ?P ?L) = {(u, ca, cf, af)}"
  proof (rule set_eqI, rule iffI)
    fix x assume hx: "x ∈ set (filter ?P ?L)"
    then have memx: "x ∈ set ?L" and px: "?P x" by auto
    obtain c ca' ce' af' where x: "x = (c, ca', ce', af')" by (cases x) auto
    from px x have cU: "c = u" by simp
    from memx x cU fin have "(u, ca', ce', af') ∈ calls g" by simp
    with uniq[OF ce] have "ca' = ca" "ce' = cf" "af' = af" by auto
    thus "x ∈ {(u, ca, cf, af)}" using x cU by simp
  next
    fix x assume "x ∈ {(u, ca, cf, af)}"
    thus "x ∈ set (filter ?P ?L)" using mem by simp
  qed
  moreover have "distinct (filter ?P ?L)" using distinctL by (rule distinct_filter)
  ultimately show ?thesis
    apply (cases "filter (λ(c, ca, ce, k). c = u) (cfg_calls_list g)")
    using singleton_iff subset_singletonD by(fastforce)+
qed

lemma call_action_at_call_site_eq:
  assumes fin: "finite (calls g)"
    and uniq: "⋀ca1 ce1 af1 ca2 ce2 af2.
                (u, ca1, ce1, af1) ∈ calls g ⟹ (u, ca2, ce2, af2) ∈ calls g
                ⟹ ca1 = ca2 ∧ ce1 = ce2 ∧ af1 = af2"
    and ce: "(u, ca, cf, af) ∈ calls g"
  shows "call_action_at_call_site g u = ca"
  unfolding call_action_at_call_site_def
  using calls_filter_singleton[OF fin uniq ce] by simp
```

Formal-entry contexts at the routed spine's lifted carrier

`formals_context` above operates on the unlifted `'a abs_state`, the shape an abstract-carrier `routed_context` caller state has. The routed executable spine (`dg_context_activation`, every current instance) instead carries `'a abs_state lifted` throughout, to represent an activation the solver has not yet covered. `formals_route_lifted`/`formals_route_lifted_gen` are the same formal-entry projection at that carrier: a caller point that is `Bot` routes to the all-`bot` formal context (the entered callee frame is then `Bot` too), the same collapse an EntryState routed instance needs for its own executable/abstract route. Like the projection above, these read the state they are handed and never re-enter, so neither mentions a specification.

```
definition formals_route_lifted ::
  "'a::numeric_domain abs_state lifted ⇒ call_action ⇒ 'a list"
where
  "formals_route_lifted d ca =
     (case ca of CallEdge dst pars args ⇒
        formals_context pars (case d of Bot ⇒ bot | Lifted d0 ⇒ d0))"

definition formals_route_lifted_gen ::
  "pp ⇒ 'a list ⇒ 'a::numeric_domain abs_state lifted ⇒ call_action ⇒ 'a list"
where
  "formals_route_lifted_gen u ctx d ca = formals_route_lifted d ca"


end
```

6.16 Routed_Context_Unit

```
theory Routed_Context_Unit
  imports Routed_Context
begin
```

The monovariant context as a routed-context instance

Every other `routed_context` instance (call-string, entry-state) routes to a non-trivial context. This theory checks the routed-context abstraction also admits the degenerate case a context-insensitive analysis needs: exactly one context, chosen the same way at every call, carrying no history and no dependence on the abstract state. `route_unit` is that routing function; nothing else about `routed_context` changes.

The context-insensitive analysis is the routed analysis registered at this routing function, exactly as the call-string and entry-state analyses are registered at theirs; nothing here is a second analysis. What this theory adds is the collapse at the end: at the unit context the activation-indexed collecting semantics is `node_collect`.

Unit routing

`route_unit` ignores every argument and always chooses the sole context `()`: no call-site history, no dependence on the caller's abstract state, no sentinel encoding. `enterc_unit` is its trace-semantic counterpart; its graph `context_policy_of_fun enterc_unit` instantiates `routed_context`'s `adm` parameter. The two are definitionally the same constant function, so `routing_adequate`'s routing agreement holds independently of any call edge, solved state, or concrete store.

```
definition route_unit :: "pp ⇒ unit ⇒ 'D ⇒ call_action ⇒ unit" where
  [simp]: "route_unit u ctx d ca = ()"

definition enterc_unit :: "cfg_node ⇒ unit ⇒ store ⇒ unit" where
  [simp]: "enterc_unit u ctx s = ()"
```

The unit context never filters a trace: every valid activation trace carries the one context `()`, so `activation_collect`'s context conjunct holds for every trace reaching `v` and the two collectors coincide. Domain-generic: no domain-specific fact is used, so every `unit`-context routed producer (Sign, Interval, ...) cites this one lemma rather than re-deriving it.

```
lemma activation_collect_unit_eq_node_collect:
  "𝒜_𝒢,context_policy_of_fun enterc_unit,(),g,S v () = 𝒞_𝒢,g,S v"
  unfolding activation_collect_of_fun node_collect_def by simp

end
```

6.17 DG_Local_State_Spec

```
theory DG_Local_State_Spec
  imports "Voblint_Framework.MCP_Spec" Transfer_Algebra "Voblint_Domain.Nonrelational_Reachability"
begin
```

What a whole-state analysis supplies, and when it is sound

A Base-style analysis answers from one pointwise abstract state per program point and never touches a global: its transfers read the manager's local value, compute, and return, so the compiled equations carry no `QueryG` and no `Side`. Such an analysis supplies seven pure edge-operation families — skip, assign, special, branch, body, return, event, with branch covering both `EA_Assume` and `EA_AssumeNot` — plus the callee entry, and this theory turns them into a `dg_spec` two ways: `state_dg_spec` on the raw states, and `lifted_state_dg_spec` on states carrying an explicit unreachable value, where a dead point collapses to `Bot` before any transfer runs.

`sound_nonrelational_transfer` is the contract those operations owe: every concrete transition the collecting semantics allows must land inside the concretization of what the corresponding operation computes. A domain discharges it once, by `interpretation`, and `state_dg_spec` becomes a sound specification. `lifted_state_dg_spec` is the structural target its executable mirror refines; `DG_Local_State_Exec_Refinement` proves that mirror sound directly.

The call boundary is fixed here rather than supplied. A call answers exactly one alternative, whose continuation is the caller value unchanged. That is sound because of what the fixed combine does later, not because of anything about the carrier: the continuation is kept for its *local* part, and whatever the callee may have invalidated — the globals — is taken from the callee exit instead. A nonrelational caller still holds global facts a callee can invalidate; it is the combine protocol that repairs them, not the absence of relations. The environment stage passes that continuation through, and the whole return happens in the assign stage as `combine`$^\sharp$: caller locals, callee globals, and the callee's `ret_var` written to the destination. An analysis that needs a different boundary overrides those three fields of its own specification instead of using these builders, the way `varEq` supplies its own `combine_env` upstream.

The transfer contract

One assumption per operation, each an inference rule from a concrete store in an abstract state's concretization to the corresponding concrete successor in the operation's own result. A domain proves one fact per operation about its own functions and interprets this locale once; nothing below asks it for more.

```
locale sound_nonrelational_transfer =
  fixes 𝒢 :: "vname ⇒ bool"
    and sk :: "'a::numeric_domain abs_state ⇒ 'a abs_state"
    and asn :: "vname ⇒ exp ⇒ 'a abs_state ⇒ 'a abs_state"
    and sp :: "special_call ⇒ vname ⇒ 'a abs_state ⇒ 'a abs_state"
    and br :: "exp ⇒ bool ⇒ 'a abs_state ⇒ 'a abs_state"
    and bd :: "pname ⇒ 'a abs_state ⇒ 'a abs_state"
    and rt :: "exp option ⇒ pname ⇒ 'a abs_state ⇒ 'a abs_state"
    and en :: "call_info ⇒ 'a abs_state ⇒ 'a abs_state"
    and ev :: "analysis_event ⇒ 'a abs_state ⇒ 'a abs_state"
  assumes tf_sound_assign_for[intro]:
    "s ∈ γ d ⟹ s(x := ⟦a⟧ₑ s) ∈ γ (asn x a d)"
  assumes tf_sound_special_for[intro]:
    "s ∈ γ d ⟹ special_result sc s v ⟹ s(x := v) ∈ γ (sp sc x d)"
  assumes tf_sound_branch_for[intro]:
    "s ∈ γ d ⟹ truthy (⟦b⟧ₑ s) = pol ⟹ s ∈ γ (br b pol d)"
  assumes tf_sound_skip_for[intro]:
    "s ∈ γ d ⟹ s ∈ γ (sk d)"
  assumes tf_sound_body_for[intro]:
    "s ∈ γ d ⟹ s ∈ γ (bd p d)"
  assumes tf_sound_return_for[intro]:
    "s ∈ γ d ⟹
       s(ret_var := (case e of None ⇒ s ret_var | Some a ⇒ ⟦a⟧ₑ s))
         ∈ γ (rt e p d)"
  assumes tf_sound_enter_entry_for[intro]:
    "s ∈ γ d ⟹
       bind_formals (ci_formals ci) (map (λe. ⟦e⟧ₑ s) (ci_args ci)) (enter_state 𝒢 s)
         ∈ γ (en ci d)"
  assumes tf_sound_event_for[intro]:
    "s ∈ γ d ⟹ s ∈ γ (ev evt d)"
```

Each obligation is stated directly as an inference rule ($\mathtt{P}_1 \Longrightarrow \ldots \Longrightarrow \mathtt{P}_n \Longrightarrow \mathtt{Q}$). Variables not fixed by the locale are implicitly generalized, so the assumptions compose directly with `rule`, `OF`, and `auto`; no separate Horn-clause restatement is needed.

```
context sound_nonrelational_transfer
begin
```

The per-edge dispatcher's soundness, which is what an equation generator's step obligation asks for: `local_spec_step` selects the operation the action names, and each selected operation is sound by one locale assumption.

```
lemma step_sound_for[intro]:
  "edge_collect a (γ d) ⊆ γ (local_spec_step sk asn sp br bd rt ev a d)"
proof (cases a)
  case (EA_Special sc x)
  then show ?thesis by (cases sc) auto
qed auto

end
```

The operations as a component

What a whole-state analysis actually supplies: eight pure operations over `'a abs_state`, packaged as a component whose handler answers nothing, whose entry yields the one callee state and leaves the caller's own value untouched, and whose return runs the fixed combine in the assign stage. The lifted sibling further down differs only in carrying the dead-code lift.

```
definition state_local_spec ::
  "(vname ⇒ bool)
   ⇒ ('a::numeric_domain abs_state ⇒ 'a abs_state)
   ⇒ (vname ⇒ exp ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (special_call ⇒ vname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (exp ⇒ bool ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (pname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (exp option ⇒ pname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (call_info ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (analysis_event ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ 'a abs_state local_spec"
where
  "state_local_spec 𝒢 sk asn sp br bd rt en ev = (|
     ls_query = (λ_ _. ⊤),
     ls_skip = (λ_. sk), ls_assign = (λ_. asn), ls_special = (λ_. sp),
     ls_branch = (λ_. br), ls_body = (λ_. bd), ls_return = (λ_. rt), ls_event = (λ_. ev),
     ls_enter = (λ_ ci p. [(fst p, en ci (fst p))]),
     ls_combine_env = (λ_ _ ci dc de. dc),
     ls_combine_assign = (λ_ ci. combine♯ 𝒢 (ci_dst ci)) |)"

lemma closed_step_state_local_spec [simp]:
  "⟦a⟧♯state_local_spec 𝒢 sk asn sp br bd rt en ev
     = local_spec_step sk asn sp br bd rt ev a"
  by (simp add: closed_step_def ls_step_def state_local_spec_def fun_eq_iff)

lemma ls_step_state_local_spec [simp]:
  "ls_step (state_local_spec 𝒢 sk asn sp br bd rt en ev) ch a
     = local_spec_step sk asn sp br bd rt ev a"
  by (simp add: ls_step_def state_local_spec_def)

lemma ls_combine_state_local_spec [simp]:
  "ls_combine (state_local_spec 𝒢 sk asn sp br bd rt en ev) ch ch' ci dc de
     = combine♯ 𝒢 (ci_dst ci) dc de"
  by (simp add: state_local_spec_def)

context sound_nonrelational_transfer
begin

abbreviation tf_spec :: "'a abs_state local_spec" where
  "tf_spec ≡ state_local_spec 𝒢 sk asn sp br bd rt en ev"

theorem state_local_spec_sound: "sound_local_spec 𝒢 gamma_state tf_spec"
proof -
  have step: "edge_collect a (γ d ∩ Collect (eval_query.channel_holds ch))
      ⊆ γ (ls_step tf_spec ch a d)" for a ch and d :: "'a abs_state"
    by (simp only: ls_step_state_local_spec)
       (rule subset_trans[OF edge_collect_mono[OF Int_lower1] step_sound_for])
  show ?thesis
    unfolding sound_local_spec_def
    using gamma_state_mono step
    by (auto simp: state_local_spec_def call_enter_CallEdge combine_collect_sound)
qed

end
```

The unlifted specification

`state_dg_spec` turns the state-level local specification into an equation-system specification via `dg_spec_of`, without the reachability lifting.

```
definition state_dg_spec ::
  "(vname ⇒ bool)
   ⇒ ('a::numeric_domain abs_state ⇒ 'a abs_state)
   ⇒ (vname ⇒ exp ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (special_call ⇒ vname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (exp ⇒ bool ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (pname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (exp option ⇒ pname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (call_info ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (analysis_event ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ ('x,'k,unit,'a abs_state,'g::bounded_semilattice_sup_bot) dg_spec"
where
  "state_dg_spec 𝒢 sk asn sp br bd rt en ev
     = dg_spec_of (state_local_spec 𝒢 sk asn sp br bd rt en ev)"

lemma dg_spec_step_state_dg_spec:
  "dg_spec_step (state_dg_spec 𝒢 sk asn sp br bd rt en ev) a
     = local_transfer (local_spec_step sk asn sp br bd rt ev a)"
  by (simp add: state_dg_spec_def)

lemma dg_spec_wf_state_dg_spec [intro, simp]:
  "dg_spec_wf (state_dg_spec 𝒢 sk asn sp br bd rt en ev)"
  by (simp add: state_dg_spec_def)

theorem (in sound_nonrelational_transfer) state_dg_spec_contract:
  "analysis_contract (state_dg_spec 𝒢 sk asn sp br bd rt en ev) (λd g. γ d) 𝒢"
  unfolding state_dg_spec_def by (rule dg_spec_of_contract[OF state_local_spec_sound])
```

The reachability-lifted construction

The same operations, each wrapped in `transfer_lift`: a `Bot` input stays `Bot`, and a result the supplied emptiness predicate flags collapses to it.

```
definition lifted_state_local_spec ::
  "(vname ⇒ bool)
   ⇒ ('a abs_state ⇒ bool)
   ⇒ ('a::numeric_domain abs_state ⇒ 'a abs_state)
   ⇒ (vname ⇒ exp ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (special_call ⇒ vname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (exp ⇒ bool ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (pname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (exp option ⇒ pname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (call_info ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (analysis_event ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ 'a abs_state lifted local_spec"
where
  "lifted_state_local_spec 𝒢 empty_pred sk asn sp br bd rt en ev = (|
     ls_query = (λ_ _. ⊤),
     ls_skip = (λ_. transfer_lift empty_pred sk),
     ls_assign = (λ_ x e. transfer_lift empty_pred (asn x e)),
     ls_special = (λ_ sc x. transfer_lift empty_pred (sp sc x)),
     ls_branch = (λ_ b pol. transfer_lift empty_pred (br b pol)),
     ls_body = (λ_ p. transfer_lift empty_pred (bd p)),
     ls_return = (λ_ e p. transfer_lift empty_pred (rt e p)),
     ls_event = (λ_ evt. transfer_lift empty_pred (ev evt)),
     ls_enter = (λ_ ci p. [(fst p, transfer_lift empty_pred (en ci) (fst p))]),
     ls_combine_env = (λ_ _ ci dc de. dc),
     ls_combine_assign = (λ_ ci dcM de.
        transfer_lift2 empty_pred (combine♯ 𝒢 (ci_dst ci)) dcM de) |)"

definition lifted_state_dg_spec ::
  "(vname ⇒ bool)
   ⇒ ('a abs_state ⇒ bool)
   ⇒ ('a::numeric_domain abs_state ⇒ 'a abs_state)
   ⇒ (vname ⇒ exp ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (special_call ⇒ vname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (exp ⇒ bool ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (pname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (exp option ⇒ pname ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (call_info ⇒ 'a abs_state ⇒ 'a abs_state)
   ⇒ (analysis_event ⇒ 'a abs_state ⇒ 'a abs_state)
```

```
   ⇒ ('x,'k,unit,'a abs_state lifted,'g::bounded_semilattice_sup_bot) dg_spec"
where
  "lifted_state_dg_spec 𝒢 empty_pred sk asn sp br bd rt en ev
     = dg_spec_of (lifted_state_local_spec 𝒢 empty_pred sk asn sp br bd rt en ev)"


lemma local_spec_step_transfer_lift:
  "local_spec_step (transfer_lift empty_pred sk)
     (λx e. transfer_lift empty_pred (asn x e))
     (λsc x. transfer_lift empty_pred (sp sc x))
     (λb pol. transfer_lift empty_pred (br b pol))
     (λp. transfer_lift empty_pred (bd p))
     (λe p. transfer_lift empty_pred (rt e p))
     (λevt. transfer_lift empty_pred (ev evt)) a
     = transfer_lift empty_pred (local_spec_step sk asn sp br bd rt ev a)"
  by (cases a) simp_all

lemma ls_step_lifted_state_local_spec [simp]:
  "ls_step (lifted_state_local_spec 𝒢 empty_pred sk asn sp br bd rt en ev) ch a
     = transfer_lift empty_pred (local_spec_step sk asn sp br bd rt ev a)"
  by (simp add: ls_step_def lifted_state_local_spec_def local_spec_step_transfer_lift)

lemma closed_step_lifted_state_local_spec [simp]:
  "⟦a⟧♯lifted_state_local_spec 𝒢 empty_pred sk asn sp br bd rt en ev
     = transfer_lift empty_pred (local_spec_step sk asn sp br bd rt ev a)"
  by (simp add: closed_step_def fun_eq_iff)

lemma dg_spec_step_lifted_state_dg_spec:
  "dg_spec_step (lifted_state_dg_spec 𝒢 empty_pred sk asn sp br bd rt en ev) a
     = local_transfer (transfer_lift empty_pred (local_spec_step sk asn sp br bd rt ev a))"
  by (simp add: lifted_state_dg_spec_def)

lemma dg_spec_wf_lifted_state_dg_spec [intro, simp]:
  "dg_spec_wf (lifted_state_dg_spec 𝒢 empty_pred sk asn sp br bd rt en ev)"
  by (simp add: lifted_state_dg_spec_def)

lemma dgs_enter_lifted_state_dg_spec:
  "enter♯ (lifted_state_dg_spec 𝒢 empty_pred sk asn sp br bd rt en ev) ci
     = local_enter_transfer (λd. [(d, transfer_lift empty_pred (en ci) d)])"
  by (simp add: lifted_state_dg_spec_def lifted_state_local_spec_def)
```

The caller continuation and the env stage are both identities, so the whole return pipeline is the return assignment applied to the raw call-site value and the callee exit.

```
lemma dg_spec_combine_transfer_lifted_state_dg_spec:
  "dg_spec_combine_transfer
     (lifted_state_dg_spec 𝒢 empty_pred sk asn sp br bd rt en ev) ci
     = local_combine_transfer
         (λdc de. transfer_lift2 empty_pred (combine♯ 𝒢 (ci_dst ci)) dc de)"
  by (simp add: lifted_state_dg_spec_def lifted_state_local_spec_def)

end
```

6.18 DG_Keyed_Split_Spec

```
theory DG_Keyed_Split_Spec
  imports DG_Local_State_Spec "Voblint_Solver.Strategy_Tree_Side_Buffering"
    "Voblint_Solver.Strategy_Program_Fold"
begin
```

One solver unknown per global name

The ownership-split lifter of `DG_Ownership_Split_Spec` keeps every program global in one shared value. Reading a global therefore depends on all of them, and writing one wakes every reader of any other. Goblint keys its globals by name instead, and this theory does the same for a whole-state component.

A global name `x` lives at its own unknown. Its value there is a global half that describes `x` alone, cut out of a whole state by `rg x`. A transfer reads only the names its edge mentions and publishes only the names its edge can write. Every other global it hands the wrapped transfer as `free`, a global half that claims nothing about the unread names, so the wrapped transfer stays sound on it. Its result says nothing useful about the unread names either, and it is not published: the next point reads them from their own unknowns again.

That last step is sound because the edge leaves the unpublished globals of a concrete store unchanged. Which names an edge may change is a fact of the concrete semantics (`edge_writes`), established in `CFG_Transfer`. Which names it reads only affects precision: a name left out is read as `free`.

Reading and publishing named globals

Reading a list of names folds their global halves into an accumulator, each value cut to the name it was read at. Publishing a list of names sends each its own cut of one whole state. The value-level counterparts are `view_of` and `pub_sides`.

```
fun read_view ::
  "('x,'k,vname,'d::semilattice_sup,'d) man ⇒ (vname ⇒ 'd ⇒ 'd) ⇒ vname list ⇒ 'd
   ⇒ ('x,'k,('d,'d) dg_state,'d) strategy_program"
where
  "read_view m rg [] acc = sp_return acc"
| "read_view m rg (x # xs) acc = man_global m x ≫= (λg. read_view m rg xs (rg x g ⊔ acc))"

fun publish_at ::
  "('x,'k,vname,'d,'d) man ⇒ (vname ⇒ 'd ⇒ 'd) ⇒ vname list ⇒ 'd
   ⇒ ('x,'k,('d,'d) dg_state,unit) strategy_program"
where
  "publish_at m rg [] r = sp_return ()"
| "publish_at m rg (x # xs) r = man_sideg m x (rg x r) ≫= (λ_. publish_at m rg xs r)"

definition view_of :: "(vname ⇒ 'd ⇒ 'd) ⇒ vname list ⇒ (vname ⇒ 'd) ⇒ 'd
    ⇒ 'd::semilattice_sup" where
  "view_of rg xs e acc = fold (λx acc. rg x (e x) ⊔ acc) xs acc"
```

The global half a whole environment describes, over a list of names.

```
definition full_view :: "(vname ⇒ 'd ⇒ 'd) ⇒ vname list ⇒ (vname ⇒ 'd)
    ⇒ 'd::bounded_semilattice_sup_bot" where
  "full_view rg xs e = view_of rg xs e bot"

definition pub_sides :: "(vname ⇒ 'k) ⇒ (vname ⇒ 'd ⇒ 'd) ⇒ vname list ⇒ 'd
    ⇒ 'x + 'k ⇒ ('d::bounded_semilattice_sup_bot, 'd) dg_state" where
  "pub_sides key rg xs r = foldr (λx acc. acc ⊔ bot(Inr (key x) := DG bot (rg x r))) xs bot"

lemma view_of_ge_acc: "acc ≤ view_of rg xs e acc"
  unfolding view_of_def
  by (induction xs arbitrary: acc) (auto intro: order_trans[OF sup_ge2])

lemma view_of_ge_mem: "x ∈ set xs ⟹ rg x (e x) ≤ view_of rg xs e acc"
  unfolding view_of_def
proof (induction xs arbitrary: acc)
  case Nil then show ?case by simp
next
  case (Cons y ys)
  show ?case
  proof (cases "x = y")
    case True
    then show ?thesis
      using view_of_ge_acc[of "rg y (e y) ⊔ acc" rg ys e]
      by (simp add: view_of_def)
  next
```

```
    case False
    with Cons show ?thesis by simp
  qed
qed

lemma view_of_le:
  assumes "acc ≤ v" and "⋀x. x ∈ set xs ⟹ rg x (e x) ≤ v"
  shows "view_of rg xs e acc ≤ v"
  using assms unfolding view_of_def by (induction xs arbitrary: acc) auto

lemma full_view_mono:
  assumes rg_mono: "⋀x v v'. v ≤ v' ⟹ rg x v ≤ rg x v'"
    and le: "⋀x. e x ≤ e' x"
  shows "full_view rg xs e ≤ full_view rg xs e'"
  unfolding full_view_def
proof (rule view_of_le)
  fix x assume "x ∈ set xs"
  then show "rg x (e x) ≤ view_of rg xs e' bot"
    using rg_mono[OF le[of x], of x] view_of_ge_mem[of x xs rg e' bot] by (blast intro: order_trans)
qed simp

lemma pub_sides_ge:
  "x ∈ set xs ⟹ DG bot (rg x r) ≤ pub_sides key rg xs r (Inr (key x))"
  unfolding pub_sides_def
  by (induction xs) (auto simp: sup_fun_def intro: le_supI1 le_supI2)
```

What a solver observes of them Reading names is observed as a dependency on each of their unknowns and nothing else; publishing names is observed as one side effect per name.

```
lemma traverse_read_view [simp]:
  "traverse_rhs (read_view (mk_dg_man d key) rg xs acc K) τ
     = traverse_rhs (K (view_of rg xs (genv key τ) acc)) τ"
  by (induction xs arbitrary: acc) (simp_all add: view_of_def sp_bind_def sp_return_def)

lemma sides_read_view [simp]:
  "sides_of_rhs (read_view (mk_dg_man d key) rg xs acc K) τ
     = sides_of_rhs (K (view_of rg xs (genv key τ) acc)) τ"
  by (induction xs arbitrary: acc) (simp_all add: view_of_def sp_bind_def sp_return_def)

lemma dep_aux_read_view [simp]:
  "dep_aux τ (read_view (mk_dg_man d key) rg xs acc K)
     = (Inr ∘ key) ‘ set xs ∪ dep_aux τ (K (view_of rg xs (genv key τ) acc))"
  by (induction xs arbitrary: acc) (auto simp: view_of_def sp_bind_def sp_return_def)

lemma traverse_publish_at [simp]:
  "traverse_rhs (publish_at (mk_dg_man d key) rg xs r K) τ = traverse_rhs (K ()) τ"
  by (induction xs) (simp_all add: sp_bind_def sp_return_def)

lemma sides_publish_at [simp]:
  "sides_of_rhs (publish_at (mk_dg_man d key) rg xs r K) τ
     = sides_of_rhs (K ()) τ ⊔ pub_sides key rg xs r"
  by (induction xs) (simp_all add: pub_sides_def sp_bind_def sp_return_def ac_simps)

lemma dep_aux_publish_at [simp]:
  "dep_aux τ (publish_at (mk_dg_man d key) rg xs r K) = dep_aux τ (K ())"
  by (induction xs) (simp_all add: sp_bind_def sp_return_def)

lemma side_path_read_view [simp]:
  "side_path τ (read_view (mk_dg_man d key) rg xs acc K)
     = side_path τ (K (view_of rg xs (genv key τ) acc))"
  by (induction xs arbitrary: acc)
     (simp_all add: view_of_def dg_read_global_def sp_read_global_def sp_bind_def sp_return_def)

lemma side_path_publish_at [simp]:
  "side_path τ (publish_at (mk_dg_man d key) rg xs r K)
     = map key xs @ side_path τ (K ())"
  by (induction xs) (simp_all add: dg_sideg_def sp_publish_def sp_bind_def sp_return_def)

lemma sp_wf_read_view [intro]:
  "(⋀x. sp_wf (man_global m x)) ⟹ sp_wf (read_view m rg xs acc)"
  by (induction xs arbitrary: acc) (auto intro!: sp_wf_bind)

lemma sp_wf_publish_at [intro]:
  "(⋀x v. sp_wf (man_sideg m x v)) ⟹ sp_wf (publish_at m rg xs r)"
  by (induction xs) (auto intro!: sp_wf_bind)
```

The keyed transfers

Each transfer reads its names, recombines them with the point's local half, runs the wrapped whole-state operation, publishes the names it may change and answers with the local half of the result. `cmb` and `rl` are the binary recombination and local projection the ownership split already uses.

```
definition keyed_transfer ::
  "('d ⇒ 'd ⇒ 'd) ⇒ ('d ⇒ 'd) ⇒ (vname ⇒ 'd ⇒ 'd) ⇒ (vname list ⇒ 'd)
   ⇒ vname list ⇒ vname list ⇒ ('d ⇒ 'd)
   ⇒ ('x,'k,vname,'d::bounded_semilattice_sup_bot,'d) man_transfer"
where
  "keyed_transfer cmb rl rg free R W f m =
     read_view m rg R (free R) ≫= (λg.
     publish_at m rg W (f (cmb (man_local m) g)) ≫= (λ_.
     sp_return (rl (f (cmb (man_local m) g)))))"

lemma dep_keyed_transfer:
  "dep_program τ (transfer_program (keyed_transfer cmb rl rg free R W f) src key)
     = insert src ((Inr ∘ key) ‘ set R)"
  by (simp add: dep_transfer_program keyed_transfer_def
      sp_compile_with_def sp_bind_def sp_return_def)

lemma side_path_keyed_transfer:
  "side_path τ (sp_compile
      (transfer_program (keyed_transfer cmb rl rg free R W f) src key)) = map key W"
  by (cases src)
     (simp_all add: transfer_program_def transfer_program_at_def keyed_transfer_def
        sp_compile_def sp_compile_with_def sp_map_def sp_bind_def sp_return_def
        dg_read_at_def sp_read_local_def sp_read_global_def)

definition keyed_combine_transfer ::
  "('d ⇒ 'd ⇒ 'd) ⇒ ('d ⇒ 'd) ⇒ (vname ⇒ 'd ⇒ 'd) ⇒ (vname list ⇒ 'd)
   ⇒ vname list ⇒ ('d ⇒ 'd ⇒ 'd)
   ⇒ ('x,'k,vname,'d::bounded_semilattice_sup_bot,'d) man_combine_transfer"
where
  "keyed_combine_transfer cmb rl rg free W h m de =
     publish_at m rg W (h (cmb (man_local m) (free [])) (cmb de (free []))) ≫= (λ_.
     sp_return (rl (h (cmb (man_local m) (free [])) (cmb de (free [])))))"

definition keyed_enter_transfer ::
  "('d ⇒ 'd ⇒ 'd) ⇒ ('d ⇒ 'd) ⇒ (vname ⇒ 'd ⇒ 'd) ⇒ (vname list ⇒ 'd)
   ⇒ vname list ⇒ vname list ⇒ ('d ⇒ 'd enter_result list)
   ⇒ ('x,'k,vname,'d::bounded_semilattice_sup_bot,'d) man_enter_transfer"
where
  "keyed_enter_transfer cmb rl rg free R W en m =
     read_view m rg R (free R) ≫= (λg.
     publish_at m rg W (⨆(c, e)←en (cmb (man_local m) g). e) ≫= (λ_.
     sp_return (map (λ(c, e). (rl c, rl e)) (en (cmb (man_local m) g)))))"
```

A call answers the alternatives its entry computes from the recombined state and publishes each name in `W` cut from the join of the entered states.

```
lemma enter_runs_keyed_enter_transfer:
  "enter_runs (keyed_enter_transfer cmb rl rg free R W en) (mk_dg_man d key) σ
     (map (λ(c, e). (rl c, rl e)) (en (cmb d (view_of rg R (genv key σ) (free R)))))
     (pub_sides key rg W
        (⨆(c, e)←en (cmb d (view_of rg R (genv key σ) (free R))). e))"
  unfolding enter_runs_def keyed_enter_transfer_def
  by (simp add: sp_bind_def sp_return_def ac_simps)

lemma enter_deps_keyed_enter_transfer:
  "enter_deps (keyed_enter_transfer cmb rl rg free R W en) (mk_dg_man d key) σ
     (map (λ(c, e). (rl c, rl e)) (en (cmb d (view_of rg R (genv key σ) (free R)))))
     ((Inr ∘ key) ‘ set R)"
  unfolding enter_deps_def keyed_enter_transfer_def
  by (simp add: sp_bind_def sp_return_def)
```

Only the manager's local value and its two global capabilities are read, so the query channel installed around a transfer does not change it.

```
lemma read_view_outer_man [simp]:
  "read_view (outer_man Q m) rg xs acc = read_view m rg xs acc"
  by (induction xs arbitrary: acc) simp_all

lemma publish_at_outer_man [simp]:
  "publish_at (outer_man Q m) rg xs r = publish_at m rg xs r"
  by (induction xs) simp_all
```

```
lemma keyed_transfer_outer_man [simp]:
  "keyed_transfer cmb rl rg free R W f (outer_man Q m) = keyed_transfer cmb rl rg free R W f m"
  by (simp add: keyed_transfer_def)
```

The lifter

Every edge runs the component's closed step through `keyed_transfer`, reading the globals its expressions mention and publishing the globals it may assign. A return reads nothing and may assign its destination. The return pipeline is wrapped once, as in the ownership split.

A call reads the globals its actuals mention and passes every other global as `free`. The context a call enters is computed from that state, and the analyzer recomputes it outside the solver from the same view of the solved environment.

```
definition keyed_split_spec ::
  "(vname ⇒ bool) ⇒ ('d ⇒ 'd ⇒ 'd) ⇒ ('d ⇒ 'd) ⇒ (vname ⇒ 'd ⇒ 'd)
   ⇒ (vname list ⇒ 'd) ⇒ 'd local_spec
   ⇒ ('x,'k,vname,'d::bounded_semilattice_sup_bot,'d) dg_spec"
where
  "keyed_split_spec 𝒢 cmb rl rg free c =
     (let step = (λa. keyed_transfer cmb rl rg free (edge_global_reads 𝒢 a)
                       (edge_global_writes 𝒢 a) ⟦a⟧♯c)
      in local_dg_spec_template⦇
         dgs_skip := step EA_Nop,
         dgs_assign := (λx e. step (EA_Assign x e)),
         dgs_special := (λsc x. step (EA_Special sc x)),
         dgs_branch := (λb pol. step (if pol then EA_Assume b else EA_AssumeNot b)),
         dgs_body := (λp. step (EA_Body p)),
         dgs_return := (λe p. step (EA_Ret e p)),
         dgs_enter := (λci. keyed_enter_transfer cmb rl rg free
            (call_global_reads 𝒢 (ci_args ci)) (global_names_in 𝒢 (ci_formals ci))
            (λd. ls_enter c (ls_channel c d) ci (d, d))),
         dgs_event := (λev. step (event_action ev)),
         dgs_combine_assign := (λci. keyed_combine_transfer cmb rl rg free
            (global_names_in 𝒢 (case_option [] (λx. [x]) (ci_dst ci)))
            (λdc de. ls_combine c (ls_channel c dc) (ls_channel c de) ci dc de)) ⦈)"


lemma dg_spec_step_keyed_split_spec [simp]:
  "dg_spec_step (keyed_split_spec 𝒢 cmb rl rg free c) a
     = keyed_transfer cmb rl rg free (edge_global_reads 𝒢 a) (edge_global_writes 𝒢 a)
         ⟦a⟧♯c"
  unfolding keyed_split_spec_def Let_def by (cases a) simp_all

lemma dgs_enter_keyed_split_spec [simp]:
  "enter♯ (keyed_split_spec 𝒢 cmb rl rg free c) ci
     = keyed_enter_transfer cmb rl rg free (call_global_reads 𝒢 (ci_args ci))
         (global_names_in 𝒢 (ci_formals ci)) (λd. ls_enter c (ls_channel c d) ci (d, d))"
  unfolding keyed_split_spec_def Let_def by simp

lemma dgs_query_keyed_split_spec [simp]:
  "dgs_query (keyed_split_spec 𝒢 cmb rl rg free c) m q = sp_return ⊤"
  unfolding keyed_split_spec_def Let_def by simp

lemma dg_spec_combine_transfer_keyed_split_spec [simp]:
  "dg_spec_combine_transfer (keyed_split_spec 𝒢 cmb rl rg free c) ci
     = keyed_combine_transfer cmb rl rg free (global_names_in 𝒢 (case_option [] (λx. [x]) (ci_dst ci)))
         (λdc de. ls_combine c (ls_channel c dc) (ls_channel c de) ci dc de)"
  unfolding dg_spec_combine_transfer_def keyed_split_spec_def Let_def
  by (simp add: local_combine_transfer_def fun_eq_iff)

lemma dg_spec_wf_keyed_split_spec [intro, simp]:
  "dg_spec_wf (keyed_split_spec 𝒢 cmb rl rg free c)"
  unfolding dg_spec_wf_def
  by (auto simp: keyed_transfer_def keyed_enter_transfer_def keyed_combine_transfer_def
      intro!: sp_wf_bind)
```

Soundness of the keyed lifter

A point's state is its local half recombined with the global half its environment describes over the program's global names `xs`. The carrier owes monotone operations, and that a name outside a read set is cut to something below `free` of that set. What it owes about concretization is one frame law, `mix`: a store described by a result, whose globals outside `W` agree with a store described against an environment, is described by the result's local half recombined with the result's own cut at the names in `W` and the environment at every other name.

```
theorem keyed_split_contract:
  fixes cmb :: "'d::bounded_semilattice_sup_bot ⇒ 'd ⇒ 'd"
  assumes sound: "sound_local_spec 𝒢 gm c"
    and cmb_mono: "⋀d d' g g'. d ≤ d' ⟹ g ≤ g' ⟹ cmb d g ≤ cmb d' g'"
    and rg_mono: "⋀x v v'. v ≤ v' ⟹ rg x v ≤ rg x v'"
    and rg_free: "⋀x R v. x ∉ set R ⟹ rg x v ≤ free R"
    and mix: "⋀l0 e r s s' W. s ∈ gm (cmb l0 (full_view rg xs e)) ⟹ s' ∈ gm r
        ⟹ (∀x. 𝒢 x ⟶ x ∉ set W ⟶ s' x = s x)
        ⟹ s' ∈ gm (cmb (rl r) (full_view rg xs (λx. if x ∈ set W then rg x r else e x)))"
  shows "analysis_contract (keyed_split_spec 𝒢 cmb rl rg free c)
          (λd e. gm (cmb d (full_view rg xs e))) 𝒢"
proof -
  have gm_mono_all: "∀x y. x ≤ y ⟶ gm x ⊆ gm y"
    using sound unfolding sound_local_spec_def by (rule conjunct1)
  then have gm_mono: "⋀x y. x ≤ y ⟹ gm x ⊆ gm y" by blast
  have env_mono: "⋀d d' e e'. d ≤ d' ⟹ (⋀x. e x ≤ e' x)
      ⟹ gm (cmb d (full_view rg xs e)) ⊆ gm (cmb d' (full_view rg xs e'))"
    by (rule gm_mono, rule cmb_mono, assumption, rule full_view_mono[OF rg_mono]) simp
  have view_ge: "full_view rg xs e ≤ view_of rg R e (free R)" for e R
    unfolding full_view_def
  proof (rule view_of_le)
    fix x assume "x ∈ set xs"
    show "rg x (e x) ≤ view_of rg R e (free R)"
    proof (cases "x ∈ set R")
      case True then show ?thesis by (rule view_of_ge_mem)
    next
      case False
      then show ?thesis using rg_free view_of_ge_acc by (blast intro: order_trans)
    qed
  qed simp
  have out: "(λx. if x ∈ set W then rg x r else e x) x ≤ (e ⊔ genv key P) x"
    if pub: "⋀x. x ∈ set W ⟹ rg x r ≤ dg_global (P (Inr (key x)))"
    for e W r x key and P :: "'z + 'w ⇒ ('d, 'd) dg_state"
    using pub[of x] by (auto intro: le_supI2)
  have pub: "rg x r ≤ dg_global ((bot ⊔ pub_sides key rg W r) (Inr (key x)))"
    if "x ∈ set W" for x W r key
    using pub_sides_ge[OF that, of rg r key] by (simp add: less_eq_dg_state_def)
  show ?thesis
  proof (unfold_locales, goal_cases wf mono step comb)
    case wf show ?case by (rule dg_spec_wf_keyed_split_spec)
  next
    case (mono d d' e e')
    then show ?case by (intro env_mono) (simp_all add: le_fun_def)
  next
    case (step a τ src key)
    let ?d = "dg_local (τ src)" and ?e = "genv key τ"
    let ?R = "edge_global_reads 𝒢 a" and ?W = "edge_global_writes 𝒢 a"
    let ?g = "view_of rg ?R ?e (free ?R)"
    let ?r = "⟦a⟧♯c (cmb ?d ?g)"
    have loc: "edge_out (keyed_split_spec 𝒢 cmb rl rg free c) a src key τ = rl ?r"
      by (simp add: dg_spec_edge_program_def traverse_transfer_program keyed_transfer_def
          sp_compile_with_def sp_bind_def sp_return_def)
    have sides: "sides_of_program
        (dg_spec_edge_program (keyed_split_spec 𝒢 cmb rl rg free c) a src key) τ
        = bot ⊔ pub_sides key rg ?W ?r"
      by (simp add: dg_spec_edge_program_def sides_transfer_program keyed_transfer_def
          sp_compile_with_def sp_bind_def sp_return_def bot_fun_def[symmetric])
    show ?case
    proof
      fix s' assume "s' ∈ edge_collect a (gm (cmb ?d (full_view rg xs ?e)))"
      then obtain s where s: "s ∈ gm (cmb ?d (full_view rg xs ?e))" and st: "s' ∈ edge_step a s"
        by (auto simp: edge_collect_def)
      have "s ∈ gm (cmb ?d ?g)"
        using s gm_mono[OF cmb_mono[OF order_refl view_ge]] by blast
      then have "s' ∈ edge_collect a (gm (cmb ?d ?g))" using st by (auto simp: edge_collect_def)
      then have r: "s' ∈ gm ?r" using closed_step_sound[OF sound] by blast
      have fr: "∀x. 𝒢 x ⟶ x ∉ set ?W ⟶ s' x = s x"
        using edge_step_frame[OF st] by auto
      have "s' ∈ gm (cmb (rl ?r) (full_view rg xs (λx. if x ∈ set ?W then rg x ?r else ?e x)))"
        by (rule mix[OF s r fr])
      also have "... ⊆ gm (cmb (rl ?r) (full_view rg xs (?e ⊔ genv key (bot ⊔ pub_sides key rg ?W ?r))))"
        by (rule env_mono[OF order_refl], rule out, erule pub)
      finally show "s' ∈ gm (cmb (edge_out (keyed_split_spec 𝒢 cmb rl rg free c) a src key τ)
          (full_view rg xs
            (genv key τ ⊔ edge_pub (keyed_split_spec 𝒢 cmb rl rg free c) a src key τ)))"
        unfolding loc sides .
```

```
    qed
  next
    case (comb s dc key τ t de ci)
    let ?e = "genv key τ" and ?W = "global_names_in 𝒢 (case_option [] (λx. [x]) (ci_dst ci))"
    let ?h = "λdc de. ls_combine c (ls_channel c dc) (ls_channel c de) ci dc de"
    let ?r = "?h (cmb dc (free [])) (cmb de (free []))"
    have free_ge: "full_view rg xs ?e ≤ free []"
      unfolding full_view_def by (rule view_of_le) (simp_all add: rg_free)
    have sc: "s ∈ gm (cmb dc (free []))"
      using comb(1) gm_mono[OF cmb_mono[OF order_refl free_ge]] by blast
    have tc: "t ∈ gm (cmb de (free []))"
      using comb(2) gm_mono[OF cmb_mono[OF order_refl free_ge]] by blast
    have r: "combine_collect 𝒢 (ci_dst ci) s t ∈ gm ?r"
      by (rule closed_combine_sound[OF sound sc tc])
    have fr: "∀x. 𝒢 x ⟶ x ∉ set ?W ⟶ combine_collect 𝒢 (ci_dst ci) s t x = t x"
      by (auto intro: combine_collect_frame)
    have "combine_collect 𝒢 (ci_dst ci) s t
       ∈ gm (cmb (rl ?r) (full_view rg xs (λx. if x ∈ set ?W then rg x ?r else ?e x)))"
      by (rule mix[OF comb(2) r fr])
    also have "... ⊆ gm (cmb (rl ?r) (full_view rg xs (?e ⊔ genv key (bot ⊔ pub_sides key rg ?W ?r))))"
      by (rule env_mono[OF order_refl], rule out, erule pub)
    finally show ?case
      by (simp add: keyed_combine_transfer_def sp_compile_with_def sp_bind_def sp_return_def
          bot_fun_def[symmetric])
  qed
qed

end
```

6.19 Analysis_Query

```
theory Analysis_Query
  imports "Voblint_Domain.Query_Lift" "Voblint_Domain.Int_Lattice"
begin

unbundle lattice_syntax
```

Queries one analysis answers for another

An analysis in a combination may ask its partners a question about the stores the current state describes, as a Goblint transfer asks `man.ask`. What an answer claims is fixed by one relation, `answer_holds q a s`: answer `a` to query `q` is true of store `s`. Answers form a meet-semilattice with top, as Goblint's per-query result lattices do (`queries.ml` at `5320a6b7`): $\top$ claims nothing, and $\sqcap$ combines what several analyses answered. The two laws below are all a combination needs to trust a combined answer; associativity and commutativity of the combination come from the class.

```
locale query_algebra =
  fixes answer_holds :: "'q ⇒ 'r::{semilattice_inf, order_top} ⇒ store ⇒ bool"
  assumes top_sound: "answer_holds q ⊤ s"
      and inf_sound: "⟦answer_holds q a s; answer_holds q b s⟧ ⟹ answer_holds q (a ⊓ b) s"
begin
```

A channel answers every query. It holds at a store when each of its answers is true there. Component transfers are proved against this predicate, never against a particular partner.

```
definition channel_holds :: "('q ⇒ 'r) ⇒ store ⇒ bool" where
  "channel_holds ch s ⟷ (∀q. answer_holds q (ch q) s)"

lemma channel_holdsD: "channel_holds ch s ⟹ answer_holds q (ch q) s"
  by (simp add: channel_holds_def)

lemma channel_holds_top [simp, intro]: "channel_holds (λ_. ⊤) s"
  by (simp add: channel_holds_def top_sound)

lemma channel_holds_inf [intro]:
  "channel_holds ch₁ s ⟹ channel_holds ch₂ s ⟹ channel_holds (λq. ch₁ q ⊓ ch₂ q) s"
  by (simp add: channel_holds_def inf_sound)

end
```

The value of an expression

Version 1 has one query kind, Goblint's `EvalInt`: which integers an expression may evaluate to (`queries.ml` line 108 at `0dc12d355`). Goblint answers it in `Lattice.Lift(IntDomTuple)`, and so does Voblint: the answer is an `int_dom` under `'a query_lift`. A comparison or logical operator evaluates to `0` or `1`, so its truth is the exact answer `1` or `0`; Goblint derives its former `MustBeEqual` and `MayBeLess` queries from `EvalInt` the same way (lines 528 to 539). The query is a constructor rather than a bare `exp` so that a later kind extends the datatype.

```
datatype query = EvalInt exp

type_synonym answer = "int_dom query_lift"

abbreviation gamma_answer :: "answer ⇒ int set" where
  "gamma_answer ≡ gamma_query_lift gamma_int_dom"
```

An answer claims that the expression evaluates into its concretization. `QTop` claims nothing: an analysis that does not understand the question declines with it. `QBot` admits no value, so a sound handler returns it only for a state that represents no concrete store. The meet combines two answers about one expression, and since the `int_dom` meet is exact, two sound analyses whose answers do not overlap describe no store in common.

```
fun eval_holds :: "query ⇒ answer ⇒ store ⇒ bool" where
  "eval_holds (EvalInt e) a s ⟷ ⟦e⟧ₑ s ∈ gamma_answer a"

interpretation eval_query: query_algebra eval_holds
proof
  fix q :: query and a b :: answer and s
  show "eval_holds q ⊤ s"
    by (cases q) simp
  show "eval_holds q a s ⟹ eval_holds q b s ⟹ eval_holds q (a ⊓ b) s"
    by (cases q) (simp add: gamma_query_lift_inf[OF gamma_inf_int_dom])
qed
```

```
declare eval_query.top_sound [simp]
```

The exact answer for a known integer, and a known integer read back from an answer, as Goblint's `ID.of_int` and `ID.to_int`. A consumer that reads `n` back may use it for the expression at every store the answer holds at.

```
definition answer_of_int :: "int ⇒ answer" where
  "answer_of_int n = QLifted (int_dom_of_int n)"

lemma gamma_answer_of_int [simp]:
  "gamma_answer (answer_of_int n) = {n}"
  by (simp add: answer_of_int_def)

fun answer_const :: "answer ⇒ int option" where
  "answer_const (QLifted d) = int_dom_constant d"
| "answer_const _ = None"

lemma eval_holds_constD:
  assumes "eval_holds (EvalInt e) a s" and "answer_const a = Some n"
  shows "⟦e⟧ₑ s = n"
  using assms gamma_int_dom_constant_subset
  by (cases a rule: answer_const.cases) auto

lemma answer_const_of_int [simp]: "answer_const (answer_of_int n) = Some n"
  by (simp add: answer_of_int_def int_dom_of_int_def int_dom_constant_def
      top_int_dom_ext_def)
```

An interval analysis knows only the interval component; the others stay at their top, so the answer claims exactly what the interval does.

```
definition answer_of_ivl :: "ivl ⇒ answer" where
  "answer_of_ivl i = QLifted ((⊤ :: int_dom)(|int_ivl := i|))"

lemma gamma_answer_of_ivl [simp]:
  "gamma_answer (answer_of_ivl i) = gamma_ivl i"
  by (simp add: answer_of_ivl_def top_int_dom_ext_def top_sign_def top_parity_def
      gamma_int_dom_def)

end
```

6.20 Local_Spec_Product

```
theory Local_Spec_Product
  imports "Voblint_Framework.DG_Spec_Sound"
begin
```

The product carrier

Two components run side by side on a pair of states. The pair is a datatype and not `'a × 'b`, for the reason `('l, 'g) dg_state` is: this repository loads `HOL-Library.Product_Lexorder`, so raw pairs already carry the lexicographic order, and the product needs the componentwise one. Every instance below, including the solver's widening and narrowing, is componentwise.

```
datatype ('a, 'b) analysis_product = Product (pleft: 'a) (pright: 'b)

instantiation analysis_product :: (ord, ord) ord
begin

definition less_eq_analysis_product ::
  "('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product ⇒ bool" where
  "less_eq_analysis_product p q ⟷ pleft p ≤ pleft q ∧ pright p ≤ pright q"

definition less_analysis_product ::
  "('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product ⇒ bool" where
  "less_analysis_product p q ⟷ p ≤ q ∧ ¬ q ≤ p"

instance ..

end

instance analysis_product :: (order, order) order
proof intro_classes
  fix x y z :: "('a, 'b) analysis_product"
  show "x < y ⟷ x ≤ y ∧ ¬ y ≤ x" by (simp add: less_analysis_product_def)
  show "x ≤ x" by (simp add: less_eq_analysis_product_def)
  show "x ≤ y ⟹ y ≤ z ⟹ x ≤ z"
    by (auto simp: less_eq_analysis_product_def intro: order_trans)
  show "x ≤ y ⟹ y ≤ x ⟹ x = y"
    by (auto simp: less_eq_analysis_product_def intro: analysis_product.expand antisym)
qed

instantiation analysis_product :: (semilattice_sup, semilattice_sup) semilattice_sup
begin

definition sup_analysis_product ::
  "('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product" where
  "sup_analysis_product p q = Product (pleft p ⊔ pleft q) (pright p ⊔ pright q)"

instance
  by standard (auto simp: less_eq_analysis_product_def sup_analysis_product_def)

end

instantiation analysis_product :: (order_bot, order_bot) order_bot
begin

definition bot_analysis_product :: "('a, 'b) analysis_product" where
  "bot_analysis_product = Product ⊥ ⊥"

instance
  by standard (simp add: less_eq_analysis_product_def bot_analysis_product_def)

end

instance analysis_product ::
  (bounded_semilattice_sup_bot, bounded_semilattice_sup_bot) bounded_semilattice_sup_bot ..

lemma Product_le_Product [simp]:
  "Product a b ≤ Product a' b' ⟷ a ≤ a' ∧ b ≤ b'"
  by (simp add: less_eq_analysis_product_def)

instantiation analysis_product ::
  ("{bounded_semilattice_sup_bot, warrowing}", "{bounded_semilattice_sup_bot, warrowing}") warrowing
begin

definition widen_analysis_product ::
```

```
  "('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product"
where
  "widen_analysis_product p q = Product ((pleft p) ∇ pleft q) ((pright p) ∇ pright q)"

definition narrow_analysis_product ::
  "('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product"
where
  "narrow_analysis_product p q =
     Product ((pleft p) Δ (pleft q)) ((pright p) Δ (pright q))"

instance
  by standard
    (simp_all add: less_eq_analysis_product_def widen_analysis_product_def
      narrow_analysis_product_def narrowing_class.narrow_ge narrowing_class.narrow_le
      widening_class.widen_ge1 widening_class.widen_ge2)

end


end
```

6.21 MCP_Spec

```
theory MCP_Spec
  imports "Voblint_Framework.DG_Spec_Sound" "Voblint_Domain.Reachability_Lift"
begin
```

Many cooperating analyses over one state

Goblint's `MCP` runs every activated analysis on its own part of one combined state (`mCP.ml` at `0dc12d355`). Voblint keeps the combined state as one type `'s`, a record with a field per registered analysis, and a component as operations on the whole record that touch only its own field. The activation list chooses which components run; the fields of inactive analyses are never read.

Every operation of a component receives the query channel of the manager Goblint hands it, `man.ask`, as a function `channel` from queries to answers: its query handler (`S.query man q`), its edge transfers, its entry (`S.enter man`) and both stages of its return. The return stages also receive the callee's channel, which Goblint's normal-call transfer in `constraints.ml` passes as `f_ask`. A handler may therefore ask further queries while it answers, and the channel that answers them is the one `ask_rec` below closes over the combined state, as Goblint's `MCP.query'` does.

The channel is a pure function of the state it describes. A component therefore neither reads nor publishes globals: the combined state covers analyses whose operations are local, and effectful analyses run as specifications of their own (`dgs_query` is the effectful handler).

A component holds only what runs. What its states describe, a set of stores, is not executable and is supplied beside it wherever soundness is stated, as `numeric_domain` keeps `gamma` out of `executable_domain`.

```
record 's local_spec =
  ls_query :: "channel ⇒ 's ⇒ channel"
  ls_skip :: "channel ⇒ 's ⇒ 's"
  ls_assign :: "channel ⇒ vname ⇒ exp ⇒ 's ⇒ 's"
  ls_special :: "channel ⇒ special_call ⇒ vname ⇒ 's ⇒ 's"
  ls_branch :: "channel ⇒ exp ⇒ bool ⇒ 's ⇒ 's"
  ls_body :: "channel ⇒ pname ⇒ 's ⇒ 's"
  ls_return :: "channel ⇒ exp option ⇒ pname ⇒ 's ⇒ 's"
  ls_event :: "channel ⇒ analysis_event ⇒ 's ⇒ 's"
  ls_enter :: "channel ⇒ call_info ⇒ 's × 's ⇒ ('s × 's) list"
  ls_combine_env :: "channel ⇒ channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's"
  ls_combine_assign :: "channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's"
```

The edge transfers are separate fields, one per kind of edge, as Goblint's `Spec` has `skip`, `assign`, `special`, `branch`, `body`, `return` and `event`. A component that overrides one of them updates that field alone. The step on an arbitrary edge is derived from them.

```
definition ls_step :: "'s local_spec ⇒ channel ⇒ edge_action ⇒ 's ⇒ 's" where
  "ls_step c ch = local_spec_step (ls_skip c ch) (ls_assign c ch) (ls_special c ch)
     (ls_branch c ch) (ls_body c ch) (ls_return c ch) (ls_event c ch)"

lemma ls_step_simps [simp]:
  "ls_step c ch EA_Nop = ls_skip c ch"
  "ls_step c ch (EA_Assign x e) = ls_assign c ch x e"
  "ls_step c ch (EA_Special sc x) = ls_special c ch sc x"
  "ls_step c ch (EA_Assume b) = ls_branch c ch b True"
  "ls_step c ch (EA_AssumeNot b) = ls_branch c ch b False"
  "ls_step c ch (EA_Body p) = ls_body c ch p"
  "ls_step c ch (EA_Ret r p) = ls_return c ch r p"
  "ls_step c ch (EA_Check l cnd) = ls_event c ch (Check_Event l cnd)"
  by (simp_all add: ls_step_def)

lemma ls_step_event_action [simp]: "ls_step c ch (event_action e) = ls_event c ch e"
  by (cases e) simp

lemma ls_step_update_other [simp]:
  "ls_step (c(|ls_query := h|)) = ls_step c"
  "ls_step (c(|ls_enter := e|)) = ls_step c"
  "ls_step (c(|ls_combine_env := ce|)) = ls_step c"
  "ls_step (c(|ls_combine_assign := ca|)) = ls_step c"
  by (simp_all add: ls_step_def fun_eq_iff)
```

A component whose edge transfers are one function of the edge, such as a lens onto a field or a fold over several components, is built from that function: each field is the function at its kind of edge.

```
definition make_local_spec ::
  "(channel ⇒ 's ⇒ channel) ⇒ (channel ⇒ edge_action ⇒ 's ⇒ 's)
   ⇒ (channel ⇒ call_info ⇒ 's × 's ⇒ ('s × 's) list)
```

```
   ⇒ (channel ⇒ channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's)
   ⇒ (channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's) ⇒ 's local_spec" where
  "make_local_spec qry st en ce ca = (|
     ls_query = qry,
     ls_skip = (λch. st ch EA_Nop),
     ls_assign = (λch x e. st ch (EA_Assign x e)),
     ls_special = (λch sc x. st ch (EA_Special sc x)),
     ls_branch = (λch b pol. st ch (if pol then EA_Assume b else EA_AssumeNot b)),
     ls_body = (λch p. st ch (EA_Body p)),
     ls_return = (λch r p. st ch (EA_Ret r p)),
     ls_event = (λch ev. st ch (event_action ev)),
     ls_enter = en, ls_combine_env = ce, ls_combine_assign = ca |)"

lemma make_local_spec_sel [simp]:
  "ls_query (make_local_spec qry st en ce ca) = qry"
  "ls_step (make_local_spec qry st en ce ca) = st"
  "ls_enter (make_local_spec qry st en ce ca) = en"
  "ls_combine_env (make_local_spec qry st en ce ca) = ce"
  "ls_combine_assign (make_local_spec qry st en ce ca) = ca"
proof -
  have "ls_step (make_local_spec qry st en ce ca) ch a = st ch a" for ch a
  proof (cases a)
    case (EA_Check l cnd)
    then show ?thesis by (simp add: make_local_spec_def ls_step_def)
  qed (simp_all add: make_local_spec_def ls_step_def)
  then show "ls_step (make_local_spec qry st en ce ca) = st" by (simp add: fun_eq_iff)
qed (simp_all add: make_local_spec_def)
```

The return runs in the two stages of Goblint's `combine_env` and `combine_assign`, which both receive the callee's channel. Only the first receives the caller's: Goblint's `combine_assign` asks about the state `combine_env` produced, which describes no concrete store on its own here, since soundness and independence speak only about the composite.

```
abbreviation ls_combine ::
  "'s local_spec ⇒ channel ⇒ channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's" where
  "ls_combine c ch ch' ci x de ≡ ls_combine_assign c ch' ci (ls_combine_env c ch ch' ci x de) de"
```

A local specification is sound for a concretization `gamma` when each operation covers the concrete behaviour, whatever the other fields of the record hold. Every operation is proved against every channel that holds at the stores it is asked about, so its proof cannot depend on who answers: the caller's store for a step, an entry, a query and the first stage of a return, and the callee's exit store for the callee's channel.

```
definition sound_local_spec ::
  "(vname ⇒ bool) ⇒ ('s::order ⇒ store set) ⇒ 's local_spec ⇒ bool" where
  "sound_local_spec 𝒢 gm c ⟷
     (∀x y. x ≤ y ⟶ gm x ⊆ gm y)
     ∧ (∀ch a x. edge_collect a (gm x ∩ Collect (eval_query.channel_holds ch))
                 ⊆ gm (ls_step c ch a x))
     ∧ (∀ch s ci p. s ∈ gm (fst p) ⟶ eval_query.channel_holds ch s ⟶
          (∃q ∈ set (ls_enter c ch ci p). s ∈ gm (fst q)
             ∧ call_enter 𝒢 (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s
                 ∈ gm (snd q)))
     ∧ (∀ch ch' s t ci x de. s ∈ gm x ⟶ eval_query.channel_holds ch s ⟶
          t ∈ gm de ⟶ eval_query.channel_holds ch' t ⟶
          combine_collect 𝒢 (ci_dst ci) s t ∈ gm (ls_combine c ch ch' ci x de))
     ∧ (∀ch s x q. s ∈ gm x ⟶ eval_query.channel_holds ch s ⟶
          eval_holds q (ls_query c ch x q) s)"
```

The step obligation splits into one law per kind of edge, each about the field that handles it: the store the edge produces lies in what the field returns, for every store and channel that hold before. A component proves the laws of the fields it defines, and a field it replaces needs only its own law again (`sound_local_spec_update`).

```
definition sound_skip :: "('s ⇒ store set) ⇒ (channel ⇒ 's ⇒ 's) ⇒ bool" where
  "sound_skip gm f ⟷
     (∀ch x s. s ∈ gm x ⟶ eval_query.channel_holds ch s ⟶ s ∈ gm (f ch x))"

definition sound_assign ::
  "('s ⇒ store set) ⇒ (channel ⇒ vname ⇒ exp ⇒ 's ⇒ 's) ⇒ bool" where
  "sound_assign gm f ⟷
     (∀ch x s y e. s ∈ gm x ⟶ eval_query.channel_holds ch s ⟶ s(y := ⟦e⟧ₑ s) ∈ gm (f ch y e x))"

definition sound_special ::
  "('s ⇒ store set) ⇒ (channel ⇒ special_call ⇒ vname ⇒ 's ⇒ 's) ⇒ bool" where
  "sound_special gm f ⟷
     (∀ch x s sc y t. s ∈ gm x ⟶ eval_query.channel_holds ch s ⟶ t ∈ special_step sc y s
        ⟶ t ∈ gm (f ch sc y x))"
```

```
definition sound_branch :: "('s ⇒ store set) ⇒ (channel ⇒ exp ⇒ bool ⇒ 's ⇒ 's) ⇒ bool" where
  "sound_branch gm f ⟷
     (∀ch x s b pol. s ∈ gm x ⟶ eval_query.channel_holds ch s ⟶ truthy (⟦b⟧e s) = pol
        ⟶ s ∈ gm (f ch b pol x))"

definition sound_body :: "('s ⇒ store set) ⇒ (channel ⇒ pname ⇒ 's ⇒ 's) ⇒ bool" where
  "sound_body gm f ⟷
     (∀ch x s p. s ∈ gm x ⟶ eval_query.channel_holds ch s ⟶ s ∈ gm (f ch p x))"

definition sound_return ::
  "('s ⇒ store set) ⇒ (channel ⇒ exp option ⇒ pname ⇒ 's ⇒ 's) ⇒ bool" where
  "sound_return gm f ⟷
     (∀ch x s r p. s ∈ gm x ⟶ eval_query.channel_holds ch s
        ⟶ s(ret_var := (case r of None ⇒ s ret_var | Some a ⇒ ⟦a⟧e s)) ∈ gm (f ch r p x))"

definition sound_event :: "('s ⇒ store set) ⇒ (channel ⇒ analysis_event ⇒ 's ⇒ 's) ⇒ bool" where
  "sound_event gm f ⟷
     (∀ch x s ev. s ∈ gm x ⟶ eval_query.channel_holds ch s ⟶ s ∈ gm (f ch ev x))"
```

The laws of the other fields. The handler's answers hold at every store its state describes. An entry covers the caller's store and the store the call enters by one of its pairs. The two return stages carry one law together, because only their composition describes a concrete store.

```
definition sound_query :: "('s ⇒ store set) ⇒ (channel ⇒ 's ⇒ channel) ⇒ bool" where
  "sound_query gm h ⟷
     (∀ch s x q. s ∈ gm x ⟶ eval_query.channel_holds ch s ⟶ eval_holds q (h ch x q) s)"

definition sound_enter ::
  "(vname ⇒ bool) ⇒ ('s ⇒ store set)
   ⇒ (channel ⇒ call_info ⇒ 's × 's ⇒ ('s × 's) list) ⇒ bool" where
  "sound_enter 𝒢 gm en ⟷
     (∀ch s ci p. s ∈ gm (fst p) ⟶ eval_query.channel_holds ch s ⟶
        (∃q ∈ set (en ch ci p). s ∈ gm (fst q)
           ∧ call_enter 𝒢 (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s ∈ gm (snd q)))"

definition sound_combine ::
  "(vname ⇒ bool) ⇒ ('s ⇒ store set)
   ⇒ (channel ⇒ channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's)
   ⇒ (channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's) ⇒ bool" where
  "sound_combine 𝒢 gm ce ca ⟷
     (∀ch ch' s t ci x de. s ∈ gm x ⟶ eval_query.channel_holds ch s ⟶
        t ∈ gm de ⟶ eval_query.channel_holds ch' t ⟶
        combine_collect 𝒢 (ci_dst ci) s t ∈ gm (ca ch' ci (ce ch ch' ci x de) de))"

theorem ls_step_sound_iff:
  "(∀ch a x. edge_collect a (gm x ∩ Collect (eval_query.channel_holds ch)) ⊆ gm (ls_step c ch a x))
   ⟷ sound_skip gm (ls_skip c) ∧ sound_assign gm (ls_assign c) ∧ sound_special gm (ls_special c)
     ∧ sound_branch gm (ls_branch c) ∧ sound_body gm (ls_body c)
     ∧ sound_return gm (ls_return c) ∧ sound_event gm (ls_event c)"
  (is "?step ⟷ ?fields")
proof
  assume step: ?step
  have at: "t ∈ gm (ls_step c ch a x)"
    if "s ∈ gm x" "eval_query.channel_holds ch s" "t ∈ edge_step a s" for ch a x s t
    using step that unfolding edge_collect_def by blast
  have pol: "s ∈ gm (ls_branch c ch b pol x)"
    if "s ∈ gm x" "eval_query.channel_holds ch s" "truthy (⟦b⟧e s) = pol" for ch b pol x s
  proof (cases pol)
    case True
    then show ?thesis
      using at[OF that(1,2), where a = "EA_Assume b" and t = s] that(3) by simp
  next
    case False
    then show ?thesis
      using at[OF that(1,2), where a = "EA_AssumeNot b" and t = s] that(3) by simp
  qed
  have ev: "s ∈ gm (ls_event c ch ev x)"
    if "s ∈ gm x" "eval_query.channel_holds ch s" for ch ev x s
  proof (cases ev)
    case (Check_Event l e)
    then show ?thesis using at[OF that, where a = "EA_Check l e" and t = s] by simp
  qed
  have special: "t ∈ gm (ls_special c ch sc y x)"
    if "s ∈ gm x" "eval_query.channel_holds ch s" "t ∈ special_step sc y s" for ch x s sc y t
    using at[OF that(1,2), where a = "EA_Special sc y" and t = t] that(3)
    by (simp only: edge_step.simps ls_step_simps)
```

```
  show ?fields
    unfolding sound_skip_def sound_assign_def sound_special_def sound_branch_def sound_body_def
      sound_return_def sound_event_def
    using at[where a = EA_Nop] at[where a = "EA_Assign _ _"]
      at[where a = "EA_Body _"] at[where a = "EA_Ret _ _"] pol ev special
    by (fastforce+)
next
  assume F: ?fields
  show ?step
  proof (intro allI subsetI)
    fix ch a x t
    assume "t ∈ edge_collect a (gm x ∩ Collect (eval_query.channel_holds ch))"
    then obtain s where s: "s ∈ gm x" "eval_query.channel_holds ch s" "t ∈ edge_step a s"
      unfolding edge_collect_def by blast
    show "t ∈ gm (ls_step c ch a x)"
    proof (cases a)
      case (EA_Special sc y)
      have "t ∈ special_step sc y s" using s(3) EA_Special by (simp only: edge_step.simps)
      then show ?thesis
        using F s(1,2) EA_Special unfolding sound_special_def by (simp only: ls_step_simps)
    next
      case (EA_Assume b)
      then have "t = s" "truthy (⟦b⟧e s) = True" using s(3) by (simp_all split: if_splits)
      then show ?thesis
        using F s(1,2) EA_Assume unfolding sound_branch_def by (simp only: ls_step_simps)
    next
      case (EA_AssumeNot b)
      then have "t = s" "truthy (⟦b⟧e s) = False" using s(3) by (simp_all split: if_splits)
      then show ?thesis
        using F s(1,2) EA_AssumeNot unfolding sound_branch_def by (simp only: ls_step_simps)
    qed (use F s in ‹auto simp: sound_skip_def sound_assign_def sound_body_def sound_return_def
                    sound_event_def›)
  qed
qed
```

The certificate is exactly the conjunction of a monotone concretization and one law per field, the two return stages sharing one law. A local specification is therefore built, taken apart and changed field by field.

```
theorem sound_local_spec_iff:
  "sound_local_spec 𝒢 gm c ⟷
     (∀x y. x ≤ y ⟶ gm x ⊆ gm y)
     ∧ sound_query gm (ls_query c)
     ∧ sound_skip gm (ls_skip c) ∧ sound_assign gm (ls_assign c)
     ∧ sound_special gm (ls_special c) ∧ sound_branch gm (ls_branch c)
     ∧ sound_body gm (ls_body c) ∧ sound_return gm (ls_return c)
     ∧ sound_event gm (ls_event c)
     ∧ sound_enter 𝒢 gm (ls_enter c)
     ∧ sound_combine 𝒢 gm (ls_combine_env c) (ls_combine_assign c)"
  unfolding sound_local_spec_def ls_step_sound_iff sound_query_def sound_enter_def
    sound_combine_def
  by (intro iffI; elim conjE; intro conjI; assumption)

lemma sound_local_specI:
  assumes "⋀x y. x ≤ y ⟹ gm x ⊆ gm y"
    and "sound_query gm (ls_query c)"
    and "sound_skip gm (ls_skip c)" and "sound_assign gm (ls_assign c)"
    and "sound_special gm (ls_special c)" and "sound_branch gm (ls_branch c)"
    and "sound_body gm (ls_body c)" and "sound_return gm (ls_return c)"
    and "sound_event gm (ls_event c)"
    and "sound_enter 𝒢 gm (ls_enter c)"
    and "sound_combine 𝒢 gm (ls_combine_env c) (ls_combine_assign c)"
  shows "sound_local_spec 𝒢 gm c"
  using assms unfolding sound_local_spec_iff by blast

lemma sound_local_specD:
  assumes "sound_local_spec 𝒢 gm c"
  shows sound_local_spec_monoD: "x ≤ y ⟹ gm x ⊆ gm y"
    and sound_local_spec_queryD: "sound_query gm (ls_query c)"
    and sound_local_spec_skipD: "sound_skip gm (ls_skip c)"
    and sound_local_spec_assignD: "sound_assign gm (ls_assign c)"
    and sound_local_spec_specialD: "sound_special gm (ls_special c)"
    and sound_local_spec_branchD: "sound_branch gm (ls_branch c)"
    and sound_local_spec_bodyD: "sound_body gm (ls_body c)"
    and sound_local_spec_returnD: "sound_return gm (ls_return c)"
    and sound_local_spec_eventD: "sound_event gm (ls_event c)"
    and sound_local_spec_enterD: "sound_enter 𝒢 gm (ls_enter c)"
    and sound_local_spec_combineD:
```

```
    "sound_combine 𝒢 gm (ls_combine_env c) (ls_combine_assign c)"
  using assms unfolding sound_local_spec_iff by blast+
```

A sound local specification stays sound when one field is replaced by one satisfying that field's law. A return stage is replaced together with the law of the composed return it forms with the other stage.

```
lemma sound_local_spec_update:
  assumes "sound_local_spec 𝒢 gm c"
  shows sound_local_spec_update_query:
      "sound_query gm h ⟹ sound_local_spec 𝒢 gm (c(|ls_query := h|))"
    and sound_local_spec_update_skip:
      "sound_skip gm sk ⟹ sound_local_spec 𝒢 gm (c(|ls_skip := sk|))"
    and sound_local_spec_update_assign:
      "sound_assign gm asn ⟹ sound_local_spec 𝒢 gm (c(|ls_assign := asn|))"
    and sound_local_spec_update_special:
      "sound_special gm sp ⟹ sound_local_spec 𝒢 gm (c(|ls_special := sp|))"
    and sound_local_spec_update_branch:
      "sound_branch gm br ⟹ sound_local_spec 𝒢 gm (c(|ls_branch := br|))"
    and sound_local_spec_update_body:
      "sound_body gm bd ⟹ sound_local_spec 𝒢 gm (c(|ls_body := bd|))"
    and sound_local_spec_update_return:
      "sound_return gm rt ⟹ sound_local_spec 𝒢 gm (c(|ls_return := rt|))"
    and sound_local_spec_update_event:
      "sound_event gm ev ⟹ sound_local_spec 𝒢 gm (c(|ls_event := ev|))"
    and sound_local_spec_update_enter:
      "sound_enter 𝒢 gm en ⟹ sound_local_spec 𝒢 gm (c(|ls_enter := en|))"
    and sound_local_spec_update_combine_env:
      "sound_combine 𝒢 gm ce (ls_combine_assign c)
       ⟹ sound_local_spec 𝒢 gm (c(|ls_combine_env := ce|))"
    and sound_local_spec_update_combine_assign:
      "sound_combine 𝒢 gm (ls_combine_env c) ca
       ⟹ sound_local_spec 𝒢 gm (c(|ls_combine_assign := ca|))"
  using assms unfolding sound_local_spec_iff by simp_all
```

Conservative defaults

A field whose concrete counterpart leaves the store unchanged has a default that decides nothing: the handler answers $\top$, and skip, body, event and branch keep the state. Each is sound for every concretization. Assign, special, return, entry and the second return stage change the store or the frame, so no default is sound for every concretization and an analysis supplies them. The first return stage defaults to the caller's state; it has no law of its own, only the composed return does.

```
definition query_unknown :: "channel ⇒ 's ⇒ channel" where
  "query_unknown ch x = ⊤"

definition skip_identity :: "channel ⇒ 's ⇒ 's" where
  "skip_identity ch x = x"

definition body_identity :: "channel ⇒ pname ⇒ 's ⇒ 's" where
  "body_identity ch p x = x"

definition event_identity :: "channel ⇒ analysis_event ⇒ 's ⇒ 's" where
  "event_identity ch ev x = x"

definition branch_identity :: "channel ⇒ exp ⇒ bool ⇒ 's ⇒ 's" where
  "branch_identity ch b pol x = x"

definition combine_env_identity :: "channel ⇒ channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's" where
  "combine_env_identity ch ch' ci x de = x"

lemma sound_query_unknown: "sound_query gm query_unknown"
  by (simp add: sound_query_def query_unknown_def)

lemma sound_skip_identity: "sound_skip gm skip_identity"
  by (simp add: sound_skip_def skip_identity_def)

lemma sound_body_identity: "sound_body gm body_identity"
  by (simp add: sound_body_def body_identity_def)

lemma sound_event_identity: "sound_event gm event_identity"
  by (simp add: sound_event_def event_identity_def)

lemma sound_branch_identity: "sound_branch gm branch_identity"
  by (simp add: sound_branch_def branch_identity_def)

lemma sound_combine_env_identity:
```

```
  "sound_combine 𝒢 gm combine_env_identity ca ⟷
     (∀ch' s t ci x de. s ∈ gm x ⟶ t ∈ gm de ⟶ eval_query.channel_holds ch' t ⟶
        combine_collect 𝒢 (ci_dst ci) s t ∈ gm (ca ch' ci x de))"
  unfolding sound_combine_def combine_env_identity_def
  using eval_query.channel_holds_top by blast
```

A local specification that supplies only the five operations without a default. An analysis starts from it and overrides the fields it decides more precisely; each override needs only its own law again (`sound_local_spec_update`).

```
definition conservative_local_spec ::
  "(channel ⇒ vname ⇒ exp ⇒ 's ⇒ 's)
   ⇒ (channel ⇒ special_call ⇒ vname ⇒ 's ⇒ 's)
   ⇒ (channel ⇒ exp option ⇒ pname ⇒ 's ⇒ 's)
   ⇒ (channel ⇒ call_info ⇒ 's × 's ⇒ ('s × 's) list)
   ⇒ (channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's) ⇒ 's local_spec" where
  "conservative_local_spec asn sp rt en ca = (|
     ls_query = query_unknown, ls_skip = skip_identity, ls_assign = asn,
     ls_special = sp, ls_branch = branch_identity, ls_body = body_identity,
     ls_return = rt, ls_event = event_identity, ls_enter = en,
     ls_combine_env = combine_env_identity, ls_combine_assign = ca |)"

theorem sound_conservative_local_spec:
  assumes "⋀x y. x ≤ y ⟹ gm x ⊆ gm y"
    and "sound_assign gm asn" and "sound_special gm sp" and "sound_return gm rt"
    and "sound_enter 𝒢 gm en" and "sound_combine 𝒢 gm combine_env_identity ca"
  shows "sound_local_spec 𝒢 gm (conservative_local_spec asn sp rt en ca)"
  by (rule sound_local_specI)
     (simp_all add: conservative_local_spec_def assms sound_query_unknown sound_skip_identity
        sound_body_identity sound_event_identity sound_branch_identity)
```

A component leaves a concretization alone when none of its operations changes what that concretization says. For components that each own one field of a record this is the lens law that updating one field leaves the others.

```
definition mcp_frame :: "'s local_spec ⇒ ('s ⇒ store set) ⇒ bool" where
  "mcp_frame c gm ⟷
     (∀ch a x. gm (ls_step c ch a x) = gm x)
     ∧ (∀ch ci p q. q ∈ set (ls_enter c ch ci p) ⟶
          gm (fst q) = gm (fst p) ∧ gm (snd q) = gm (snd p))
     ∧ (∀ch ch' ci x de. gm (ls_combine c ch ch' ci x de) = gm x)"
```

Cooperating analyses come as pairs of a concretization and a component. They are independent when each leaves the others' concretizations alone.

```
type_synonym 's certified_component = "('s ⇒ store set) × 's local_spec"

fun mcp_independent :: "'s certified_component list ⇒ bool" where
  "mcp_independent [] ⟷ True"
| "mcp_independent ((g, c) # gcs) ⟷
     (∀(g', c') ∈ set gcs. mcp_frame c g' ∧ mcp_frame c' g) ∧ mcp_independent gcs"
```

The query channel

A handler answers a query given the channel through which it may ask further ones. `ask_rec` closes that recursion over one state, as Goblint's `MCP.query'` does: a query already being asked answers ⊤ instead of recursing, and every other query is answered by the handler under a channel that remembers it. The depth bound plays the role it plays in `ask_with`: queries range over an infinite type, so cycle detection alone does not make the recursion total, and exhausting the bound aborts the generated program. Logically the fallback is ⊤, so the bound plays no part in soundness. Goblint also caches answers per transfer; that changes only how often a handler runs.

```
fun ask_rec :: "(channel ⇒ 's ⇒ channel) ⇒ nat ⇒ query set ⇒ 's ⇒ channel" where
  "ask_rec H 0 asked x q =
     Code.abort (STR ''query recursion exceeded query_depth'') (λ_. ⊤)"
| "ask_rec H (Suc n) asked x q =
     (if q ∈ asked then ⊤ else H (ask_rec H n (insert q asked) x) x q)"
```

A handler that answers soundly under every sound channel makes the closed channel sound: by induction on the depth, each level's channel holds, so the handler's answer under it holds.

```
lemma ask_rec_sound:
  assumes "⋀ch q. eval_query.channel_holds ch s ⟹ eval_holds q (H ch x q) s"
  shows "eval_query.channel_holds (ask_rec H n asked x) s"
proof (induction n arbitrary: asked)
  case 0
  then show ?case by (simp add: eval_query.channel_holds_def)
next
```

```
  case (Suc n)
  then show ?case
    using assms by (auto simp: eval_query.channel_holds_def)
qed

definition ls_channel :: "'s local_spec ⇒ 's ⇒ channel" where
  "ls_channel c x = ask_rec (ls_query c) query_depth {} x"

lemma ls_channel_sound:
  assumes "sound_local_spec 𝒢 γ_c c" and "s ∈ γ_c x"
  shows "eval_query.channel_holds (ls_channel c x) s"
  unfolding ls_channel_def
  by (rule ask_rec_sound) (use assms in ‹simp add: sound_local_spec_def›)

lemma ls_channel_const: "ls_query c = (λch x. h x) ⟹ ls_channel c = h"
  by (simp add: ls_channel_def query_depth_def fun_eq_iff eval_nat_numeral)
```

The combined operations

The combined operations run the active components in turn. Soundness needs only `mcp_frame`: a component keeps what the other components' states describe, not their representation, so the soundness proof says nothing about the order of the fold. A component that updates only its own field of a product does leave the other fields unchanged, and for those the order does not matter; Goblint runs every component on the predecessor's part of the state, which is the same computation. Every component gets the same channel, as every Goblint component gets the same `man.ask`.

```
definition mcp_step :: "'s local_spec list ⇒ channel ⇒ edge_action ⇒ 's ⇒ 's" where
  "mcp_step cs ch a x = fold (λc y. ls_step c ch a y) cs x"

definition mcp_en_from ::
  "'s local_spec list ⇒ channel ⇒ call_info ⇒ 's × 's ⇒ 's enter_result list" where
  "mcp_en_from cs ch ci p = fold (λc ps. concat (map (ls_enter c ch ci) ps)) cs [p]"

definition mcp_comb ::
  "'s local_spec list ⇒ channel ⇒ channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's" where
  "mcp_comb cs ch ch' ci x de = fold (λc y. ls_combine c ch ch' ci y de) cs x"
```

The combined handler meets every component's answer, each given the same channel, as `MCP.query'` meets the results of all analyses.

```
definition mcp_qry :: "'s local_spec list ⇒ channel ⇒ 's ⇒ channel" where
  "mcp_qry cs ch x q = fold (λc r. r ⊓ ls_query c ch x q) cs ⊤"

definition mcp_gamma :: "('s ⇒ store set) list ⇒ 's ⇒ store set" where
  "mcp_gamma Γ x = (⋂γ_i ∈ set Γ. γ_i x)"
```

Soundness of the combination

The combined state is sound for the intersection `mcp_gamma` of the components' concretizations. Step, combine, entry and query folds each stay sound because a component leaves the others' concretizations unchanged.

```
lemma mcp_gamma_Cons [simp]: "mcp_gamma (g # gs) x = g x ∩ mcp_gamma gs x"
  by (simp add: mcp_gamma_def)

lemma mcp_gamma_Nil [simp]: "mcp_gamma [] x = UNIV"
  by (simp add: mcp_gamma_def)
```

Components a fold runs leave an independent concretization as it was.

```
lemma gamma_fold_step:
  "(∀c ∈ set cs. mcp_frame c g) ⟹ g (fold (λc y. ls_step c ch a y) cs x) = g x"
  by (induction cs arbitrary: x) (simp_all add: mcp_frame_def)

lemma gamma_fold_comb:
  "(∀c ∈ set cs. mcp_frame c g) ⟹ g (fold (λc y. ls_combine c ch ch' ci y de) cs x) = g x"
  by (induction cs arbitrary: x) (simp_all add: mcp_frame_def)

lemma mcp_gamma_frame:
  "(∀g ∈ set gs. mcp_frame c g) ⟹ mcp_gamma gs (ls_step c ch a x) = mcp_gamma gs x"
  "(∀g ∈ set gs. mcp_frame c g) ⟹ mcp_gamma gs (ls_combine c ch ch' ci x de) = mcp_gamma gs x"
  by (auto simp: mcp_gamma_def mcp_frame_def intro!: INF_cong)

lemma mcp_independent_Cons_frames:
  assumes "mcp_independent ((g, c) # gcs)"
  shows "∀g' ∈ fst ‘ set gcs. mcp_frame c g'" "∀c' ∈ snd ‘ set gcs. mcp_frame c' g"
```

```
    "mcp_independent gcs"
  using assms by auto

lemma mcp_step_sound:
  assumes "∀(g, c) ∈ set gcs. sound_local_spec 𝒢 g c" and "mcp_independent gcs"
  shows "edge_collect a (mcp_gamma (map fst gcs) x ∩ Collect (eval_query.channel_holds ch))
          ⊆ mcp_gamma (map fst gcs) (mcp_step (map snd gcs) ch a x)"
  using assms
proof (induction gcs arbitrary: x)
  case Nil
  then show ?case by (simp add: mcp_step_def)
next
  case (Cons gc gcs)
  obtain g c where gc: "gc = (g, c)" by fastforce
  let ?O = "Collect (eval_query.channel_holds ch)"
  let ?x1 = "ls_step c ch a x"
  have frames: "∀g' ∈ fst ‘ set gcs. mcp_frame c g'" "∀c' ∈ snd ‘ set gcs. mcp_frame c' g"
    "mcp_independent gcs"
    using mcp_independent_Cons_frames[OF Cons.prems(2)[unfolded gc]] by simp_all
  have own: "edge_collect a (g x ∩ ?O) ⊆ g ?x1"
    using Cons.prems(1) unfolding gc by (simp add: sound_local_spec_def)
  have own_kept: "g (mcp_step (map snd gcs) ch a ?x1) = g ?x1"
    unfolding mcp_step_def by (rule gamma_fold_step) (use frames in auto)
  have rest: "edge_collect a (mcp_gamma (map fst gcs) ?x1 ∩ ?O)
      ⊆ mcp_gamma (map fst gcs) (mcp_step (map snd gcs) ch a ?x1)"
    by (rule Cons.IH) (use Cons.prems(1) frames in auto)
  have rest_eq: "mcp_gamma (map fst gcs) ?x1 = mcp_gamma (map fst gcs) x"
    by (rule mcp_gamma_frame(1)) (use frames in auto)
  have "edge_collect a (mcp_gamma (map fst (gc # gcs)) x ∩ ?O)
        ⊆ edge_collect a (g x ∩ ?O) ∩ edge_collect a (mcp_gamma (map fst gcs) ?x1 ∩ ?O)"
    unfolding gc by (intro Int_greatest edge_collect_mono) (auto simp: rest_eq)
  also have "... ⊆ mcp_gamma (map fst (gc # gcs)) (mcp_step (map snd (gc # gcs)) ch a x)"
    using own own_kept rest unfolding gc by (auto simp: mcp_step_def)
  finally show ?case .
qed

lemma mcp_comb_sound:
  assumes "∀(g, c) ∈ set gcs. sound_local_spec 𝒢 g c" and "mcp_independent gcs"
    and "s ∈ mcp_gamma (map fst gcs) x" and "eval_query.channel_holds ch s"
    and "t ∈ mcp_gamma (map fst gcs) de" and "eval_query.channel_holds ch' t"
  shows "combine_collect 𝒢 (ci_dst ci) s t
          ∈ mcp_gamma (map fst gcs) (mcp_comb (map snd gcs) ch ch' ci x de)"
  using assms
proof (induction gcs arbitrary: x)
  case Nil
  then show ?case by simp
next
  case (Cons gc gcs)
  obtain g c where gc: "gc = (g, c)" by fastforce
  let ?x1 = "ls_combine c ch ch' ci x de"
  have frames: "∀g' ∈ fst ‘ set gcs. mcp_frame c g'" "∀c' ∈ snd ‘ set gcs. mcp_frame c' g"
    "mcp_independent gcs"
    using mcp_independent_Cons_frames[OF Cons.prems(2)[unfolded gc]] by simp_all
  have own: "combine_collect 𝒢 (ci_dst ci) s t ∈ g ?x1"
    using Cons.prems(1,3-6) unfolding gc by (simp add: sound_local_spec_def)
  have own_kept: "g (mcp_comb (map snd gcs) ch ch' ci ?x1 de) = g ?x1"
    unfolding mcp_comb_def by (rule gamma_fold_comb) (use frames in auto)
  have rest_eq: "mcp_gamma (map fst gcs) ?x1 = mcp_gamma (map fst gcs) x"
    by (rule mcp_gamma_frame(2)) (use frames in auto)
  have rest: "combine_collect 𝒢 (ci_dst ci) s t
      ∈ mcp_gamma (map fst gcs) (mcp_comb (map snd gcs) ch ch' ci ?x1 de)"
    by (rule Cons.IH) (use Cons.prems frames rest_eq gc in auto)
  show ?case
    using own own_kept rest unfolding gc by (simp add: mcp_comb_def)
qed
```

Entry threads a list of pairs through the components. The invariant carries the concretizations of the components already run (`ds`), whose fields are set in both halves, and the components still to run (`gcs`), whose fields are still the caller's.

```
lemma mcp_en_fold_sound:
  assumes "∀(g, c) ∈ set gcs. sound_local_spec 𝒢 g c" and "mcp_independent gcs"
    and "∀c ∈ snd ‘ set gcs. ∀d ∈ set ds. mcp_frame c d"
    and "∃p ∈ set ps. s ∈ mcp_gamma (map fst gcs) (fst p) ∧ s ∈ mcp_gamma ds (fst p)
                    ∧ E ∈ mcp_gamma ds (snd p)"
    and A: "eval_query.channel_holds ch s"
    and E: "E = call_enter 𝒢 (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s"
```

```
  shows "∃p ∈ set (fold (λc ps. concat (map (ls_enter c ch ci) ps)) (map snd gcs) ps).
           s ∈ mcp_gamma (map fst gcs) (fst p) ∧ s ∈ mcp_gamma ds (fst p)
           ∧ E ∈ mcp_gamma (map fst gcs) (snd p) ∧ E ∈ mcp_gamma ds (snd p)"
  using assms(1-4)
proof (induction gcs arbitrary: ps ds)
  case Nil
  then show ?case by simp
next
  case (Cons gc gcs)
  obtain g c where gc: "gc = (g, c)" by fastforce
  have frames: "∀g' ∈ fst ` set gcs. mcp_frame c g'" "∀c' ∈ snd ` set gcs. mcp_frame c' g"
    "mcp_independent gcs"
    using mcp_independent_Cons_frames[OF Cons.prems(2)[unfolded gc]] by simp_all
  obtain p where p: "p ∈ set ps" "s ∈ g (fst p)" "s ∈ mcp_gamma (map fst gcs) (fst p)"
      "s ∈ mcp_gamma ds (fst p)" "E ∈ mcp_gamma ds (snd p)"
    using Cons.prems(4) unfolding gc by auto
  have "sound_local_spec 𝒢 g c" using Cons.prems(1) unfolding gc by simp
  then obtain q where q: "q ∈ set (ls_enter c ch ci p)" "s ∈ g (fst q)" "E ∈ g (snd q)"
    using p(2) A unfolding E sound_local_spec_def by meson
  have keep: "d (fst q) = d (fst p) ∧ d (snd q) = d (snd p)"
    if "d ∈ fst ` set gcs ∪ set ds" for d
  proof -
    have "mcp_frame c d" using that frames(1) Cons.prems(3) unfolding gc by auto
    then show ?thesis using q(1) unfolding mcp_frame_def by blast
  qed
  have q_in: "q ∈ set (concat (map (ls_enter c ch ci) ps))"
    using p(1) q(1) by auto
  have "∃p ∈ set (fold (λc ps. concat (map (ls_enter c ch ci) ps)) (map snd gcs)
                  (concat (map (ls_enter c ch ci) ps))).
          s ∈ mcp_gamma (map fst gcs) (fst p) ∧ s ∈ mcp_gamma (g # ds) (fst p)
          ∧ E ∈ mcp_gamma (map fst gcs) (snd p) ∧ E ∈ mcp_gamma (g # ds) (snd p)"
  proof (rule Cons.IH)
    show "∀(g, c) ∈ set gcs. sound_local_spec 𝒢 g c" using Cons.prems(1) by simp
    show "mcp_independent gcs" by (fact frames(3))
    show "∀c' ∈ snd ` set gcs. ∀d ∈ set (g # ds). mcp_frame c' d"
      using frames(2) Cons.prems(3) unfolding gc by auto
    show "∃p ∈ set (concat (map (ls_enter c ch ci) ps)).
            s ∈ mcp_gamma (map fst gcs) (fst p) ∧ s ∈ mcp_gamma (g # ds) (fst p)
            ∧ E ∈ mcp_gamma (g # ds) (snd p)"
    proof (rule bexI[OF _ q_in])
      show "s ∈ mcp_gamma (map fst gcs) (fst q) ∧ s ∈ mcp_gamma (g # ds) (fst q)
              ∧ E ∈ mcp_gamma (g # ds) (snd q)"
      proof -
        have "mcp_gamma (map fst gcs) (fst q) = mcp_gamma (map fst gcs) (fst p)"
            "mcp_gamma ds (fst q) = mcp_gamma ds (fst p)"
            "mcp_gamma ds (snd q) = mcp_gamma ds (snd p)"
          using keep by (auto simp: mcp_gamma_def intro!: INF_cong)
        then show ?thesis using p(3-5) q(2,3) by simp
      qed
    qed
  qed
  then show ?case unfolding gc by (auto simp: mcp_gamma_def)
qed

lemma mcp_qry_fold_sound:
  "s ∈ mcp_gamma (map fst gcs) x ⟹ eval_query.channel_holds ch s
   ⟹ ∀(g, c) ∈ set gcs. sound_local_spec 𝒢 g c
   ⟹ eval_holds q r s
   ⟹ eval_holds q (fold (λc r. r ⊓ ls_query c ch x q) (map snd gcs) r) s"
proof (induction gcs arbitrary: r)
  case (Cons gc gcs)
  obtain g c where gc: "gc = (g, c)" by fastforce
  have "eval_holds q (ls_query c ch x q) s"
    using Cons.prems unfolding gc by (simp add: sound_local_spec_def)
  with Cons.prems(4) have "eval_holds q (r ⊓ ls_query c ch x q) s"
    by (rule eval_query.inf_sound)
  then show ?case
    using Cons.IH[of "r ⊓ ls_query c ch x q"] Cons.prems(1-3) unfolding gc by simp
qed simp
```

A component as a specification

A component runs as the specification whose fields are its own operations, each given the component's closed channel on the state it is asked about. The specification reads no global and publishes none. Its channel is a pure function of the state, so a transfer asks it whatever it needs while it runs, as a Goblint transfer asks

`man.ask`. Entry starts both halves at the caller's record.

The step a component takes on an edge, with its own channel.

```
definition closed_step :: "'s local_spec ⇒ edge_action ⇒ 's ⇒ 's" where
  "closed_step c a d = ls_step c (ls_channel c d) a d"
```

$[\![a]\!]^\sharp_c\ d$ is the abstract transfer of action `a`, the counterpart of the abstract evaluation $[\![e]\!]^\sharp\ d$; the subscript names the component.

```
abbreviation abs_step :: "edge_action ⇒ 's local_spec ⇒ 's ⇒ 's"
  ("[_]#_") where
  "[a]#c ≡ closed_step c a"

definition dg_spec_of :: "'s local_spec ⇒ ('x,'k,'v,'s::bot,'G) dg_spec" where
  "dg_spec_of c = local_dg_spec_template(|
     dgs_skip := local_transfer [EA_Nop]#c,
     dgs_assign := (λx e. local_transfer [EA_Assign x e]#c),
     dgs_special := (λsc x. local_transfer [EA_Special sc x]#c),
     dgs_branch := (λb pol. local_transfer
                     [if pol then EA_Assume b else EA_AssumeNot b]#c),
     dgs_body := (λp. local_transfer [EA_Body p]#c),
     dgs_return := (λe p. local_transfer [EA_Ret e p]#c),
     dgs_enter := (λci. local_enter_transfer (λd. ls_enter c (ls_channel c d) ci (d, d))),
     dgs_event := (λev. local_transfer [event_action ev]#c),
     dgs_combine_env := (λci. local_combine_transfer
        (λdc de. ls_combine_env c (ls_channel c dc) (ls_channel c de) ci dc de)),
     dgs_combine_assign := (λci. local_combine_transfer
        (λdc de. ls_combine_assign c (ls_channel c de) ci dc de)),
     dgs_query := local_query (ls_channel c) |)"


lemma dg_spec_of_simps [simp]:
  "skip# (dg_spec_of c) = local_transfer [EA_Nop]#c"
  "assign# (dg_spec_of c) x e = local_transfer [EA_Assign x e]#c"
  "special# (dg_spec_of c) sc x = local_transfer [EA_Special sc x]#c"
  "branch# (dg_spec_of c) b pol
     = local_transfer [if pol then EA_Assume b else EA_AssumeNot b]#c"
  "body# (dg_spec_of c) p = local_transfer [EA_Body p]#c"
  "return# (dg_spec_of c) eo p = local_transfer [EA_Ret eo p]#c"
  "enter# (dg_spec_of c) ci
     = local_enter_transfer (λd. ls_enter c (ls_channel c d) ci (d, d))"
  "event# (dg_spec_of c) ev = local_transfer [event_action ev]#c"
  "combine_env# (dg_spec_of c) ci = local_combine_transfer
     (λdc de. ls_combine_env c (ls_channel c dc) (ls_channel c de) ci dc de)"
  "combine_assign# (dg_spec_of c) ci = local_combine_transfer
     (λdc de. ls_combine_assign c (ls_channel c de) ci dc de)"
  "dgs_query (dg_spec_of c) = local_query (ls_channel c)"
  by (simp_all add: dg_spec_of_def)

lemma dg_spec_step_dg_spec_of [simp]:
  "dg_spec_step (dg_spec_of c) a = local_transfer [a]#c"
  by (cases a) simp_all

lemma dg_spec_combine_transfer_dg_spec_of [simp]:
  "dg_spec_combine_transfer (dg_spec_of c) ci
     = local_combine_transfer (λdc de. ls_combine c (ls_channel c dc) (ls_channel c de) ci dc de)"
  by (intro ext) (simp add: dg_spec_combine_transfer_local local_combine_transfer_def)

lemma dg_spec_wf_dg_spec_of [intro, simp]: "dg_spec_wf (dg_spec_of c)"
  by (auto simp: dg_spec_wf_def local_query_def local_enter_transfer_def
      local_combine_transfer_def local_transfer_def)
```

Run as a specification, a sound component's step and return are sound outright: its own channel holds at every store its state describes, so the laws it proves against every holding channel apply to that one.

```
lemma closed_step_sound:
  assumes sound: "sound_local_spec 𝒢 gm c"
  shows "edge_collect a (gm d) ⊆ gm ([a]#c d)"
proof -
  have "gm d = gm d ∩ Collect (eval_query.channel_holds (ls_channel c d))"
    using ls_channel_sound[OF sound] by blast
  moreover have "edge_collect a (gm d ∩ Collect (eval_query.channel_holds (ls_channel c d)))
      ⊆ gm (ls_step c (ls_channel c d) a d)"
    using sound unfolding sound_local_spec_def by blast
  ultimately show ?thesis by (simp add: closed_step_def)
qed
```

```
lemma closed_combine_sound:
  assumes sound: "sound_local_spec 𝒢 gm c" and "s ∈ gm dc" and "t ∈ gm de"
  shows "combine_collect 𝒢 (ci_dst ci) s t
         ∈ gm (ls_combine c (ls_channel c dc) (ls_channel c de) ci dc de)"
  using assms ls_channel_sound[OF sound assms(2)] ls_channel_sound[OF sound assms(3)]
  unfolding sound_local_spec_def by blast

theorem dg_spec_of_contract:
  assumes sound: "sound_local_spec 𝒢 gm c"
  shows "analysis_contract (dg_spec_of c) (λd g. gm d) 𝒢"
proof (unfold_locales, goal_cases wf mono step comb)
  case wf
  then show ?case by (rule dg_spec_wf_dg_spec_of)
next
  case mono
  then show ?case using sound unfolding sound_local_spec_def by metis
next
  case (step a τ src gk)
  then show ?case
    using closed_step_sound[OF sound] by (simp add: dg_spec_edge_program_def)
next
  case comb
  then show ?case
    by (simp add: local_combine_transfer_def closed_combine_sound[OF sound])
qed
```

Several components as one

The combination of several components is itself a component. One component is its own combination, so a single analysis and an `MCP` of one analysis are the same specification. Several components run their returns one after the other, each through both of its stages: the combination's env stage runs every component's composite, and its assign stage keeps the state it is handed. Running every env stage before every assign stage would need the stages of different components to commute, which independence of their concretizations does not give. The composite needs the caller's channel, which only the env stage receives.

```
fun mcp_combine :: "'s local_spec list ⇒ 's local_spec" where
  "mcp_combine [c] = c"
| "mcp_combine cs =
     make_local_spec (mcp_qry cs) (mcp_step cs) (mcp_en_from cs) (mcp_comb cs) (λch' ci dc de. dc)"

definition mcp_spec :: "'s local_spec list ⇒ ('x,'k,'v,'s::bot,'G) dg_spec" where
  "mcp_spec cs = dg_spec_of (mcp_combine cs)"

theorem mcp_combine_sound:
  assumes sound: "∀(γ_i, c_i) ∈ set comps. sound_local_spec 𝒢 γ_i c_i"
    and indep: "mcp_independent comps" and ne: "comps ≠ []"
  shows "sound_local_spec 𝒢 (mcp_gamma (map fst comps)) (mcp_combine (map snd comps))"
proof (cases "∃x. comps = [x]")
  case True
  then obtain γ_i c_i where "comps = [(γ_i, c_i)]" by fastforce
  then show ?thesis using sound by (simp add: sound_local_spec_def mcp_gamma_def)
next
  case False
  have comb: "mcp_combine (map snd comps) =
     make_local_spec (mcp_qry (map snd comps)) (mcp_step (map snd comps))
       (mcp_en_from (map snd comps)) (mcp_comb (map snd comps)) (λch' ci dc de. dc)"
    using False ne by (cases "map snd comps" rule: mcp_combine.cases) (auto simp: Cons_eq_map_conv)
  have enter: "∃q ∈ set (mcp_en_from (map snd comps) ch ci p).
                 s ∈ mcp_gamma (map fst comps) (fst q)
                 ∧ call_enter 𝒢 (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s
                     ∈ mcp_gamma (map fst comps) (snd q)"
    if "s ∈ mcp_gamma (map fst comps) (fst p)" "eval_query.channel_holds ch s" for ch s ci p
    using mcp_en_fold_sound[OF sound indep, of "[]" "[p]" s _ ch ci] that
    by (auto simp: mcp_en_from_def)
  have mono: "∀x y. x ≤ y ⟶ mcp_gamma (map fst comps) x ⊆ mcp_gamma (map fst comps) y"
    using sound by (fastforce simp: mcp_gamma_def sound_local_spec_def)
  have qry: "∀ch s x q. s ∈ mcp_gamma (map fst comps) x ⟶ eval_query.channel_holds ch s
               ⟶ eval_holds q (mcp_qry (map snd comps) ch x q) s"
    unfolding mcp_qry_def using mcp_qry_fold_sound[OF _ _ sound] by simp
  show ?thesis
    unfolding comb sound_local_spec_def
    using mono mcp_step_sound[OF sound indep] mcp_comb_sound[OF sound indep] enter qry
    by simp
qed
```

```
theorem mcp_contract:
  assumes "∀(γ_i, c_i) ∈ set comps. sound_local_spec 𝒢 γ_i c_i" and "mcp_independent comps"
    and "comps ≠ []"
  shows "analysis_contract (mcp_spec (map snd comps)) (λd g. mcp_gamma (map fst comps) d) 𝒢"
  unfolding mcp_spec_def by (rule dg_spec_of_contract[OF mcp_combine_sound[OF assms]])
```

A component on one field of a larger state

A component over its own carrier `'c` runs on one field of the combined state through a lens: every operation reads the field, and writes back only the field, as Goblint's `inner_man` hands a component its own part of the `MCP` state. The channel passes through unchanged: it describes the stores of the whole state, which are the stores the field describes as well.

```
definition lens_of :: "('s ⇒ 'c) ⇒ ('s ⇒ 'c ⇒ 's) ⇒ 'c local_spec ⇒ 's local_spec"
where
  "lens_of get put c = make_local_spec
     (λch x. ls_query c ch (get x))
     (λch a x. put x (ls_step c ch a (get x)))
     (λch ci p. map (λ(q, e). (put (fst p) q, put (snd p) e))
                (ls_enter c ch ci (get (fst p), get (snd p))))
     (λch ch' ci x de. put x (ls_combine_env c ch ch' ci (get x) (get de)))
     (λch' ci x de. put x (ls_combine_assign c ch' ci (get x) (get de)))"

lemma get_mc_comb_lens_of:
  assumes "⋀x v. get (put x v) = v"
  shows
    "get (ls_combine (lens_of get put c) ch ch' ci x de) = ls_combine c ch ch' ci (get x) (get de)"
  by (simp add: lens_of_def assms)

theorem lens_of_sound:
  assumes sound: "sound_local_spec 𝒢 g c"
    and get_put: "⋀x v. get (put x v) = v"
    and get_mono: "⋀x y. x ≤ y ⟹ get x ≤ get y"
  shows "sound_local_spec 𝒢 (λx. g (get x)) (lens_of get put c)"
proof -
  have enter: "∃q ∈ set (ls_enter (lens_of get put c) ch ci p).
                 s ∈ g (get (fst q))
                 ∧ call_enter 𝒢 (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s
                     ∈ g (get (snd q))"
    if s_in: "s ∈ g (get (fst p))" and A: "eval_query.channel_holds ch s" for ch s ci p
  proof -
    have "s ∈ g (fst (get (fst p), get (snd p)))" using s_in by simp
    then obtain q where "q ∈ set (ls_enter c ch ci (get (fst p), get (snd p)))" "s ∈ g (fst q)"
        "call_enter 𝒢 (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s ∈ g (snd q)"
      using sound A unfolding sound_local_spec_def by metis
    then show ?thesis
      by (intro bexI[of _ "(put (fst p) (fst q), put (snd p) (snd q))"])
         (auto simp: lens_of_def get_put)
  qed
  show ?thesis
    using sound get_mono enter
    unfolding sound_local_spec_def get_mc_comb_lens_of[of get put, OF get_put]
    by (simp add: lens_of_def get_put)
qed
```

Replacing a component's handler

The handler is the only operation that neither changes a state nor is framed, so any other handler that answers soundly for the same states may replace it. This is how an analysis written without a query handler becomes a provider once it runs in the combined state.

```
definition with_qry :: "(channel ⇒ 's ⇒ channel) ⇒ 's local_spec ⇒ 's local_spec" where
  "with_qry h c = c(|ls_query := h|)"

theorem with_qry_sound:
  assumes "sound_local_spec 𝒢 gm c"
    and "⋀ch s x q. s ∈ gm x ⟹ eval_query.channel_holds ch s ⟹ eval_holds q (h ch x q) s"
  shows "sound_local_spec 𝒢 gm (with_qry h c)"
  using assms unfolding sound_local_spec_def with_qry_def by simp

lemma with_qry_frame: "mcp_frame c g ⟹ mcp_frame (with_qry h c) g"
  by (simp add: mcp_frame_def with_qry_def)
```

Normalizing what a component produces

A normalization `k` that keeps a state's concretization may be applied to what a component produces: each step, the entered half of each entry, and the return after its second stage. The caller half of an entry is the caller's state and stays as it is. The first stage of a return is left alone, because the component's soundness speaks only about the composite. The combined state of several analyses uses this to become unreachable as soon as one active analysis is, as Goblint's `MCP` raises `Deadcode`.

```
definition map_local_spec :: "('s ⇒ 's) ⇒ 's local_spec ⇒ 's local_spec" where
  "map_local_spec k c = make_local_spec (ls_query c)
     (λch a x. k (ls_step c ch a x))
     (λch ci p. map (λ(q, e). (q, k e)) (ls_enter c ch ci p))
     (ls_combine_env c)
     (λch' ci x de. k (ls_combine_assign c ch' ci x de))"

theorem map_local_spec_sound:
  assumes sound: "sound_local_spec 𝒢 g c"
    and keep: "⋀x. g (k x) = g x"
  shows "sound_local_spec 𝒢 g (map_local_spec k c)"
proof -
  have enter: "∃q ∈ set (ls_enter (map_local_spec k c) ch ci p).
                 s ∈ g (fst q)
                 ∧ call_enter 𝒢 (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s ∈ g (snd q)"
    if s_in: "s ∈ g (fst p)" and A: "eval_query.channel_holds ch s" for ch s ci p
  proof -
    obtain q where "q ∈ set (ls_enter c ch ci p)" "s ∈ g (fst q)"
        "call_enter 𝒢 (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s ∈ g (snd q)"
      using sound s_in A unfolding sound_local_spec_def by metis
    then show ?thesis
      by (intro bexI[of _ "(fst q, k (snd q))"]) (auto simp: map_local_spec_def keep)
  qed
  show ?thesis
    using sound enter unfolding sound_local_spec_def
    by (simp add: map_local_spec_def keep)
qed
```

Components whose entry answers once

An entry that answers a single alternative, paired with the caller's own state, is what the routed pipeline consumes. Lenses, combination and normalization keep that shape.

```
definition single_entry :: "'s local_spec ⇒ bool" where
  "single_entry c ⟷ (∀ch ci p. ∃e. ls_enter c ch ci p = [(fst p, e)])"

lemma single_entryD:
  "single_entry c ⟹ ls_enter c ch ci p = [(fst p, snd (hd (ls_enter c ch ci p)))]"
  unfolding single_entry_def by (metis list.sel(1) snd_conv)

lemma single_entry_lens_of:
  assumes single: "single_entry c" and put_get: "⋀x. put x (get x) = x"
  shows "single_entry (lens_of get put c)"
  unfolding single_entry_def
proof (intro allI)
  fix ch ci p
  obtain e where "ls_enter c ch ci (get (fst p), get (snd p)) = [(get (fst p), e)]"
    using single unfolding single_entry_def by (metis fst_conv)
  then show "∃e. ls_enter (lens_of get put c) ch ci p = [(fst p, e)]"
    by (simp add: lens_of_def put_get)
qed

lemma single_entry_map_local_spec:
  assumes "single_entry c"
  shows "single_entry (map_local_spec k c)"
  unfolding single_entry_def
proof (intro allI)
  fix ch ci p
  obtain e where "ls_enter c ch ci p = [(fst p, e)]"
    using assms unfolding single_entry_def by blast
  then show "∃e. ls_enter (map_local_spec k c) ch ci p = [(fst p, e)]"
    by (simp add: map_local_spec_def)
qed

lemma single_entry_with_qry: "single_entry c ⟹ single_entry (with_qry h c)"
  by (simp add: single_entry_def with_qry_def)
```

```
lemma single_entry_mcp_en_fold:
  assumes "∀c ∈ set cs. single_entry c"
  shows "∃e. fold (λc ps. concat (map (ls_enter c ch ci) ps)) cs [(d, e0)] = [(d, e)]"
  using assms
proof (induction cs arbitrary: e0)
  case (Cons c cs)
  obtain e1 where "ls_enter c ch ci (d, e0) = [(d, e1)]"
    using Cons.prems unfolding single_entry_def by (metis fst_conv list.set_intros(1))
  then show ?case using Cons.IH[of e1] Cons.prems by simp
qed simp

lemma single_entry_mcp_combine:
  assumes single: "∀c ∈ set cs. single_entry c" and ne: "cs ≠ []"
  shows "single_entry (mcp_combine cs)"
proof (cases "∃c. cs = [c]")
  case True
  then show ?thesis using single by auto
next
  case False
  then have "ls_enter (mcp_combine cs) = mcp_en_from cs"
    using ne by (cases cs rule: mcp_combine.cases) auto
  then show ?thesis
    unfolding single_entry_def mcp_en_from_def
    using single_entry_mcp_en_fold[OF single] by (metis prod.collapse)
qed
```

One field of a reachability-lifted record

The combined state is a record of fields under one reachability lift, whose `Bot` is the state no analysis can reach. Each field has a bottom of its own: a reachability-lifted field's is `Bot`, and a carrier with an unreachable element of its own uses that. A field's lens reads the field's bottom from the unreachable state, and writing any other value into it starts from the record whose fields are all at bottom. Writing bottom into the unreachable state leaves it unreachable, so writing back what was read changes nothing.

```
definition lift_get :: "('r ⇒ 'c::bot) ⇒ 'r lifted ⇒ 'c" where
  "lift_get f x = (case x of Bot ⇒ ⊥ | Lifted r ⇒ f r)"

definition lift_put :: "('r ⇒ 'c::bot ⇒ 'r) ⇒ 'r::bot lifted ⇒ 'c ⇒ 'r lifted" where
  "lift_put u x v =
     (case x of
        Bot ⇒ (if v = ⊥ then Bot else Lifted (u ⊥ v))
      | Lifted r ⇒ Lifted (u r v))"

lemma lift_get_simps [simp]:
  "lift_get f Bot = ⊥" "lift_get f (Lifted r) = f r"
  by (simp_all add: lift_get_def)

lemma lift_get_put:
  assumes "⋀r v. f (u r v) = v" and "f ⊥ = ⊥"
  shows "lift_get f (lift_put u x v) = v"
  using assms by (cases x) (simp_all add: lift_put_def)

lemma lift_put_get:
  assumes "⋀r. u r (f r) = r"
  shows "lift_put u x (lift_get f x) = x"
  using assms by (cases x) (simp_all add: lift_put_def)

lemma lift_get_mono:
  fixes f :: "'r::semilattice_sup ⇒ 'c::order_bot"
  assumes "⋀r r'. r ≤ r' ⟹ f r ≤ f r'"
  shows "x ≤ y ⟹ lift_get f x ≤ lift_get f y"
  using assms by (cases x; cases y) simp_all

lemma lift_get_put_other:
  assumes "⋀r v. f2 (u1 r v) = f2 r" and "f2 ⊥ = ⊥"
  shows "lift_get f2 (lift_put u1 x v) = lift_get f2 x"
  using assms by (cases x) (simp_all add: lift_put_def)
```

Components that each own a distinct field are independent: a list of distinct analyses, each run on its own field, satisfies `mcp_independent` as soon as every one leaves every other field alone.

```
lemma mcp_independent_map:
  assumes "distinct as"
    and "⋀a b. a ∈ set as ⟹ b ∈ set as ⟹ a ≠ b ⟹ mcp_frame (c a) (g b)"
  shows "mcp_independent (map (λa. (g a, c a)) as)"
```

```
  using assms by (induction as) auto
end
```

6.22 Channel_Wrappers

```
theory Channel_Wrappers
  imports "Voblint_Framework.MCP_Spec"
begin
```

Consulting the answers at an assignment

A component that was written without queries can still use them through a wrapper around one transfer. At `x = e` it asks for the value of `e`; if the answer is a single integer `n`, the wrapper assigns the literal `n` instead of evaluating `e` itself. The component's own assignment of that literal then covers the concrete successor, because `e` evaluates to `n` at every store the answer holds at.

```
definition assign_ask ::
  "(channel ⇒ vname ⇒ exp ⇒ 'D ⇒ 'D) ⇒ channel ⇒ vname ⇒ exp ⇒ 'D ⇒ 'D" where
  "assign_ask asn ch x e d =
     (case answer_const (ch (EvalInt e)) of
        Some n ⇒ asn ch x (N n) d
      | None ⇒ asn ch x e d)"
```

The wrapper keeps an assignment's law: where it substitutes the literal, the answer holds at the store, so the literal evaluates to what `e` does.

```
lemma assign_ask_sound:
  assumes "sound_assign gm asn"
  shows "sound_assign gm (assign_ask asn)"
  unfolding sound_assign_def
proof (intro allI impI)
  fix ch x s y e
  assume s: "s ∈ gm x" and o: "eval_query.channel_holds ch s"
  have base: "s(y := ⟦e'⟧ₑ s) ∈ gm (asn ch y e' x)" for e'
    using assms s o unfolding sound_assign_def by blast
  show "s(y := ⟦e⟧ₑ s) ∈ gm (assign_ask asn ch y e x)"
  proof (cases "answer_const (ch (EvalInt e))")
    case None
    then show ?thesis using base[of e] by (simp add: assign_ask_def)
  next
    case (Some n)
    have "⟦e⟧ₑ s = n"
      by (rule eval_holds_constD[OF eval_query.channel_holdsD[OF o] Some])
    then show ?thesis using Some base[of "N n"] by (simp add: assign_ask_def)
  qed
qed
```

A component that asks at its assignments

The same wrapper around a whole component: at every assignment it asks the channel for the value of the right-hand side and runs the assignment on the literal when the answer is a single integer. It changes neither the component's concretization nor its entry, so soundness, framing and a single entry carry over. Where every answer is $\top$, as when no active analysis answers, it is the component itself.

```
definition ask_assign :: "'s local_spec ⇒ 's local_spec" where
  "ask_assign c = c⦇ls_assign := assign_ask (ls_assign c)⦈"

lemma ask_assign_step_cases:
  "ls_step (ask_assign c) ch a x = ls_step c ch a x
   ∨ (∃y e n. a = EA_Assign y e ∧ answer_const (ch (EvalInt e)) = Some n
        ∧ ls_step (ask_assign c) ch a x = ls_step c ch (EA_Assign y (N n)) x)"
  by (cases a) (auto simp: ask_assign_def assign_ask_def split: option.splits)

lemma answer_const_top [simp]: "answer_const ⊤ = None"
  by (simp add: top_query_lift_def)

lemma single_entry_ask_assign: "single_entry c ⟹ single_entry (ask_assign c)"
  by (simp add: single_entry_def ask_assign_def)

theorem ask_assign_frame:
  assumes frame: "mcp_frame c g"
  shows "mcp_frame (ask_assign c) g"
proof -
  have st: "g (ls_step c ch a x) = g x" for ch a x
    using frame unfolding mcp_frame_def by blast
  have "g (ls_step (ask_assign c) ch a x) = g x" for ch a x
    using ask_assign_step_cases[of c ch a x] st by metis
  then show ?thesis
```

```
    using frame unfolding mcp_frame_def by (simp add: ask_assign_def)
qed

theorem ask_assign_sound:
  assumes sound: "sound_local_spec 𝒢 gm c"
  shows "sound_local_spec 𝒢 gm (ask_assign c)"
  unfolding ask_assign_def
  by (rule sound_local_spec_update_assign[OF sound
        assign_ask_sound[OF sound_local_spec_assignD[OF sound]]])

end
```

6.23 Call_String_Context

```
theory Call_String_Context
  imports "Voblint_CFG.CFG_Def"
begin
```

A reusable bounded call-string context

The call-string route/enter-context pair: this theory owns only the call-string *data* and the one closed-term equality that makes `routed_context`'s routing agreement trivial for any bound `k` — it deliberately does not import `Routed_Context` or `DG_Ctx_Activation`, and fixes no domain, no solver, and no CFG. `cs_route`/`cs_context` (defined below) plug into `routed_context`'s `route`/`enterc` parameters at whatever concrete instantiation a caller chooses; nothing here decides what that instantiation is.

The call-string type and its two projections

Most recent call site first, unbounded as a type — `k` only bounds the *values* `cs_route`/`cs_context` produce, not the type itself. This mirrors Goblint's `C.t` being a value-level choice under a type-level bound only because OCaml forces the bound to be a type; here `k` is ordinary `nat` data.

```
type_synonym call_string = "cfg_node list"
```

The equation-level route: push the call site, keep only the most recent `k`. Ignores the entered abstract/executable value `d` entirely, so it is representation-independent for *any* domain without a per-domain commute lemma (`cs_route_indep_of_data` below).

```
definition cs_route :: "nat ⇒ pp ⇒ call_string ⇒ 'd ⇒ call_action ⇒ call_string" where
  "cs_route k u ctx d ca = take k (u # ctx)"
```

The trace-semantic context function whose graph, `context_policy_of_fun (cs_context k)`, instantiates `routed_context`'s `adm`: same closed term as `cs_route`, over the concrete `store` the activation-trace semantics supplies instead of an abstract/executable `'d`.

```
definition cs_context :: "nat ⇒ cfg_node ⇒ call_string ⇒ store ⇒ call_string" where
  "cs_context k u ctx s = take k (u # ctx)"
```

The two facts every instance needs

Routing agreement, generically: since neither side reads `d`/`s`, this holds for *any* `k`, `u`, `ctx`, `d`, `ca`, `s` with zero case analysis.

```
lemma cs_route_context_agree: "cs_route k u ctx d ca = cs_context k u ctx s"
  by (simp add: cs_route_def cs_context_def)
```

Representation independence: `cs_route` agrees with itself under *any* substitution for its data argument, since `d` never appears on the right-hand side. A caller needing `cs_route k u ctx (d::'exec) ca = cs_route k u ctx (repr d) ca` for a specific representation map `repr` – the map from an executable state to its function, say – gets it as an instance of this lemma, or just as directly by unfolding `cs_route_def` at the call site — both are one line.

```
lemma cs_route_indep_of_data: "cs_route k u ctx d ca = cs_route k u ctx d' ca"
  by (simp add: cs_route_def)
```

Bounded length

The one fact that makes this context space finite: routing truncates, so no context a `k`-bounded run produces is longer than `k`. Over a graph with finitely many nodes that bounds the whole space before any solve is attempted, rather than per run.

```
lemma cs_route_length: "length (cs_route k u ctx d ca) ≤ k"
  by (simp add: cs_route_def)
```

Truncation behaviour

Below the bound, pushing a call site loses nothing — a plain cons, not a truncation. Needed for any future precision argument: contexts stay distinct as long as the call histories being separated fit under `k`.

```
lemma cs_route_below_bound:
  "length ctx < k ⟹ cs_route k u ctx d ca = u # ctx"
  by (simp add: cs_route_def)
```

```
lemma cs_context_below_bound:
  "length ctx < k ⟹ cs_context k u ctx s = u # ctx"
  by (simp add: cs_context_def)
```

The central algebraic property of a bounded call string: projecting a longer bound's context down to a shorter one agrees with routing at the shorter bound directly. This is the `take k1 ctx_k2 = ctx_k1` fact any future `k1 <= k2` refinement theorem needs to relate the two analyses' unknown spaces.

```
lemma cs_route_k_mono:
  assumes "k1 ≤ k2"
  shows "take k1 (cs_route k2 u ctx d ca) = cs_route k1 u ctx d ca"
  using assms by (simp add: cs_route_def)
```

The call-string-keyed global-key type

The global-key space a call-string-routed equation system publishes into: one `Global` slot for the real globals, plus one `Seed` slot per (procedure entry, call-string) pair for the routed callee entry states. The `Seed` payload carries the call string itself, so the key shape does not depend on the bound `k` — only the payload's length does. Still pure data: no domain, no solver, no CFG.

```
datatype call_string_gk = Global | Seed (seed_pp: pp) (seed_cs: "cfg_node list")
```

end

6.24 Solved_Table

```
theory Solved_Table
  imports "Voblint_Domain.Nonrelational_Reachability" "Voblint_CFG.CFG_Def"
begin
```

Solved analysis results

The canonical, domain-generic shape of a finished analysis: a set of covered `pp × 'ctx` keys together with a total lookup into a per-point abstract state. Checks, reports, and rendering are downstream consumers of this table, not siblings of it.

The datatype constrains neither the key set nor the payloads. What the rest of this text describes — finitely many keys, and canonical payloads — is `wf_solved_table`, stated at the end of this theory. Its two halves are not established alike: an adapter canonicalizes every payload it publishes, so canonicality is unconditional, while finiteness holds only for those context policies whose whole context space is bounded in advance, and is carried as a hypothesis everywhere else.

`Bot` means "the solver's canonical result at this key is an outer `Bot`", or the key is absent from `covered_keys` altogether; `Lifted a` means "the canonical result is `Lifted`", with the local unknown projected to the abstract state `a`. This is a structural reading, not a semantic-emptiness test: raw solver values are canonicalized – a witness-bottom `Lifted` payload collapsed to `Bot` – *before* they reach this boundary (every public result adapter routes through `canonicalize_lift`), so by the time a value reaches `lookup_table` here, `Bot` denotes no store. The converse does not hold: the emptiness test is sound and need not be complete, so a `Lifted` state may still denote no store (a product whose interval field says `x = 2` and whose parity field says `x` is odd stays `Lifted`).

Per-point reachability

Per-point reachability is `'a lifted` itself (`Voblint_Domain.Reachability_Lift`), not a nominal copy of it: the reachability reading is fixed below rather than left to the caller, and stated directly in terms of `Bot`/ `Lifted`. `map_lift` is the functorial map (payload rewriting that leaves reachability alone).

A point of an abstract-store table concretizes through the reachability lift: `Bot` to nothing, a `Lifted` store to its concretization. It is the lift at γ, so a result generic in its published value reads back to exactly this at an abstract-store instance.

```
lemma gamma_state_of_reachable_env [simp]:
  "γ (case p of Bot ⇒ bot | Lifted st ⇒ st) = γ⊥ p"
  for p :: "'a::numeric_domain abs_state lifted"
  by (cases p) simp_all
```

The result table

`covered_keys` is the authoritative coverage surface — the keys the solver actually reached — while `table_at` is merely a total lookup function, defined everywhere but meaningful only on the key set. `lookup_table` keeps that distinction explicit rather than trusting `table_at` to answer sensibly off the key set.

`'ctx` carries no ordering or enumeration constraint: real context types include interval-vector and call-string contexts, which have neither.

```
datatype (plugins del: quickcheck_narrowing) ('ctx, 'a) solved_table =
  Solved_Table
    (covered_keys: "(pp × 'ctx) set")
    (table_at: "pp ⇒ 'ctx ⇒ 'a lifted")

definition table_contexts :: "('ctx, 'a) solved_table ⇒ pp ⇒ 'ctx set" where
  "table_contexts r v = snd ‘ Set.filter (λ(v', ctx). v' = v) (covered_keys r)"

definition lookup_table ::
  "('ctx, 'a) solved_table ⇒ pp ⇒ 'ctx ⇒ 'a lifted" where
  "lookup_table r v ctx =
     (if (v, ctx) ∈ covered_keys r then table_at r v ctx else Bot)"

lemma table_contexts_iff [simp]: "ctx ∈ table_contexts r v ⟷ (v, ctx) ∈ covered_keys r"
  unfolding table_contexts_def by force

lemma lookup_table_absent [simp]:
  "(v, ctx) ∉ covered_keys r ⟹ lookup_table r v ctx = Bot"
  unfolding lookup_table_def by simp
```

Telling "analysed and dead" from "never analysed"

`lookup_table` answers `Bot` for two unrelated reasons: the solver covered the key and stored `Bot` there, or it never covered the key at all. The first is a proved statement about the program; the second is the absence of one. A reader of the answer alone cannot tell which happened, so a claim about concrete unreachability must not be stated over it.

```
datatype (plugins del: quickcheck_narrowing) 'a context_result =
    Uncovered
  | Covered "'a lifted"
```

`'a context_result` keeps the two apart at the lookup, rather than pushing the distinction into `'a lifted`: making non-coverage a lattice element would put solver bookkeeping into the abstract domain and raise questions – how it orders against `Bot`, what a join with it means – that have no semantic answer. Coverage is metadata about a solve; `Bot` is a statement about a program.

```
definition lookup_coverage ::
  "('ctx, 'a) solved_table ⇒ pp ⇒ 'ctx ⇒ 'a context_result" where
  "lookup_coverage r v ctx =
     (if (v, ctx) ∈ covered_keys r then Covered (table_at r v ctx) else Uncovered)"

lemma lookup_coverage_covered [simp]:
  "(v, ctx) ∈ covered_keys r ⟹
     lookup_coverage r v ctx = Covered (table_at r v ctx)"
  unfolding lookup_coverage_def by simp

lemma lookup_coverage_uncovered [simp]:
  "(v, ctx) ∉ covered_keys r ⟹ lookup_coverage r v ctx = Uncovered"
  unfolding lookup_coverage_def by simp

lemma lookup_coverage_eq_Uncovered_iff:
  "lookup_coverage r v ctx = Uncovered ⟷ (v, ctx) ∉ covered_keys r"
  unfolding lookup_coverage_def by simp

lemma lookup_coverage_eq_Covered_iff:
  "lookup_coverage r v ctx = Covered x ⟷
     (v, ctx) ∈ covered_keys r ∧ table_at r v ctx = x"
  unfolding lookup_coverage_def by auto
```

Where the information is lost, named so that losing it is a visible step. `lookup_table` is exactly this totalization, which is why it is safe to keep for callers that only ever read a covered key – and why a soundness claim may not be stated over it directly.

```
fun context_result_or_bot :: "'a context_result ⇒ 'a lifted" where
  "context_result_or_bot Uncovered = Bot"
| "context_result_or_bot (Covered x) = x"

lemma lookup_table_eq_or_bot:
  "lookup_table r v ctx =
     context_result_or_bot (lookup_coverage r v ctx)"
  unfolding lookup_table_def lookup_coverage_def by simp
```

The logical core of every "the reported point is unreachable" claim, stated once here rather than per domain. It starts from `Covered Bot`, so an uncovered node cannot satisfy it however the caller obtained its inclusion.

```
lemma reported_covered_unreachable_empty:
  assumes lookup: "lookup_coverage r v ctx = Covered Bot"
    and sound: "C ⊆ gamma_lift gam (lookup_table r v ctx)"
  shows "C = {}"
  using sound unfolding lookup_table_eq_or_bot lookup by simp
```

The node-level reading, quantifying over the contexts at `v`. It asks only that every *stored* result there is `Bot`; an uncovered context is permitted rather than required to be absent, since whatever an uncovered context contributes is a question for the solve that produced the table, not for this predicate. A single `Covered (Lifted s)` at any context defeats it, which is the whole point: that context may still carry executions.

Read the vacuous case before using this: a node with *no* stored context satisfies it, since there is nothing to be non-`Bot`. On its own that would be a hole — "never analysed" passing as "proved dead". It is not one only where a caller separately knows that an unstored context contributes nothing, which is a property of the solve, not of this table. Do not lift this predicate out of a setting that establishes it.

```
definition result_node_is_bottom ::
  "('ctx, 'a) solved_table ⇒ pp ⇒ bool" where
  "result_node_is_bottom r v ⟷
     (∀ctx a. lookup_coverage r v ctx = Covered a ⟶ a = Bot)"
```

```
lemma result_node_is_bottomI:
  assumes "⋀ctx a. lookup_coverage r v ctx = Covered a ⟹ a = Bot"
  shows "result_node_is_bottom r v"
  unfolding result_node_is_bottom_def using assms by blast

lemma result_node_is_bottomD:
  assumes bot: "result_node_is_bottom r v"
    and covered: "(v, ctx) ∈ covered_keys r"
  shows "lookup_coverage r v ctx = Covered Bot"
proof -
  have eq: "lookup_coverage r v ctx = Covered (table_at r v ctx)"
    using covered by simp
  have "table_at r v ctx = Bot"
    using bot eq unfolding result_node_is_bottom_def by blast
  with eq show ?thesis by simp
qed

lemma lookup_table_LiftedD [dest]:
  "lookup_table r v ctx = Lifted st ⟹ ctx ∈ table_contexts r v"
  using lookup_table_absent by fastforce
```

Well-formed results

`Solved_Table` is an ordinary constructor, so the two properties the opening text describes are not enforced by the type and have to be stated.

Finiteness is the load-bearing one, and its failure is silent rather than loud: a per-node aggregate over `table_contexts` folds with `Finite_Set.fold`, which answers with its unit on an infinite carrier. The contextual check report's verdict aggregate is one such fold: over an infinite key set it reports a check dead whatever its contexts say.

Canonicality is what licenses reading `Bot` as concrete emptiness rather than as the solver's own structural answer. `Lifted` gets no such reading: the supplied emptiness predicate is sound, not complete. It is stated against a supplied emptiness predicate, the same way the normalization operations upstream are, so this layer keeps its freedom from any class constraint on the payload.

```
definition finite_solved_table :: "('ctx, 'a) solved_table ⇒ bool" where
  "finite_solved_table r ⟷ finite (covered_keys r)"

definition wf_solved_table ::
  "('a ⇒ bool) ⇒ ('ctx, 'a) solved_table ⇒ bool" where
  "wf_solved_table empty_pred r ⟷
     finite_solved_table r
   ∧ (∀v ctx st. lookup_table r v ctx = Lifted st ⟶ ¬ empty_pred st)"

lemma wf_solved_table_finite [dest]:
  "wf_solved_table empty_pred r ⟹ finite_solved_table r"
  unfolding wf_solved_table_def by simp

lemma wf_solved_table_LiftedD [dest]:
  "⟦wf_solved_table empty_pred r; lookup_table r v ctx = Lifted st⟧
     ⟹ ¬ empty_pred st"
  unfolding wf_solved_table_def by blast

lemma finite_table_contexts:
  assumes "finite_solved_table r"
  shows "finite (table_contexts r v)"
  using assms unfolding finite_solved_table_def table_contexts_def by simp
```

What canonicality buys: on a well-formed result the structural reading and the concrete one coincide, provided the supplied predicate is the exact emptiness test its adapters use.

```
lemma wf_solved_table_gamma_point_eq_empty_iff:
  fixes r :: "('ctx, 'a::numeric_domain abs_state) solved_table"
    and empty_pred :: "'a abs_state ⇒ bool"
  assumes wf: "wf_solved_table empty_pred r"
    and exact: "⋀st. empty_pred st ⟷ γ st = {}"
  shows "γ⊥ (lookup_table r v ctx) = {} ⟷ lookup_table r v ctx = Bot"
proof (cases "lookup_table r v ctx")
  case Bot
  then show ?thesis by simp
next
  case (Lifted st)
  have "¬ empty_pred st" using wf Lifted by blast
  with exact Lifted show ?thesis by simp
qed
```

end

6.25 Checks

```
theory Checks
  imports "Voblint_CFG.CFG_Def"
begin
```

Ordinary assertion checks: store-only, no ghost-domain or additional trace-projection dependency

A check table relates program points to the boolean conditions placed there; several checks may share one point. `checks_proven` states that every concrete store the caller's reachable-store function admits at a checked point satisfies that point's condition. The reachable-store function is left abstract (`pp ⇒ store set`) so an instance supplies whatever collecting semantics its own `analysis_contract` interpretation already proves sound: this theory adds no new soundness anchor, only the store-level check layer on top of an existing one.

A compiled graph's table is read off its `EA_Check` edges: each edge contributes its source node and its condition.

```
type_synonym checks = "(pp × exp) set"

definition checks_proven :: "checks ⇒ (pp ⇒ store set) ⇒ bool" where
  "checks_proven ck reach ⟷ (∀v c. (v, c) ∈ ck ⟶ (∀s ∈ reach v. truthy (⟦c⟧e s)))"

lemma checks_provenI [intro]:
  "(⋀v c s. (v, c) ∈ ck ⟹ s ∈ reach v ⟹ truthy (⟦c⟧e s)) ⟹ checks_proven ck reach"
  unfolding checks_proven_def by blast
```

The one-step destruction dual of `checks_provenI`, so a caller with a `checks_proven` fact in hand reaches the store-level conclusion without unfolding the definition. It carries no soundness content of its own, and is named for what it is rather than `_sound`: the real soundness work happens where a caller establishes `checks_proven` for a concrete `reach`, by composing this theory with an existing `node_collect` $\leq$ `gamma_state(...)`-shaped corollary; this theory never states or proves that inclusion itself.

```
lemma checks_provenD [dest]:
  assumes proven: "checks_proven ck reach"
    and at_v: "(v, c) ∈ ck"
    and mem: "s ∈ reach v"
  shows "truthy (⟦c⟧e s)"
  using proven at_v mem unfolding checks_proven_def by blast

end
```

6.26 Check_Result

```
theory Check_Result
  imports Main
begin
```

A flat three-valued check verdict

`check_result` as a flat join-semilattice: `Check_Unknown` is the top element, `Check_Proved`/`Check_Refuted` are incomparable, and their join is `Check_Unknown`. The canonical use is aggregating a check's verdict across several independently-sound sources of evidence (e.g. one abstract state per reachable calling context) without joining the underlying abstract states first: two sources agreeing stay that verdict, any disagreement collapses to `Check_Unknown` rather than asserting a verdict neither source alone established.

```
datatype check_result = Check_Proved | Check_Refuted | Check_Unknown

instantiation check_result :: semilattice_sup
begin

definition less_eq_check_result :: "check_result ⇒ check_result ⇒ bool" where
  "less_eq_check_result x y ⟷ x = y ∨ y = Check_Unknown"

definition less_check_result :: "check_result ⇒ check_result ⇒ bool" where
  "less_check_result x y ⟷ x ≤ y ∧ ¬ y ≤ x"

definition sup_check_result :: "check_result ⇒ check_result ⇒ check_result" where
  "sup_check_result x y = (if x = y then x else Check_Unknown)"

instance
  by standard
     (auto simp: less_eq_check_result_def less_check_result_def sup_check_result_def)

end
```

There is no dual bottom: `Check_Proved` and `Check_Refuted` have no common lower verdict, and the absence of any evidence at all is expressed one level up, by `check_result lifted`, not inside this type.

```
instantiation check_result :: top
begin

definition top_check_result :: check_result where
  "top_check_result = Check_Unknown"

instance ..

end

instance check_result :: order_top
  by standard (simp add: less_eq_check_result_def top_check_result_def)

end
```

6.27 Abstract_Checks

```
theory Abstract_Checks
  imports Check_Result Checks "Voblint_Domain.Numeric_Queries"
    "Voblint_Domain.Forward_Domain" "Voblint_Domain.Three_Valued"
begin
```

A domain-generic sound decision procedure for compiled checks

A check needs two capabilities every domain with an `exp` evaluator already has: `aval_abs_sound`-shaped soundness (`sound_evaluator`, the same reuse point the `sound_refinement` locale takes for its own `aval_abs`) and the relational queries of `sound_numeric_queries` (`Voblint_Domain.Numeric_Queries`). Extending `sound_numeric_queries` directly, rather than fixing four raw entailment/refutation predicates here, means there is exactly one relational query interface in this codebase – `less`/`eq` – and every check-discharge consumer of it inherits whatever a domain already proved for `sound_numeric_queries` instead of restating it.

The per-domain guard machinery (the `sound_refinement` locale's `bfilter`/`afilter`) would also decide a check: if `bfilter c False d` represents no states, `c` is soundly established on `d`, whenever `gamma bot = {}` — a sound sufficient condition, not an iff, since no completeness result for `bfilter` is proved here. It is not usable as an executable decision procedure over `'a abs_state`, though: that is a raw function `vname ⇒ 'a` over the infinite type `vname`, so its emptiness test is not code-generable.

This layer decides entailment directly off `aval_abs` results via `less`/ `eq`. The judgments are sound but intentionally incomplete: two atomic values with overlapping concretizations are neither provably related nor provably unrelated, and are reported `None`/`Check_Unknown`, never misclassified.

```
locale sound_check_query =
  sound_numeric_queries less eq + sound_evaluator γ_S aval_abs
  for less :: "'a::numeric_domain ⇒ 'a ⇒ bool option"
    and eq :: "'a ⇒ 'a ⇒ bool option"
    and γ_S :: "'d ⇒ store set"
    and aval_abs :: "exp ⇒ 'd ⇒ 'a" ("⟦_⟧♯")
begin
```

A single three-valued decision procedure over `exp`

`check_query` replaces a mutually recursive true/false judgment pair with one function into `bool option`, matching the shape `sound_numeric_queries` already gives its two atomic queries: `Some True` means definitely true, `Some False` definitely false, `None` undecided. `Not` negates through `map_option`; `And`/`Or` combine through `and_opt`/`or_opt`; `Less`/`Eq` read `less`/`eq` directly; every other expression falls back to testing it against zero through `eq`, negated – `truthy` is C's non-zero test, so an expression is true exactly when it is not equal to zero.

```
definition truthy_query :: "exp ⇒ 'd ⇒ bool option" where
 [code]: "truthy_query e d = map_option HOL.Not (eq (⟦e⟧♯ d) (⟦N 0⟧♯ d))"

lemma truthy_query_sound:
  assumes mem: "s ∈ γ_S d"
    and query: "truthy_query e d = Some r"
  shows "truthy (⟦e⟧_e s) = r"
  using assms aval_abs_sound eq_sound truthy_query_def
  by fastforce

fun check_query :: "exp ⇒ 'd ⇒ bool option" where
    "check_query (Not b) d = map_option HOL.Not (check_query b d)"
  | "check_query (And b1 b2) d = and_opt (check_query b1 d) (check_query b2 d)"
  | "check_query (Or b1 b2) d = or_opt (check_query b1 d) (check_query b2 d)"
  | "check_query (Less a b) d = less (⟦a⟧♯ d) (⟦b⟧♯ d)"
  | "check_query (LessEq a b) d = map_option HOL.Not (less (⟦b⟧♯ d) (⟦a⟧♯ d))"
  | "check_query (Greater a b) d = less (⟦b⟧♯ d) (⟦a⟧♯ d)"
  | "check_query (GreaterEq a b) d = map_option HOL.Not (less (⟦a⟧♯ d) (⟦b⟧♯ d))"
  | "check_query (Eq a b) d = eq (⟦a⟧♯ d) (⟦b⟧♯ d)"
  | "check_query (NotEq a b) d = map_option HOL.Not (eq (⟦a⟧♯ d) (⟦b⟧♯ d))"
  | "check_query e d = truthy_query e d"
```

Soundness of the arithmetic fallback, proved once and cited by every induction case it covers (`N`/`V`/`Plus`/`Minus`/`Times`). Naming the fallback `truthy_query` rather than inlining the `map_option` expression keeps `check_query.simps` a plain one-step rewrite for these five cases, so `auto` can chain straight into this lemma instead of first landing on `map_option`'s own case split.

```
theorem check_query_sound:
  assumes mem: "s ∈ γ_S d"
  shows "check_query c d = Some r ⟹ truthy (⟦c⟧_e s) = r"
using truthy_query_sound[OF mem] proof (induction c arbitrary: r)
```

```
  case (And b1 b2)
  then show ?case
    using and_opt_sound by auto
next
  case (Or b1 b2)
```

Unlike `And`, this case names the two concrete disjuncts: `or_opt_sound` quantifies over the propositions its two options stand for, and nothing in the goal determines them by unification.

```
  then show ?case
    using or_opt_sound[of "check_query b1 d" "check_query b2 d" r
        "truthy (⟦b1⟧ₑ s)" "truthy (⟦b2⟧ₑ s)"]
    by auto
next
  case (Less a b)
  then show ?case using aval_abs_sound mem less_sound by auto
next
  case (Eq a b)
  then show ?case using aval_abs_sound mem eq_sound by auto
next
  case (LessEq a b)
  have query: "less (⟦b⟧♯ d) (⟦a⟧♯ d) = Some (¬ r)"
    using LessEq.prems by (auto split: option.splits)
  have relation: "(⟦b⟧ₑ s < ⟦a⟧ₑ s) = (¬ r)"
    by (rule less_sound[OF query aval_abs_sound[OF mem] aval_abs_sound[OF mem]])
  show ?case using relation by auto
next
  case (Greater a b)
  have query: "less (⟦b⟧♯ d) (⟦a⟧♯ d) = Some r"
    using Greater.prems by simp
  have relation: "(⟦b⟧ₑ s < ⟦a⟧ₑ s) = r"
    by (rule less_sound[OF query aval_abs_sound[OF mem] aval_abs_sound[OF mem]])
  show ?case using relation by simp
next
  case (GreaterEq a b)
  have query: "less (⟦a⟧♯ d) (⟦b⟧♯ d) = Some (¬ r)"
    using GreaterEq.prems by (auto split: option.splits)
  have relation: "(⟦a⟧ₑ s < ⟦b⟧ₑ s) = (¬ r)"
    by (rule less_sound[OF query aval_abs_sound[OF mem] aval_abs_sound[OF mem]])
  show ?case using relation by auto
next
  case (NotEq a b)
  have query: "eq (⟦a⟧♯ d) (⟦b⟧♯ d) = Some (¬ r)"
    using NotEq.prems by (auto split: option.splits)
  have relation: "(⟦a⟧ₑ s = ⟦b⟧ₑ s) = (¬ r)"
    by (rule eq_sound[OF query aval_abs_sound[OF mem] aval_abs_sound[OF mem]])
  show ?case using relation by auto
qed (fastforce+)
```

Executable three-way classification

`classify_check` turns the check query into proved, refuted or unknown. A proved check holds in every concretized store and a refuted one fails in each.

```
definition classify_check :: "exp ⇒ 'd ⇒ check_result" where
  "classify_check c d =
     (case check_query c d of
        Some True ⇒ Check_Proved
      | Some False ⇒ Check_Refuted
      | None ⇒ Check_Unknown)"

lemma classify_check_proved:
  assumes "classify_check c d = Check_Proved" and "s ∈ γ_S d"
  shows "truthy (⟦c⟧ₑ s)"
  using assms check_query_sound
  unfolding classify_check_def
  by (fastforce split: option.splits bool.splits)

lemma classify_check_refuted:
  assumes "classify_check c d = Check_Refuted" and "s ∈ γ_S d"
  shows "¬ truthy (⟦c⟧ₑ s)"
  using assms check_query_sound
  unfolding classify_check_def
  by (fastforce split: option.splits bool.splits)
```

`Check_Unknown` carries no semantic claim: no lemma concludes `truthy (⟦c⟧ₑ s)` or its negation from it, by design.

Node-indexed bridge to checks_proven

```
definition abstract_checks_proven :: "checks ⇒ (pp ⇒ 'd) ⇒ bool" where
  "abstract_checks_proven ck env ⟷ (∀v c. (v, c) ∈ ck ⟶ check_query c (env v) = Some True)"

lemma abstract_checks_provenI [intro]:
  "(⋀v c. (v, c) ∈ ck ⟹ check_query c (env v) = Some True) ⟹ abstract_checks_proven ck env"
  unfolding abstract_checks_proven_def by blast

lemma abstract_checks_provenD [dest]:
  "abstract_checks_proven ck env ⟹ (v, c) ∈ ck ⟹ check_query c (env v) = Some True"
  unfolding abstract_checks_proven_def by blast

theorem abstract_checks_proven_sound:
  assumes node_sound: "⋀v c. (v, c) ∈ ck ⟹ reach v ≤ γ_S (env v)"
    and checked: "abstract_checks_proven ck env"
  shows "checks_proven ck reach"
proof (rule checks_provenI)
  fix v c s
  assume ck': "(v, c) ∈ ck" and mem: "s ∈ reach v"
  have proven: "check_query c (env v) = Some True"
    using checked ck' by blast
  have in_gamma: "s ∈ γ_S (env v)"
    using node_sound[OF ck'] mem by blast
  show "truthy (⟦c⟧_e s)"
    using check_query_sound[OF in_gamma proven] by simp
qed

end


end
```

6.28 Check_Report

```
theory Check_Report
  imports Check_Result CFG_Enumeration
begin
```

Whole-program executable check report

A single, domain-generic report over every compiled `EA_Check` edge: one entry per check, at its own source node, classified by whatever `classify` function a domain supplies (`sign_classify_check` and its siblings, or `sound_check_query.classify_check` generically). Entries are read directly off `intra` through the same deterministic order `cfg_intra_list` already gives the TD bridge, rather than sorting `pp` or `exp` values by hand, so the report has exactly one entry per `EA_Check` edge and no separate table can drift from it.

This layer is deliberately generic in `classify`: it depends on neither `Abstract_Checks` nor a specific domain, only on a plain `exp ⇒ 's ⇒ check_result` function and its own soundness obligations below.

```
type_synonym check_report_entry = "pp × exp × check_result"

definition classify_checks ::
    "cfg ⇒ (pp ⇒ 's) ⇒ (exp ⇒ 's ⇒ check_result) ⇒ check_report_entry list" where
  "classify_checks g env classify =
     map (λ(u, a, v). (u, ea_check_cond a, classify (ea_check_cond a) (env u)))
       (filter (λ(u, a, v). is_EA_Check a) (cfg_intra_list g))"
```

Membership unfolds to an `EA_Check` edge at the entry's own source node, classified there — the shape every soundness and worked-example use of `classify_checks` goes through.

```
lemma classify_checks_mem_iff:
  assumes "finite (intra g)"
  shows "(v, c, r) ∈ set (classify_checks g env classify)
     ⟷ (∃l tgt. (v, EA_Check l c, tgt) ∈ intra g) ∧ r = classify c (env v)"
  unfolding classify_checks_def set_map set_filter
  using set_cfg_intra_list[OF assms]
  by (auto simp: image_iff split: edge_action.splits)
     (metis edge_action.collapse surjective_pairing)
```

Soundness bridge: a `Check_Proved` report entry's condition genuinely holds at every reaching store, and a `Check_Refuted` entry's condition genuinely fails — given the same two facts every per-domain instance already proves once: a `classify_check_proved`/`classify_check_refuted`-shaped soundness obligation for the domain's own `classify`, and node-local collecting soundness (`reach v ≤ γ_S (env v)`) for the checked node. Neither theorem invents new domain reasoning; both only relocate an existing per-node fact to every entry of the whole-program report. `Check_Unknown` gets no counterpart, matching `classify_check` itself.

```
theorem classify_checks_proved_sound:
  fixes γ_S :: "'s ⇒ store set"
  assumes fin: "finite (intra g)"
    and mem: "(v, c, Check_Proved) ∈ set (classify_checks g env classify)"
    and classify_proved: "⋀d s. classify c d = Check_Proved ⟹ s ∈ γ_S d ⟹ truthy (⟦c⟧_e s)"
    and node_sound: "reach v ≤ γ_S (env v)"
  shows "∀s ∈ reach v. truthy (⟦c⟧_e s)"
proof
  fix s assume s: "s ∈ reach v"
  have "classify c (env v) = Check_Proved"
    using mem classify_checks_mem_iff[OF fin, of v c Check_Proved env classify] by auto
  moreover have "s ∈ γ_S (env v)" using node_sound s by blast
  ultimately show "truthy (⟦c⟧_e s)" using classify_proved by blast
qed

theorem classify_checks_refuted_sound:
  fixes γ_S :: "'s ⇒ store set"
  assumes fin: "finite (intra g)"
    and mem: "(v, c, Check_Refuted) ∈ set (classify_checks g env classify)"
    and classify_refuted: "⋀d s. classify c d = Check_Refuted
                           ⟹ s ∈ γ_S d ⟹ ¬ truthy (⟦c⟧_e s)"
    and node_sound: "reach v <= γ_S (env v)"
  shows "∀s ∈ reach v. ~ truthy (⟦c⟧_e s)"
proof
  fix s assume s: "s : reach v"
  have "classify c (env v) = Check_Refuted"
    using mem classify_checks_mem_iff[OF fin, of v c Check_Refuted env classify] by auto
  moreover have "s : γ_S (env v)" using node_sound s by blast
  ultimately show "~ truthy (⟦c⟧_e s)" using classify_refuted by blast
qed
```

State-carrying sibling of `classify_checks`: the same traversal and classification, with the node's own abstract state attached to each entry. `classify_checks` stays the soundness-critical primitive every domain cites directly; the projection lemma below shows this report is exactly that one with a fourth field added, so `classify_checks_proved_sound` and `classify_checks_refuted_sound` reach it through the projection without restatement.

```
definition classify_checks_with_state ::
    "cfg ⇒ (pp ⇒ 's) ⇒ (exp ⇒ 's ⇒ check_result) ⇒ (pp × exp × check_result × 's) list" where
  "classify_checks_with_state g env classify =
     map (λ(u, c, r). (u, c, r, env u)) (classify_checks g env classify)"

lemma classify_checks_with_state_proj [simp]:
  "map (λ(u, c, r, s). (u, c, r)) (classify_checks_with_state g env classify)
     = classify_checks g env classify"
  unfolding classify_checks_with_state_def by (simp add: comp_def case_prod_beta)

end
```

6.29 Contextual_Check_Report

```
theory Contextual_Check_Report
  imports Check_Report Solved_Table Analysis_Query
begin
```

Contextual check verdicts over a solved result table

`classify_checks` reads exactly one abstract state per checked node. A context-sensitive result has one state per `(node, context)` pair instead, and some of those pairs represent no concrete execution at all: a branch that is dead inside one activation, or an activation a node is never reached under. Classifying such a point against its stored state answers vacuously: a classifier's soundness obligation only constrains the stores its state represents, so on a state representing none, either definite answer satisfies that obligation vacuously — a verdict read there is evidence of nothing, and reporting it as `Check_Proved` would fabricate a proof about executions that do not exist.

`contextual_verdict` keeps that case outside `check_result` rather than folding it into one of its three values. `Dead` means "no concrete execution is represented here"; it is neither `Check_Unknown`, which says only that the abstraction decided the condition neither way on a state representing some stores — it asserts no reachability, since those stores may over-approximate a point no execution reaches — nor `Check_Proved`.

`contextual_verdict` is literally `check_result lifted` (`Voblint_Domain.Reachability_Lift`): `Dead` is `Bot`, `Decided` is `Lifted`, and its order/bottom/join are `'a lifted`'s own generic instance at `check_result` (itself `semilattice_sup`, from `Voblint_Framework.Check_Result`) constructor for constructor: `Dead` $\leq$ `_` always, a `Decided` never sits below `Dead`, and two `Decided` verdicts compare and join through `check_result`'s own flat order. Reusing it here keeps one lattice construction for "structurally absent vs. present with a payload" instead of two.

```
type_synonym contextual_verdict = "check_result lifted"

abbreviation Dead :: contextual_verdict where
  "Dead ≡ Bot"

abbreviation Decided :: "check_result ⇒ contextual_verdict" where
  "Decided r ≡ Lifted r"
```

Classifying one contextual point

The reachability decision is not remade here: it was already made when the raw solved local unknown crossed the result boundary, one layer before the lifted readback relabels it structurally. A witness-bottom `Lifted` payload is collapsed to `Bot` by `canonicalize_lift`, so by the time a value is read back — as every public result adapter's raw value is — `Bot` already denotes no store. A `Lifted` state may still denote none, since the emptiness test is sound and incomplete, and classifying it is then vacuous but harmless. `classify_point` only refuses to classify against a state that represents nothing: a covered-but-dead activation, whose raw stored state a solver run could otherwise leave as a witness-bottom `Lifted`, reaches `classify_point` as `Bot` rather than fabricating a vacuous `Check_Proved` off an empty state.

```
fun classify_point ::
  "(exp ⇒ 'a ⇒ check_result) ⇒ exp ⇒ 'a lifted ⇒ contextual_verdict"
where
  "classify_point classify c Bot = Dead"
| "classify_point classify c (Lifted st) = Decided (classify c st)"
```

Classifying by query

A check `c` is decided by one question, the value of its truth as an integer, which every analysis in a product answers and whose combined answer is the meet. Goblint's `assert` analysis asks `EvalInt (c != 0)` through `Queries.eval_bool`; a comparison or logical operator already evaluates to `0` or `1`, so for those Voblint asks about `c` itself and an analysis sees the comparison rather than its test against zero. An exact `1` proves the check and an exact `0` refutes it. `QBot` holds at no store, so a sound combined answer is `QBot` only where the state represents no store, and `classify_answer` reads it as `Dead`. Anything else decides nothing.

```
fun bool_valued :: "exp ⇒ bool" where
  "bool_valued (Not _) = True"
| "bool_valued (And _ _) = True"
| "bool_valued (Or _ _) = True"
| "bool_valued (Less _ _) = True"
| "bool_valued (LessEq _ _) = True"
| "bool_valued (Greater _ _) = True"
| "bool_valued (GreaterEq _ _) = True"
| "bool_valued (Eq _ _) = True"
```

```
| "bool_valued (NotEq _ _) = True"
| "bool_valued _ = False"

lemma aval_bool_valued:
  "bool_valued e ⟹ ⟦e⟧e s = (if truthy (⟦e⟧e s) then 1 else 0)"
  by (cases e) auto

definition check_truth_exp :: "exp ⇒ exp" where
  "check_truth_exp c = (if bool_valued c then c else NotEq c (N 0))"

lemma bool_valued_check_truth_exp [simp]: "bool_valued (check_truth_exp c)"
  by (simp add: check_truth_exp_def)

lemma aval_check_truth_exp:
  "⟦check_truth_exp c⟧e s = (if truthy (⟦c⟧e s) then 1 else 0)"
  by (auto simp: check_truth_exp_def dest: aval_bool_valued[of c s])

abbreviation check_query_of :: "exp ⇒ query" where
  "check_query_of c ≡ EvalInt (check_truth_exp c)"

fun classify_answer :: "answer ⇒ contextual_verdict" where
  "classify_answer QBot = Dead"
| "classify_answer a =
     (if answer_const a = Some 1 then Decided Check_Proved
      else if answer_const a = Some 0 then Decided Check_Refuted
      else Decided Check_Unknown)"

lemma classify_answer_proved:
  assumes "classify_answer a = Decided Check_Proved"
    and "eval_holds (check_query_of c) a s"
  shows "truthy (⟦c⟧e s)"
proof -
  have "answer_const a = Some 1"
    using assms(1) by (cases a rule: classify_answer.cases)
      (auto split: option.splits if_splits simp del: answer_const.simps)
  from eval_holds_constD[OF assms(2) this] show ?thesis
    by (simp add: aval_check_truth_exp split: if_splits)
qed

lemma classify_answer_refuted:
  assumes "classify_answer a = Decided Check_Refuted"
    and "eval_holds (check_query_of c) a s"
  shows "¬ truthy (⟦c⟧e s)"
proof -
  have "answer_const a = Some 0"
    using assms(1) by (cases a rule: classify_answer.cases)
      (auto split: option.splits if_splits simp del: answer_const.simps)
  from eval_holds_constD[OF assms(2) this] show ?thesis
    by (simp add: aval_check_truth_exp split: if_splits)
qed

lemma classify_answer_dead:
  assumes "classify_answer a = Dead"
  shows "¬ eval_holds (EvalInt e) a s"
  using assms by (cases a) (auto split: if_splits)
```

Aggregating the contexts observed at one check

Folding (⊔) from `Dead` over a finite set of observations. The fold's unit is the empty-set answer, so a check with no covered context needs no separate guard. The `[code]` equation is the usual `comp_fun_idem` transfer to a list fold: over an unordered set the value is well defined only because (⊔) is commutative, associative, and idempotent.

```
definition aggregate_verdicts :: "contextual_verdict set ⇒ contextual_verdict" where
  "aggregate_verdicts vs = Finite_Set.fold sup Dead vs"

lemma aggregate_verdicts_code [code]:
  "aggregate_verdicts (set vs) = List.fold sup vs Dead"
proof -
  interpret ci: comp_fun_idem "sup :: contextual_verdict ⇒ _"
    by (rule comp_fun_idem_sup)
  show ?thesis unfolding aggregate_verdicts_def by (rule ci.fold_set_fold)
qed

lemma aggregate_verdicts_empty [simp]: "aggregate_verdicts {} = Dead"
  unfolding aggregate_verdicts_def by simp
```

```
lemma aggregate_verdicts_insert [simp]:
  "finite vs ⟹ aggregate_verdicts (insert v vs) = v ⊔ aggregate_verdicts vs"
proof -
  assume "finite vs"
  interpret ci: comp_fun_idem "sup :: contextual_verdict ⇒ _"
    by (rule comp_fun_idem_sup)
  show ?thesis
    unfolding aggregate_verdicts_def using ‹finite vs› by simp
qed

lemma sup_contextual_verdict_eq_Dead_iff [simp]:
  "(x ⊔ y = Dead) ⟷ x = Dead ∧ y = Dead"
  by (cases x; cases y) simp_all
```

The finiteness premise is not bookkeeping: `Finite_Set.fold` returns its identity on an infinite carrier, so an infinite set of observations aggregates to `Dead` whatever its members are, and only the left-to-right direction would survive without it.

```
lemma aggregate_verdicts_eq_Dead_iff:
  assumes "finite vs"
  shows "aggregate_verdicts vs = Dead ⟷ (∀v ∈ vs. v = Dead)"
  using assms by (induction vs rule: finite_induct) simp_all
```

The opposite direction, for a report that has only one observation per check and therefore no dead case to distinguish: every entry is `Decided`. Deadness is not asserted either way — such a report simply does not carry the channel that would express it.

```
definition decided_report ::
  "check_report_entry list ⇒ (pp × exp × contextual_verdict) list" where
  "decided_report = map (λ(u, cnd, r). (u, cnd, Decided r))"

lemma length_decided_report [simp]: "length (decided_report rs) = length rs"
  unfolding decided_report_def by simp
```

Whole-program contextual check report

The context-sensitive sibling of `classify_checks`: the same `EA_Check` traversal in the same `cfg_intra_list` order, but reading a `('ctx, 'a) solved_table` instead of a single `pp ⇒ 's` environment, and retaining one verdict per context covered at the checked node rather than collapsing them at construction time. `classify_checks_ctx_positions` below is the load-bearing consequence: the node and condition columns coincide with `classify_checks`'s, so a positional consumer pairing this report with the source's own check list stays aligned.

The per-check observations are a set, not a list. `'ctx` carries no ordering constraint — `table_contexts` is a set for precisely that reason, and real context types such as interval vectors have no total order — so no canonical list exists to produce. Nothing downstream needs one: `aggregate_verdicts` is order-independent by construction.

```
definition classify_checks_ctx ::
    "cfg ⇒ ('ctx, 'a) solved_table ⇒ (exp ⇒ 'a ⇒ check_result)
       ⇒ (pp × exp × ('ctx × contextual_verdict) set) list" where
  "classify_checks_ctx g r classify =
     map (λ(u, a, v). (u, ea_check_cond a,
            (λctx. (ctx, classify_point classify (ea_check_cond a) (lookup_table r u ctx)))
              ‘ table_contexts r u))
       (filter (λ(u, a, v). is_EA_Check a) (cfg_intra_list g))"

definition classify_checks_verdicts ::
    "cfg ⇒ ('ctx, 'a) solved_table ⇒ (exp ⇒ 'a ⇒ check_result)
       ⇒ (pp × exp × contextual_verdict) list" where
  "classify_checks_verdicts g r classify =
     map (λ(u, c, vs). (u, c, aggregate_verdicts (snd ‘ vs))) (classify_checks_ctx g r classify)"

lemma classify_checks_verdicts_proj [simp]:
  "map (λ(u, c, vs). (u, c, aggregate_verdicts (snd ‘ vs))) (classify_checks_ctx g r classify)
     = classify_checks_verdicts g r classify"
  unfolding classify_checks_verdicts_def by (rule refl)

lemma classify_checks_ctx_positions:
  "map (λ(u, c, vs). (u, c)) (classify_checks_ctx g r classify)
     = map (λ(u, c, res). (u, c)) (classify_checks g env classify')"
  unfolding classify_checks_ctx_def classify_checks_def
  by (simp add: comp_def case_prod_beta)
```

Membership unfolds to an `EA_Check` edge at the entry's own source node, with that node's whole context fan-out attached — the same shape `classify_checks_mem_iff` gives the context-insensitive report.

```
lemma classify_checks_ctx_mem_iff:
  assumes "finite (intra g)"
  shows "(v, c, vs) ∈ set (classify_checks_ctx g r classify)
     ⟷ (∃l tgt. (v, EA_Check l c, tgt) ∈ intra g)
         ∧ vs = (λctx. (ctx, classify_point classify c (lookup_table r v ctx))) ‘ table_contexts r v"
  unfolding classify_checks_ctx_def set_map set_filter
  using set_cfg_intra_list[OF assms]
  by (auto simp: image_iff split: edge_action.splits)
     (metis edge_action.collapse surjective_pairing)
```

Contextual proved/refuted soundness

The `Decided Check_Proved`/`Decided Check_Refuted` analogues of `classify_checks_proved_sound`/`classify_checks_refuted_sou` an aggregated verdict of `Decided Check_Proved` forces every observation folded into it to be `Dead` or `Decided Check_Proved` — never a disagreeing `Decided Check_Refuted` or `Decided Check_Unknown`, since either would collapse the join to `Decided Check_Unknown` instead. `Dead` observations contribute nothing and are filtered out at the call site, not here.

The order-theoretic route: every member of a finite set sits below the set's own fold-join (`aggregate_verdicts_member_le`), a decided, non-`Check_Unknown` aggregate already forces the set finite (`aggregate_verdicts_decided_finite`, since an infinite carrier folds to the identity `Dead`), and sitting below such a verdict has only two shapes (`contextual_verdict_le_Deci` Composing the three gives `aggregate_verdicts_decided_dest` directly, with no case-bash over the fold step: disagreement, from either side, could only have collapsed the join to `Decided Check_Unknown` instead.

```
lemma aggregate_verdicts_member_le:
  assumes "finite vs" and "v ∈ vs"
  shows "v ≤ aggregate_verdicts vs"
  using assms
  by (induction vs rule: finite_induct) (auto intro: order_trans sup_ge2)

lemma contextual_verdict_le_Decided_iff [simp]:
  assumes "r ≠ Check_Unknown"
  shows "v ≤ Decided r ⟷ v = Dead ∨ v = Decided r"
  using assms by (cases v; cases r) (simp_all add: less_eq_check_result_def)

lemma aggregate_verdicts_decided_finite:
  "aggregate_verdicts vs = Decided r ⟹ finite vs"
  by (rule ccontr) (simp add: aggregate_verdicts_def)

lemma aggregate_verdicts_decided_dest:
  assumes agg: "aggregate_verdicts vs = Decided r" and known: "r ≠ Check_Unknown"
  shows "∀v ∈ vs. v = Dead ∨ v = Decided r"
proof
  fix v
  assume "v ∈ vs"
  have "v ≤ aggregate_verdicts vs"
    using aggregate_verdicts_member_le[OF aggregate_verdicts_decided_finite[OF agg] ‹v ∈ vs›] .
  with agg known show "v = Dead ∨ v = Decided r" by simp
qed
```

One condition at one point, over its contexts

The verdict a table gives a condition at a point: the classifier's verdict at every context the point was solved at, aggregated. A check and an arithmetic obligation are both read this way.

```
definition point_verdict ::
    "('ctx, 'a) solved_table ⇒ (exp ⇒ 'a ⇒ check_result) ⇒ pp ⇒ exp
       ⇒ contextual_verdict" where
  "point_verdict r classify v e =
     aggregate_verdicts
       ((λctx. classify_point classify e (lookup_table r v ctx)) ‘ table_contexts r v)"
```

A decided, non-`Check_Unknown` point verdict is the classifier's own reading at every `Lifted` context of the point.

```
lemma point_verdict_decided:
  assumes agg: "point_verdict r classify v e = Decided res" and known: "res ≠ Check_Unknown"
    and look: "lookup_table r v ctx = Lifted st"
  shows "classify e st = res"
proof -
  have "classify_point classify e (lookup_table r v ctx)
          ∈ (λc. classify_point classify e (lookup_table r v c)) ‘ table_contexts r v"
    using lookup_table_LiftedD[OF look] by blast
  with aggregate_verdicts_decided_dest[OF agg[unfolded point_verdict_def] known] look
  show ?thesis by auto
```

```
qed
```

A point with a `Lifted` entry among finitely many contexts is not `Dead`.

```
lemma point_verdict_not_dead:
  assumes fin: "finite (table_contexts r v)" and look: "lookup_table r v ctx = Lifted st"
  shows "point_verdict r classify v e ≠ Dead"
proof
  assume "point_verdict r classify v e = Dead"
  with fin have "∀x ∈ (λc. classify_point classify e (lookup_table r v c)) ‘ table_contexts r v.
                   x = Dead"
    unfolding point_verdict_def by (simp add: aggregate_verdicts_eq_Dead_iff)
  then have "classify_point classify e (lookup_table r v ctx) = Dead"
    using lookup_table_LiftedD[OF look] by blast
  with look show False by simp
qed
```

The context-indexed report analogue of `classify_checks_proved_sound`/ `classify_checks_refuted_sound`, stated once for any decided, non-`Check_Unknown` aggregate: it forces the classifier's own reading at every `Lifted` context to agree. The image set being folded is shown finite from the aggregate equation itself (`Finite_Set.fold_infinite`: an infinite carrier folds to the identity `Dead`, which a decided, non-`Check_Unknown` result already rules out), so no separate finiteness assumption on the covered contexts is needed.

```
lemma classify_checks_verdicts_mem_iff:
  assumes "finite (intra g)"
  shows "(v, c, vr) ∈ set (classify_checks_verdicts g ar classify)
     ⟷ (∃l tgt. (v, EA_Check l c, tgt) ∈ intra g)
         ∧ vr = aggregate_verdicts
                  ((λctx. classify_point classify c (lookup_table ar v ctx)) ‘ table_contexts ar v)"
proof -
  have img: "snd ‘ (λctx. (ctx, classify_point classify c (lookup_table ar v ctx)))
                ‘ table_contexts ar v
           = (λctx. classify_point classify c (lookup_table ar v ctx)) ‘ table_contexts ar v"
    by (simp add: image_comp comp_def)
  have "(v, c, vr) ∈ set (classify_checks_verdicts g ar classify)
      ⟷ (∃vs. (v, c, vs) ∈ set (classify_checks_ctx g ar classify)
                ∧ vr = aggregate_verdicts (snd ‘ vs))"
    unfolding classify_checks_verdicts_def set_map by (force simp: image_iff)
  also have "... ⟷ (∃l tgt. (v, EA_Check l c, tgt) ∈ intra g)
         ∧ vr = aggregate_verdicts
                  ((λctx. classify_point classify c (lookup_table ar v ctx)) ‘ table_contexts ar v)"
    unfolding classify_checks_ctx_mem_iff[OF assms] using img by auto
  finally show ?thesis .
qed

theorem classify_checks_ctx_decided_sound:
  assumes fin: "finite (intra g)"
    and mem: "(v, c, Decided r) ∈ set (classify_checks_verdicts g ar classify)"
    and reach: "lookup_table ar v ctx = Lifted st"
    and known: "r ≠ Check_Unknown"
  shows "classify c st = r"
proof -
  have "point_verdict ar classify v c = Decided r"
    using classify_checks_verdicts_mem_iff[OF fin, of v c "Decided r" ar classify] mem
    by (simp add: point_verdict_def)
  then show ?thesis by (rule point_verdict_decided[OF _ known reach])
qed

end
```

6.30 Check_Answer

```
theory Check_Answer
  imports Abstract_Checks Contextual_Check_Report
begin
```

A domain's answer to the value of an expression

A domain with a check decision procedure answers `EvalInt e` for every comparison or logical operator `e`: such an expression evaluates to `0` or `1`, so a decided `sound_check_query.check_query` is the exact integer, and an undecided one is the interval `[0, 1]`. Other expressions it declines with $\top$. A check is then classified from this answer exactly as the domain's own `classify_check` classified it.

```
definition bool_answer :: "bool option ⇒ answer" where
  "bool_answer r =
     (case r of
        Some b ⇒ answer_of_int (if b then 1 else 0)
      | None ⇒ answer_of_ivl (Ivl (Fin 0) (Fin 1)))"

lemma classify_answer_of_int:
  "classify_answer (answer_of_int n) =
     Decided (if n = 1 then Check_Proved else if n = 0 then Check_Refuted else Check_Unknown)"
  using answer_const_of_int[of n] unfolding answer_of_int_def by simp

lemma answer_const_bool_range: "answer_const (answer_of_ivl (Ivl (Fin 0) (Fin 1))) = None"
proof -
  have "congruence_constant (⊤ :: congruence) = None"
    by (cases "congruence_constant (⊤ :: congruence)")
       (auto dest!: gamma_congruence_constant simp: set_eq_iff, presburger)
  then show ?thesis
    by (simp add: answer_of_ivl_def int_dom_constant_def top_int_dom_ext_def top_sign_def)
qed

lemma classify_bool_answer:
  "classify_answer (bool_answer r) =
     Decided (case r of Some True ⇒ Check_Proved | Some False ⇒ Check_Refuted
                      | None ⇒ Check_Unknown)"
proof (cases r)
  case None
  then show ?thesis
    using answer_const_bool_range by (simp add: bool_answer_def answer_of_ivl_def)
qed (auto simp: bool_answer_def classify_answer_of_int)

context sound_check_query
begin

fun eval_answer :: "'d ⇒ query ⇒ answer" where
  "eval_answer d (EvalInt e) = (if bool_valued e then bool_answer (check_query e d) else ⊤)"

lemma eval_answer_sound:
  assumes mem: "s ∈ γ_S d"
  shows "eval_holds q (eval_answer d q) s"
proof (cases q)
  case (EvalInt e)
  show ?thesis
  proof (cases "bool_valued e")
    case True
    have v: "⟦e⟧_e s = (if truthy (⟦e⟧_e s) then 1 else 0)"
      by (rule aval_bool_valued[OF True])
    show ?thesis
    proof (cases "check_query e d")
      case None
      then show ?thesis
        using EvalInt True v by (simp add: bool_answer_def split: if_splits)
    next
      case (Some b)
      have "truthy (⟦e⟧_e s) = b"
        by (rule check_query_sound[OF mem Some])
      then show ?thesis
        using EvalInt True v Some by (simp add: bool_answer_def split: if_splits)
    qed
  qed (simp add: EvalInt)
qed

lemma check_query_check_truth_exp: "check_query (check_truth_exp c) d = check_query c d"
  by (cases c) (simp_all add: check_truth_exp_def truthy_query_def)
```

```
theorem classify_eval_answer:
  "classify_answer (eval_answer d (check_query_of c)) = Decided (classify_check c d)"
  by (simp add: check_query_check_truth_exp classify_bool_answer classify_check_def
      split: option.splits bool.splits)

end
```

Deciding a check from an answer

The one check consumer: ask the state's handler the check's question and read the verdict off the answer, as Goblint's `assert` analysis reads `Queries.eval_bool`. `classify_answer` reads an answer that admits no value as `Dead`, but the consumer reports it as `Check_Unknown`: a check is reported dead only where the state itself is `Bot`, never from the answer to its query.

```
definition answer_check :: "('v ⇒ query ⇒ answer) ⇒ exp ⇒ 'v ⇒ check_result" where
  "answer_check q c v =
     (case classify_answer (q v (check_query_of c)) of Decided r ⇒ r | _ ⇒ Check_Unknown)"

lemma answer_check_proved:
  assumes "answer_check q c v = Check_Proved"
    and "eval_holds (check_query_of c) (q v (check_query_of c)) s"
  shows "truthy (⟦c⟧e s)"
  using assms classify_answer_proved
  by (auto simp: answer_check_def split: lifted.splits)

lemma answer_check_refuted:
  assumes "answer_check q c v = Check_Refuted"
    and "eval_holds (check_query_of c) (q v (check_query_of c)) s"
  shows "¬ truthy (⟦c⟧e s)"
  using assms classify_answer_refuted
  by (auto simp: answer_check_def split: lifted.splits)

lemma (in sound_check_query) answer_check_eval_answer:
  "answer_check eval_answer = classify_check"
  by (simp add: fun_eq_iff answer_check_def classify_eval_answer del: eval_answer.simps)

end
```

6.31 DG_Analysis_Adapter

```
theory DG_Analysis_Adapter
  imports Routed_Context Contextual_Check_Report
begin
```

Public result and check-report adapter for a routed DG analysis

Every concrete routed-context analysis needs the same public `solved_table`/check report/soundness triple, and each one follows the identical shape once a routed D/G equation system is solved: read the local unknown at every covered `(node, context)` pair, together with the analysis-wide global unknown, into an `solved_table`, classify every compiled check against it via `classify_checks_verdicts`, and discharge that report's own soundness from the activation-indexed collecting semantics (`Voblint_Framework.Activation_Backbone`) already available once EDGE/CALL/COMB are in hand from `routed_context`. `dg_analysis_adapter` derives that whole triple once, generic in a domain instance (a `classify` function with its own soundness obligations), a context instance (an interpretation of `routed_context`), and the published value (a publication map `rd` from a local and a global value of the solver's carrier into a value type `'v` with its own concretization γ_V and exact emptiness test $\texttt{empty}_V$; a non-relational analysis publishes its abstract store), leaving solver choice orthogonal: which concrete solver produced `sigma`/`sg` is an interpretation-site argument, never a locale parameter, matching how Interval's own Warrow/join/per-origin solver choice is already orthogonal to its context.

`gammaDG_rd` asks that publishing a local value together with a global one keeps exactly the stores the pair denotes, γ_{DG} `d g' = gamma_lift` γ_V `(rd d g')`. The global value the table pairs with every local one is the solved global unknown at `Inr buffer_key`, the pairing the coverage assumption `sg_cov` of `dg_context_activation` concretizes. A concretization that reads both components, an ownership split say, therefore interprets this locale directly. An analysis whose concretization ignores the global half instantiates `rd` at a function constant in its second argument.

```
locale dg_analysis_adapter =
  routed_context S γ_DG 𝒢 g buffer_key global_of route bot0 d₀ g₀ sigma vars x0 sg
    seed
    "static_resolve g" is_bot γ_M adm
  for S :: "(pp × 'c, 'k, 'n, 'D::bounded_semilattice_sup_bot,
             'G::bounded_semilattice_sup_bot) dg_spec"
    and γ_DG :: "'D ⇒ ('n ⇒ 'G) ⇒ store set"
    and 𝒢 :: "vname ⇒ bool"
    and g buffer_key and global_of :: "'n ⇒ 'k"
    and route :: "pp ⇒ 'c ⇒ 'D ⇒ call_action ⇒ 'c"
    and bot0 d₀ :: 'D and g₀ :: 'G
    and sigma :: "pp × 'c + 'k ⇒ ('D, 'G) dg_state"
    and vars :: "(pp × 'c) set"
    and x0 :: "pp × 'c"
    and sg :: "pp × 'c + 'k ⇒ 'M"
    and seed :: "pp ⇒ 'c ⇒ 'k"
    and is_bot :: "'D ⇒ bool"
    and γ_M :: "'M ⇒ store set"
    and adm :: "'c context_policy" +
  fixes rd :: "'D ⇒ ('n ⇒ 'G) ⇒ 'v lifted"
    and γ_V :: "'v ⇒ store set"
    and empty_V :: "'v ⇒ bool"
    and classify :: "exp ⇒ 'v ⇒ check_result"
  assumes gammaDG_rd: "⋀d g'. γ_DG d g' = gamma_lift γ_V (rd d g')"
    and empty_sound: "⋀v. empty_V v ⟹ γ_V v = {}"
    and classify_proved:
    "⋀c d s. classify c d = Check_Proved ⟹ s ∈ γ_V d ⟹ truthy (⟦c⟧ₑ s)"
    and classify_refuted:
    "⋀c d s. classify c d = Check_Refuted ⟹ s ∈ γ_V d ⟹ ¬ truthy (⟦c⟧ₑ s)"
    and vars_finite: "finite vars"
begin
```

The public result table

Built from the locale's own solved `vars`/`sigma` pair, mirroring how `dg_result_for` builds an `solved_table` from an already-solved key set and reader. The solved local unknown and the solved global at `Inr buffer_key` are published into `'v lifted` by `rd`, and the one collapse this table then needs is `canonicalize_lift` against $\texttt{empty}_V$, so a `Lifted` payload that denotes no store without the solver's own `Bot` tag reads as `Bot` here, matching `classify_point`'s documented discipline of never fabricating a vacuous `Check_Proved` off an empty state.

```
definition analyse_result :: "('c, 'v) solved_table" where
  "analyse_result = Solved_Table vars
     (λv ctx. canonicalize_lift empty_V
```

```
       (rd (dg_local (sigma (Inl (v, ctx)))) (genv global_of sigma)))"

lemma lookup_table_analyse_result:
  "lookup_table analyse_result v ctx =
     (if (v, ctx) ∈ vars
      then canonicalize_lift empty_V
             (rd (dg_local (sigma (Inl (v, ctx)))) (genv global_of sigma))
      else Bot)"
  unfolding lookup_table_def analyse_result_def by simp
```

Canonicality holds unconditionally here: `canonicalize_lift` is exactly what rules out a `Lifted` payload that is secretly empty, and it runs on every entry the table publishes.

Finiteness cannot be proved here, because `vars` is a fixed arbitrary set and this locale deliberately does not know which solver produced it — that is what keeps it independent of solver choice. It is therefore an assumption, `vars_finite`, and an interpretation discharges it from the solver it actually ran: `finite_stabl_solve` says a terminating solve stabilizes finitely many unknowns, and the interpretation site is the only place that knows `vars` is that stable set.

Nothing about the context type enters, which is why entry-state contexts are covered as readily as monovariant ones despite their context space being unbounded.

```
lemma analyse_result_canonical:
  assumes "lookup_table analyse_result v ctx = Lifted st"  shows "¬ empty_V st"
  using assms
  unfolding lookup_table_analyse_result
  by (cases "rd (dg_local (sigma (Inl (v, ctx)))) (genv global_of sigma)")
     (auto simp: normalize_lift_def split: if_splits)

theorem wf_analyse_result: "wf_solved_table empty_V analyse_result"
  unfolding wf_solved_table_def finite_solved_table_def
  using vars_finite analyse_result_canonical by (simp add: analyse_result_def)
```

The soundness bridge γ_M `(sg (Inl (v, ctx)))` steps need: a covered point's guarded reading agrees exactly with `analyse_result`'s own `lookup_table` answer at the same key, `Bot` concretizing to the empty set and `Lifted` to γ_V of its payload.

Collapsing a payload that denotes nothing to `Bot` does not change what the value denotes.

```
lemma gamma_lift_canonicalize [simp]:
  "gamma_lift γ_V (canonicalize_lift empty_V x) = gamma_lift γ_V x"
  by (cases x) (simp_all add: normalize_lift_def empty_sound)

lemma gammaM_sg_eq_lookup_table:
  assumes cov: "(v, ctx) ∈ vars"
  shows "γ_M (sg (Inl (v, ctx))) = gamma_lift γ_V (lookup_table analyse_result v ctx)"
proof -
  have "γ_M (sg (Inl (v, ctx))) = γ_DG (dg_local (sigma (Inl (v, ctx))))
          (genv global_of sigma)"
    using cov by simp
  also have "... = gamma_lift γ_V
                     (rd (dg_local (sigma (Inl (v, ctx)))) (genv global_of sigma))"
    by (rule gammaDG_rd)
  also have "... = gamma_lift γ_V (lookup_table analyse_result v ctx)"
    using cov unfolding lookup_table_analyse_result by simp
  finally show ?thesis .
qed
```

The node-soundness bridge a concrete instance's own report-soundness proof needs, phrased directly against `analyse_result` rather than against `analyse_report_ctx` (defined below): every activation-collected store at `v` is concretized by `analyse_result`'s own reading, $\bot$ when uncovered or witness-bottom, its `Lifted` payload otherwise — composing `activation_collect_dg_sound` with `gammaM_sg_eq_lookup_table`, case-split on coverage. A concrete instance whose own public result reads through this same `analyse_result` (up to a proved value equality) gets its own node-soundness bridge from this lemma directly, instead of re-deriving it from `routed_context`'s primitives by hand.

```
lemma analyse_result_node_sound:
  fixes S0 :: "store set" and c_0 :: 'c
  assumes entry_cov: "(cfg_entry g, c_0) ∈ vars"
    and init_sound: "S0 ⊆ γ_DG d_0 e_0"
    and init_env_le: "e_0 ≤ genv global_of sigma"
  shows "A_G,adm,c_0,g,S0 v ctx
           ⊆ gamma_lift γ_V (lookup_table analyse_result v ctx)"
proof -
  have "A_G,adm,c_0,g,S0 v ctx
```

```
        ⊆ γ_M (sg (Inl (v, ctx)))"
    by (rule activation_collect_dg_sound[OF entry_cov init_sound init_env_le])
  also have "... = gamma_lift γ_V (lookup_table analyse_result v ctx)"
  proof (cases "(v, ctx) ∈ vars")
    case True
    show ?thesis by (rule gammaM_sg_eq_lookup_table[OF True])
  next
    case False
    hence "γ_M (sg (Inl (v, ctx))) = {}" by simp
    moreover from False have "lookup_table analyse_result v ctx = Bot"
      unfolding lookup_table_analyse_result by simp
    ultimately show ?thesis by simp
  qed
  finally show ?thesis by simp
qed
```

`analyse_result_node_sound` is stated in the vocabulary `Voblint_Framework.Solved_Table` states its lookup facts in: every covered lookup safely abstracts the activation-collected stores at its key. It is the general statement, with no bottom case baked in — the unreachability reading is one instantiation of it, below.

Note what the coverage case split above establishes and this inherits: inside this locale an *uncovered* key is not merely uninformative, since γ_M `(sg (Inl (v, ctx)))` is empty off `vars` and the collect is contained in it either way. That is a property of a solve satisfying this locale's assumptions, not of `lookup_table`, which is why the unreachability corollary still demands `Covered Bot`: a caller reading a result table without these assumptions in hand has no such guarantee.

```
lemma analyse_result_covered_unreachable:
  fixes S0 :: "store set" and c₀ :: 'c
  assumes entry_cov: "(cfg_entry g, c₀) ∈ vars"
    and init_sound: "S0 ⊆ γ_DG d₀ e₀"
    and init_env_le: "e₀ ≤ genv global_of sigma"
    and dead: "lookup_coverage analyse_result v ctx = Covered Bot"
  shows "A_{G,adm,c₀,g,S0} v ctx = {}"
  by (rule reported_covered_unreachable_empty
      [OF dead analyse_result_node_sound[OF entry_cov init_sound init_env_le]])

lemma covered_keys_analyse_result [simp]: "covered_keys analyse_result = vars"
  unfolding analyse_result_def by simp
```

The node-level statement, one activation bucket at a time. A covered context is `Bot` by the node predicate; an uncovered one reads `Bot` from `lookup_table` anyway, and here that is not a shrug — this locale's own soundness already puts the collect inside an empty concretization there. So every bucket is empty and so is their union.

`node_collect` decomposes into those buckets only for a context relation that comes from a function, which this locale does not assume of `adm`; the decomposition is therefore a hypothesis, discharged by `node_collect_eq_Union_activation_of_` at every instance whose relation has that shape.

```
lemma analyse_result_node_unreachable:
  fixes S0 :: "store set" and c₀ :: 'c
  assumes entry_cov: "(cfg_entry g, c₀) ∈ vars"
    and init_sound: "S0 ⊆ γ_DG d₀ e₀"
    and init_env_le: "e₀ ≤ genv global_of sigma"
    and union: "C_{G,g,S0} v
                = (⋃ctx. A_{G,adm,c₀,g,S0} v ctx)"
    and dead: "result_node_is_bottom analyse_result v"
  shows "C_{G,g,S0} v = {}"
proof -
  have bucket: "A_{G,adm,c₀,g,S0} v ctx = {}" for ctx
  proof -
    have bot: "lookup_table analyse_result v ctx = Bot"
    proof (cases "(v, ctx) ∈ vars")
      case True
      have "lookup_coverage analyse_result v ctx = Covered Bot"
        using True by (intro result_node_is_bottomD[OF dead]) simp
      then show ?thesis unfolding lookup_table_eq_or_bot by simp
    next
      case False
      then show ?thesis unfolding lookup_table_analyse_result by simp
    qed
    have "A_{G,adm,c₀,g,S0} v ctx
          ⊆ gamma_lift γ_V (lookup_table analyse_result v ctx)"
      by (rule analyse_result_node_sound[OF entry_cov init_sound init_env_le])
    then show ?thesis unfolding bot by simp
  qed
  show ?thesis unfolding union using bucket by simp
```

```
qed
```

The context-indexed check report

The report the CLI prints, defined at the context-indexed level rather than after the join: one verdict per check, aggregated over exactly the contexts the solved table covers at that node. Aggregating first would classify against a joined state and lose the precision each context carries.

```
definition analyse_report_ctx :: "(pp × exp × contextual_verdict) list" where
  "analyse_report_ctx = classify_checks_verdicts g analyse_result classify"
```

Report soundness

`S0` and c_0 are the two facts genuinely external to this locale: which concrete stores the analysis actually starts from, and which context the solved system covers the entry point under. Every existing concrete instance (e.g. Interval's entry-state and call-string contexts) discharges both as per-instance facts about its own solved system, not as locale theorems, since they depend on which keys a terminated solve actually reached — `entry_cov` mirrors those instances' own `entry_cov` context assumption, and `init_sound` is the initial-store premise every activation-trace-level soundness theorem in this development already takes.

Both endpoints below reach their classifier obligation the same way, and that route is this lemma: a collected store at a decided check sits inside some reachable state the report classified, and the report's verdict is what the classifier said about that very state. Only the appeal to `classify_proved` or `classify_refuted` afterwards differs.

```
lemma analyse_report_ctx_decided:
  fixes S0 :: "store set" and c₀ :: 'c and v :: cfg_node and c :: exp
  assumes mem: "(v, c, Decided r) ∈ set analyse_report_ctx"
    and entry_cov: "(cfg_entry g, c₀) ∈ vars"
    and init_sound: "S0 ⊆ γ_DG d₀ e₀"
    and init_env_le: "e₀ ≤ genv global_of sigma"
    and smem: "s ∈ 𝒜_𝒢,adm,c₀,g,S0 v ctx"
    and known: "r ≠ Check_Unknown"
  obtains st where "s ∈ γ_V st" and "classify c st = r"
proof -
  have node_sound: "𝒜_𝒢,adm,c₀,g,S0 v ctx
     ⊆ gamma_lift γ_V (lookup_table analyse_result v ctx)"
    by (rule analyse_result_node_sound[OF entry_cov init_sound init_env_le])
  obtain st where reach: "lookup_table analyse_result v ctx = Lifted st"
    and sst: "s ∈ γ_V st"
    using smem node_sound by (cases "lookup_table analyse_result v ctx") auto
  have "classify c st = r"
    using classify_checks_ctx_decided_sound[OF finE
           mem[unfolded analyse_report_ctx_def] reach known] .
  with sst show ?thesis by (rule that)
qed

theorem analyse_report_ctx_proved_sound:
  fixes S0 :: "store set" and c₀ :: 'c and v :: cfg_node and c :: exp
  assumes mem: "(v, c, Decided Check_Proved) ∈ set analyse_report_ctx"
    and entry_cov: "(cfg_entry g, c₀) ∈ vars"
    and init_sound: "S0 ⊆ γ_DG d₀ e₀"
    and init_env_le: "e₀ ≤ genv global_of sigma"
  shows "⋀ctx s. s ∈ 𝒜_𝒢,adm,c₀,g,S0 v ctx
          ⟹ truthy ([[c]]_e s)"
proof -
  fix ctx s
  assume smem: "s ∈ 𝒜_𝒢,adm,c₀,g,S0 v ctx"
  obtain st where sst: "s ∈ γ_V st" and "classify c st = Check_Proved"
    by (rule analyse_report_ctx_decided[OF mem entry_cov init_sound init_env_le smem]) simp
  thus "truthy ([[c]]_e s)" using classify_proved[OF _ sst] by blast
qed

theorem analyse_report_ctx_refuted_sound:
  fixes S0 :: "store set" and c₀ :: 'c and v :: cfg_node and c :: exp
  assumes mem: "(v, c, Decided Check_Refuted) ∈ set analyse_report_ctx"
    and entry_cov: "(cfg_entry g, c₀) ∈ vars"
    and init_sound: "S0 ⊆ γ_DG d₀ e₀"
    and init_env_le: "e₀ ≤ genv global_of sigma"
  shows "⋀ctx s. s ∈ 𝒜_𝒢,adm,c₀,g,S0 v ctx
          ⟹ ¬ truthy ([[c]]_e s)"
proof -
  fix ctx s
```

```
  assume smem: "s ∈ 𝒜_{𝒢,adm,c₀,g,S0} v ctx"
  obtain st where sst: "s ∈ γ_V st" and "classify c st = Check_Refuted"
    by (rule analyse_report_ctx_decided[OF mem entry_cov init_sound init_env_le smem]) simp
  thus "¬ truthy (⟦c⟧_e s)" using classify_refuted[OF _ sst] by blast
qed

end

end
```

6.32 Routed_Analysis_Sound

```
theory Routed_Analysis_Sound
  imports DG_Analysis_Adapter Routed_Context_Unit
begin
```

Reading a solved routed system as an analysis result

A routed analysis is assembled from two independent choices: which abstract domain it computes in, and which context policy it routes calls by. This theory is the composition step that does not depend on either. It takes a solved routed equation system and derives the analysis-level soundness statement, so a concrete analysis supplies its domain facts and its policy facts and interprets this once, rather than repeating the derivation per (domain, policy) pair.

`solved_local_reader` is the one reader every instance uses: at a covered unknown it hands back the solution's local half, and elsewhere `bot`. Reading a global key gives `bot` too – the routed seed keys carry entry contributions, not program-point values, and a result table never reads them.

```
definition solved_local_reader ::
  "(pp × 'c) set ⇒ (pp × 'c + 'k ⇒ ('D::bounded_semilattice_sup_bot, 'G) dg_state)
   ⇒ pp × 'c + 'k ⇒ 'D"
where
  "solved_local_reader vars sigma k =
     (case k of Inl vc ⇒ (if vc ∈ vars then dg_local (sigma (Inl vc)) else bot)
              | Inr _ ⇒ bot)"

lemma solved_local_reader_covered [simp]:
  "vc ∈ vars ⟹ solved_local_reader vars sigma (Inl vc) = dg_local (sigma (Inl vc))"
  by (simp add: solved_local_reader_def)

lemma solved_local_reader_uncovered [simp]:
  "vc ∉ vars ⟹ solved_local_reader vars sigma (Inl vc) = bot"
  by (simp add: solved_local_reader_def)

lemma solved_local_reader_global [simp]:
  "solved_local_reader vars sigma (Inr k) = bot"
  by (simp add: solved_local_reader_def)
```

The reader returns only the local half. The global half enters through the concretization the locale below pairs with it, which publishes each local value together with the solved global at `Inr buffer_key`. Under that pairing the two coverage obligations `dg_context_activation` asks for hold by construction, given `gammaDG_rd` and that the publication map takes `bot` to `Bot`. Neither depends on the domain or the context policy, so no instance need prove them again.

The composition locale

Everything a routed analysis needs above its solved system, in one place: the domain enters through `S` and γ_{DG}, the context policy through `route`, `adm` and `seed`, and the solved system through `sigma`/`vars`. The fixed reader is `solved_local_reader`, and its concretization reads the solved global at `Inr buffer_key` as the second argument of `rd`, so the two coverage obligations are the one-line lemmas above.

An instance is then a single **interpretation**, and the theorems below are what it gets: a published result table, its per-node soundness, and the check report's proved/refuted verdicts.

```
locale routed_analysis =
  dg_analysis_adapter S γ_DG 𝒢 g buffer_key global_of route bot0 d₀ g₀ sigma vars x0
    "solved_local_reader vars sigma" seed is_bot
    "λd. gamma_lift γ_V (rd d (genv global_of sigma))"
    adm rd γ_V empty_V classify
  for S :: "(pp × 'c, 'k, 'n, 'D::bounded_semilattice_sup_bot,
              'G::bounded_semilattice_sup_bot) dg_spec"
    and γ_DG :: "'D ⇒ ('n ⇒ 'G) ⇒ store set"
    and 𝒢 :: "vname ⇒ bool"
    and g buffer_key and global_of :: "'n ⇒ 'k"
    and route :: "pp ⇒ 'c ⇒ 'D ⇒ call_action ⇒ 'c"
    and bot0 d₀ :: 'D and g₀ :: 'G
    and sigma :: "pp × 'c + 'k ⇒ ('D, 'G) dg_state"
    and vars :: "(pp × 'c) set"
    and x0 :: "pp × 'c"
    and seed :: "pp ⇒ 'c ⇒ 'k"
    and is_bot :: "'D ⇒ bool"
    and adm :: "'c context_policy"
    and rd :: "'D ⇒ ('n ⇒ 'G) ⇒ 'v lifted"
```

```
    and γV :: "'v ⇒ store set"
    and emptyV :: "'v ⇒ bool"
    and classify :: "exp ⇒ 'v ⇒ check_result"

end
```

Chapter 7

Voblint_Exec

7.1 Default_St_Base

```
theory Default_St_Base
  imports "Voblint_VIMP.VIMP_Syntax" "HOL-Library.AList"
begin
```

Finite default-map representation

An abstract state has to be a finite thing before a solver can run on it. A `default_dict` is a finite dictionary that answers every name it does not list with a default. A `default_st_rep` is two of them, one per partition: a local dictionary and a global one. Reading a location consults the dictionary of its partition. Nothing forces a stored value to differ from its default.

Two such pairs can describe the same reading, so the type is quotiented by agreement of all lookups. That identifies list order and redundant duplicate entries only where they do not affect the reading: `map_of` answers with the first match, so reordering entries that disagree on a key is a genuine change and the quotient keeps it.

This theory settles the representation, that quotient, and the order, equality and point update defined on it. It does not decide which variable names are local and which are global – a `location` records a classification this theory never makes. That classifier arrives only in `Default_St_Transfer`, which is built on this.

Why two defaults, rather than a sparse map over a fixed `top`. Nipkow's `Abs_State` stores only the variables it has heard of and reads every other one as `top`. That is the smaller representation, and it makes the order and the join shorter to state, so the extra pair of defaults here needs a reason. Three, in fact, and each is load-bearing on its own.

- **C zero-initialization.** Every domain's entry state gives globals a non-`top` value and locals `top` — ⟪`(STop, []), (SZero, [])`⟫ for Sign, and the parity, interval and product analyses match it. It over-approximates `cinit_stores` $\mathcal{G}$ `= {s. ∀x. `$\mathcal{G}$ `x ⟶ s x = 0}`, which quantifies over *all* names the classifier calls global, for an arbitrary classifier and with no finiteness hypothesis. A fixed-`top` map can only express that by materializing every global, which makes the entry state a function of the program's declaration list and pushes that dependency into every soundness statement that mentions it.
- **The ownership split needs `bot` on the discarded side.** Publishing a state's global half to a shared unknown sends locals to `bot`, and `bot` is what makes that half a unit for the join that combines publications from every call site. `top` there would not be a unit. These half-states are not transient: they are stored in solver unknowns and compared against the solution.
- **The solver needs a least element on this type.** The ownership-split specification types both its local and global unknowns at this carrier and demands `bounded_semilattice_sup_bot` of it, so unknowns can start at `bot`. That element is the everywhere-`bot` state, which a sparse map over a fixed `top` cannot represent at all.

  What the defaults cost is visible throughout this theory: the order has to compare them, extensional equality has to pin them, and the emptiness test in `Default_St_Reachability` has to reason about them. That cost is the price of the three points above, not an accident of the first representation tried.

Default dictionaries

```
type_synonym 'a default_dict = "'a × (vname × 'a) list"

fun default_dict_get :: "'a default_dict => vname => 'a" where
  "default_dict_get (d, ps) x = (case map_of ps x of Some a => a | None => d)"

fun default_dict_set :: "'a default_dict => vname => 'a => 'a default_dict" where
  "default_dict_set (d, ps) x a = (d, (x, a) # AList.delete x ps)"

lemma map_of_resolved_delete:
  "map_of (AList.delete k ps) k' =
     (if k = k' then None else map_of ps k')"
  by (simp add: AList.delete_conv')

lemma map_of_resolved_none_iff:
  "map_of ps k = None ⟷ k ∉ set (map fst ps)"
  by (induction ps) auto

lemma default_dict_get_set [simp]:
  "default_dict_get (default_dict_set m x a) y =
     (if x = y then a else default_dict_get m y)"
```

```
  by (cases m) (simp add: map_of_resolved_delete)
```

A name escaping a finite list, drawn from any infinite supply. It is not listed, so a dictionary answers it with its default. Consumers instantiate the supply with all vnames, or with the ones a classifier leaves local.

```
lemma obtain_fresh_vname:
  assumes "infinite A"
  obtains x where "x ∈ A" and "x ∉ set xs"
proof -
  have "infinite (A - set xs)"
    by (rule Diff_infinite_finite[OF _ assms]) simp
  then obtain x where "x ∈ A - set xs"
    using infinite_imp_nonempty by blast
  then have "x ∈ A" "x ∉ set xs" by simp_all
  then show thesis by (rule that)
qed
```

The order on one dictionary compares the defaults and the finitely many listed names. An unlisted name reads the default on both sides, so this decides the pointwise order over all names.

```
fun le_default_dict :: "('a::order) default_dict => 'a default_dict => bool" where
  "le_default_dict (d, ps) (e, qs) ⟷
     d ≤ e ∧
     list_all (λx. default_dict_get (d, ps) x ≤ default_dict_get (e, qs) x)
       (map fst ps @ map fst qs)"

lemma le_default_dict_iff:
  "le_default_dict m n ⟷ (∀x. default_dict_get m x ≤ default_dict_get n x)"
proof -
  obtain d ps where m: "m = (d, ps)" by (cases m)
  obtain e qs where n: "n = (e, qs)" by (cases n)
  show ?thesis
  proof
    assume le: "le_default_dict m n"
    show "∀x. default_dict_get m x ≤ default_dict_get n x"
    proof
      fix x
      show "default_dict_get m x ≤ default_dict_get n x"
      proof (cases "x ∈ set (map fst ps @ map fst qs)")
        case True
        with le show ?thesis unfolding m n
          by (auto simp: list_all_iff simp del: default_dict_get.simps)
      next
        case False
        then have "map_of ps x = None" "map_of qs x = None"
          by (simp_all add: map_of_resolved_none_iff)
        with le show ?thesis unfolding m n by simp
      qed
    qed
  next
    assume le: "∀x. default_dict_get m x ≤ default_dict_get n x"
    obtain y :: vname where "y ∈ UNIV" and fresh: "y ∉ set (map fst ps @ map fst qs)"
      by (rule obtain_fresh_vname[OF infinite_literal])
    then have "map_of ps y = None" "map_of qs y = None"
      by (simp_all add: map_of_resolved_none_iff)
    with le[rule_format, of y] have "d ≤ e" unfolding m n by simp
    with le show "le_default_dict m n" unfolding m n by (simp add: list_all_iff)
  qed
qed
```

Locations and raw lookup

A `location` names a local or a global variable, and a raw state is a pair of dictionaries read by `default_st_rep_get`.

```
datatype location =
  Local_Location (location_vname: vname)
| Global_Location (location_vname: vname)

lemma all_location_iff:
  "(∀loc. P loc) ⟷ (∀x. P (Local_Location x)) ∧ (∀x. P (Global_Location x))"
  by (metis location.exhaust)
```

The local dictionary comes first, the global one second.

```
type_synonym 'a default_st_rep = "'a default_dict × 'a default_dict"

fun default_st_rep_get ::
```

```
  "('a::bot) default_st_rep => location => 'a" where
  "default_st_rep_get (l, g) (Local_Location x) = default_dict_get l x"
| "default_st_rep_get (l, g) (Global_Location x) = default_dict_get g x"
```

Extensional equality

Two raw states are equal when they agree on every location's lookup, whatever their dictionaries look like; `eq_default_st_rep` is an equivalence.

```
definition eq_default_st_rep ::
  "('a::bot) default_st_rep => 'a default_st_rep => bool"
where
  "eq_default_st_rep s t ⟷
     default_st_rep_get s = default_st_rep_get t"

lemma equivp_eq_default_st_rep: "equivp eq_default_st_rep"
  unfolding eq_default_st_rep_def
  by (rule equivpI) (auto intro: reflpI sympI transpI)
```

The two rules every congruence proof below goes through: a raw operation respects `eq_default_st_rep` exactly when its lookup equation says the result depends on the arguments only through their lookups. Stating that once keeps the individual congruence proofs down to their own lookup rules.

```
lemma eq_default_st_repI [intro]:
  assumes "⋀loc. default_st_rep_get s loc = default_st_rep_get t loc"
  shows "eq_default_st_rep s t"
  unfolding eq_default_st_rep_def fun_eq_iff using assms by blast

lemma eq_default_st_repD:
  assumes "eq_default_st_rep s t"
  shows "default_st_rep_get s loc = default_st_rep_get t loc"
  using assms unfolding eq_default_st_rep_def fun_eq_iff by blast
```

Executable pointwise order

`le_default_st_rep_code` compares both dictionaries with `le_default_dict` and is shown equal to the pointwise order on lookups.

```
fun le_default_st_rep_code ::
  "('a::order_bot) default_st_rep => 'a default_st_rep => bool"
where
  "le_default_st_rep_code (l1, g1) (l2, g2) ⟷
     le_default_dict l1 l2 ∧ le_default_dict g1 g2"
```

The characterization avoids the pair pattern, so that quotient-level statements can be discharged by `transfer` alone instead of re-opening both representatives.

```
lemma le_default_st_rep_code_iff:
  "le_default_st_rep_code s t ⟷
    (∀loc. default_st_rep_get s loc ≤ default_st_rep_get t loc)"
  by (cases s; cases t) (simp add: all_location_iff le_default_dict_iff)
```

The extensional quotient, lookup and point update

`default_st` is the quotient of raw states by extensional equality. Lookup and point update lift to it because both respect the equivalence.

```
quotient_type 'a default_st =
  "('a::bot) default_st_rep" / "eq_default_st_rep"
  morphisms rep_default_st Abs_default_st
  by (rule equivp_eq_default_st_rep)

lift_definition default_st_get ::
  "('a::bot) default_st => location => 'a"
  is default_st_rep_get
  by (simp add: eq_default_st_rep_def)

fun default_st_rep_set ::
  "('a::bot) default_st_rep => location => 'a => 'a default_st_rep" where
  "default_st_rep_set (l, g) (Local_Location x) a = (default_dict_set l x a, g)"
| "default_st_rep_set (l, g) (Global_Location x) a = (l, default_dict_set g x a)"

lemma default_st_rep_get_set [simp]:
  "default_st_rep_get (default_st_rep_set s loc a) loc' =
     (if loc = loc' then a else default_st_rep_get s loc')"
```

```
  by (cases s; cases loc; cases loc') simp_all

lemma eq_default_st_rep_set:
  assumes "eq_default_st_rep s t"
  shows "eq_default_st_rep (default_st_rep_set s loc a)
      (default_st_rep_set t loc a)"
  by (rule eq_default_st_repI) (simp add: eq_default_st_repD[OF assms])

lift_definition default_st_set ::
  "('a::bot) default_st => location => 'a => 'a default_st"
  is default_st_rep_set
  by (rule eq_default_st_rep_set)
```

Lookup is written `s⟨l⟩` and point update `s⟨l := a⟩`. The notation is a bundle: every theory that uses it opens it after `begin` and closes it with `unbundle no` before `end`, so it never reaches an importing theory and never changes a term. Both forms bind tighter than application, so `f s⟨l⟩` reads `f (s⟨l⟩)` and updates chain as `s⟨l := a⟩⟨l'⟩`. The state with local dictionary `(dl, ls)` and global dictionary `(dg, gs)` is written `⦇(dl, ls), (dg, gs)⦈`. Theories past `default_st_to_fun` open `default_st_syntax`, which adds the notation for the function a state represents to this bundle.

```
abbreviation default_st_mk ::
  "('a::bot) default_dict => 'a default_dict => 'a default_st" where
  "default_st_mk l g ≡ Abs_default_st (l, g)"

bundle default_st_carrier_syntax
begin
notation default_st_get ("_⟨_⟩" [1000, 0] 1000)
notation default_st_set ("_⟨_ :=/ _⟩" [1000, 0, 0] 1000)
notation default_st_mk ("⦇_,/ _⦈")
end

unbundle default_st_carrier_syntax

lemma default_st_get_Abs [simp]:
  "(Abs_default_st s)⟨loc⟩ = default_st_rep_get s loc"
  by transfer simp

lemma Abs_default_st_rep_default_st [simp]:
  "Abs_default_st (rep_default_st s) = s"
  by (fact Lifting.Quotient_abs_rep [OF Quotient_default_st])

lemma default_st_get_rep:
  "s⟨loc⟩ = default_st_rep_get (rep_default_st s) loc"
  by (simp add: default_st_get.rep_eq)
```

One lookup equation for the update, rather than a same/different pair: the conditional is what a caller has to reason about anyway, and the two special cases fall out of it by `simp`.

```
lemma default_st_get_set [simp]:
  "s⟨loc := a⟩⟨loc'⟩ = (if loc = loc' then a else s⟨loc'⟩)"
  by transfer simp

lemma default_st_eq_iff:
  "s = t ⟷
     default_st_get s = default_st_get t"
  by transfer (simp add: eq_default_st_rep_def)
```

The quotient-level counterpart of `eq_default_st_repI`: two quotient states are equal as soon as they look the same everywhere. Left untagged – as an `[intro]` rule it would attack every equation at this type by extensionality, which is rarely what a goal about a concrete state wants.

```
lemma default_st_eqI:
  assumes "⋀loc. s⟨loc⟩ = t⟨loc⟩"
  shows "s = t"
  using assms by (simp add: default_st_eq_iff fun_eq_iff)
```

Order, bottom and executable equality

Bottom is the pair of empty dictionaries with bottom defaults; the order lifts `le_default_st_rep_code`, which also yields an executable equality.

```
instantiation default_st :: (bot) bot
begin
definition bot_default_st ::
  "('a::bot) default_st"
```

```
where
  "bot_default_st = ⦇(bot, []), (bot, [])⦈"
instance ..
end

instantiation default_st :: (order_bot) ord
begin
lift_definition less_eq_default_st ::
  "('a::order_bot) default_st =>
   'a default_st => bool"
  is le_default_st_rep_code
  by (auto simp: le_default_st_rep_code_iff eq_default_st_rep_def)

definition less_default_st ::
  "('a::order_bot) default_st =>
   'a default_st => bool"
where
  "less_default_st s t ⟷
     s ≤ t ∧ ¬ t ≤ s"

instance ..
end


lemma le_default_st_iff:
  fixes s t :: "('a::order_bot) default_st"
  shows "s ≤ t ⟷ (∀loc. s⟨loc⟩ ≤ t⟨loc⟩)"
  by transfer (rule le_default_st_rep_code_iff)


instance default_st :: (order_bot) order
proof intro_classes
  fix s t u :: "('a::order_bot) default_st"
  show "(s < t) ⟷ (s ≤ t ∧ ¬ t ≤ s)"
    by (simp add: less_default_st_def)

  show "s ≤ s"
    by (simp add: le_default_st_iff)
  show "s ≤ t ⟹ t ≤ u ⟹ s ≤ u"
    by (auto simp: le_default_st_iff intro: order_trans)
  show "s ≤ t ⟹ t ≤ s ⟹ s = t"
    by (auto simp: le_default_st_iff
      default_st_eq_iff fun_eq_iff intro: order_antisym)
qed

context
  includes lattice_syntax
begin

lemma default_st_get_bot [simp]:
  "(⊥ :: ('a::bot) default_st)⟨loc⟩ = ⊥"
  unfolding bot_default_st_def by (cases loc) simp_all

end


lemma bot_le_default_st:
  "(bot :: ('a::order_bot) default_st) ≤ s"
  by (simp add: le_default_st_iff)

instance default_st :: (order_bot) order_bot
  by standard (rule bot_le_default_st)
instantiation default_st :: ("{order_bot,equal}") equal
begin
definition equal_default_st ::
  "('a::{order_bot,equal}) default_st =>
   'a default_st => bool"
where
  "equal_default_st s t = (s ≤ t ∧ t ≤ s)"
instance
  by standard (auto simp: equal_default_st_def intro: order_antisym)
end

unbundle no default_st_carrier_syntax

end
```

7.2 Default_St_Algebra

```
theory Default_St_Algebra
  imports Default_St_Base "Voblint_Domain.State_Concretization"
begin

unbundle default_st_carrier_syntax
```

Pointwise abstract-domain operations

Join, widening and narrowing are the same construction three times: apply an element operation to both dictionaries of a state, at the default and at every name either argument lists. This theory gives that construction once, instantiates it three times, and installs the lattice and warrowing classes the solver needs.

The combinator's support is the two argument supports with duplicate keys removed. Keeping both copies would denote the same lookup function, but every join would concatenate, and a solver iteration would then grow the materialized list without bound while each lookup scans it.

The pointwise combinator

```
fun map2_default_dict ::
  "('a => 'a => 'a) => 'a default_dict => 'a default_dict => 'a default_dict"
where
  "map2_default_dict f (d1, ps1) (d2, ps2) =
     (f d1 d2,
      map (λx. (x, f (default_dict_get (d1, ps1) x) (default_dict_get (d2, ps2) x)))
        (remdups (map fst ps1 @ map fst ps2)))"

fun map2_default_st_rep ::
  "('a => 'a => 'a) => ('a::bot) default_st_rep => 'a default_st_rep =>
   'a default_st_rep"
where
  "map2_default_st_rep f (l1, g1) (l2, g2) =
     (map2_default_dict f l1 l2, map2_default_dict f g1 g2)"

lemma map_of_map_key_list:
  "map_of (map (λk. (k, g k)) ks) x =
     (if x ∈ set ks then Some (g x) else None)"
  by (induction ks) auto

lemma default_dict_get_map2 [simp]:
  "default_dict_get (map2_default_dict f m1 m2) x =
     f (default_dict_get m1 x) (default_dict_get m2 x)"
  by (cases m1; cases m2)
    (auto simp: map_of_map_key_list map_of_resolved_none_iff[THEN iffD2])
```

From here on the combinator is read through its lookup equation. Its defining equation would otherwise unfold first and hide it from `default_dict_get_map2`.

```
declare map2_default_dict.simps [simp del]

lemma default_st_rep_get_map2 [simp]:
  "default_st_rep_get (map2_default_st_rep f s t) loc =
     f (default_st_rep_get s loc) (default_st_rep_get t loc)"
  by (cases s; cases t; cases loc) simp_all

lemma eq_default_st_rep_map2:
  assumes "eq_default_st_rep s1 s2"
    and "eq_default_st_rep t1 t2"
  shows "eq_default_st_rep (map2_default_st_rep f s1 t1)
      (map2_default_st_rep f s2 t2)"
  by (rule eq_default_st_repI)
     (simp add: eq_default_st_repD[OF assms(1)] eq_default_st_repD[OF assms(2)])
```

Join and semilattice structure

Join is the pointwise join of both dictionaries, which respects extensional equality and so lifts to `default_st` as a bounded join-semilattice.

```
definition merge_default_st_rep ::
  "('a::bounded_semilattice_sup_bot) default_st_rep =>
   'a default_st_rep => 'a default_st_rep"
where
  "merge_default_st_rep s t = map2_default_st_rep (⊔) s t"
```

```
lemma default_st_rep_get_merge [simp]:
  "default_st_rep_get (merge_default_st_rep s t) loc =
     default_st_rep_get s loc ⊔ default_st_rep_get t loc"
  by (simp add: merge_default_st_rep_def)

lemma eq_default_st_rep_merge:
  assumes "eq_default_st_rep s1 s2"
    and "eq_default_st_rep t1 t2"
  shows "eq_default_st_rep (merge_default_st_rep s1 t1)
      (merge_default_st_rep s2 t2)"
  using assms unfolding merge_default_st_rep_def
  by (rule eq_default_st_rep_map2)

instantiation default_st ::
  (bounded_semilattice_sup_bot) sup
begin
lift_definition sup_default_st ::
  "('a::bounded_semilattice_sup_bot) default_st =>
   'a default_st => 'a default_st"
  is merge_default_st_rep
  by (rule eq_default_st_rep_merge)
instance ..
end

lemma default_st_get_sup [simp]:
  "(s ⊔ t)⟨loc⟩ = s⟨loc⟩ ⊔ t⟨loc⟩"
  by transfer (rule default_st_rep_get_merge)

instance default_st ::
  (bounded_semilattice_sup_bot) semilattice_sup
proof intro_classes
  fix s t u :: "('a::bounded_semilattice_sup_bot) default_st"
  show "s ≤ s ⊔ t"
    by (simp add: le_default_st_iff)
  show "t ≤ s ⊔ t"
    by (simp add: le_default_st_iff)
  show "s ≤ u ⟹ t ≤ u ⟹ s ⊔ t ≤ u"
    by (simp add: le_default_st_iff)
qed

instance default_st ::
  (bounded_semilattice_sup_bot) bounded_semilattice_sup_bot ..
```

Widening and narrowing

Widening and narrowing act pointwise as well, inheriting the element domain's `warrowing` laws location by location.

```
definition widen_default_st_rep ::
  "('a::{bounded_semilattice_sup_bot, warrowing}) default_st_rep =>
   'a default_st_rep => 'a default_st_rep"
where
  "widen_default_st_rep s t = map2_default_st_rep (∇) s t"

lemma default_st_rep_get_widen [simp]:
  "default_st_rep_get (widen_default_st_rep s t) loc =
     default_st_rep_get s loc ∇ default_st_rep_get t loc"
  by (simp add: widen_default_st_rep_def)

lemma eq_default_st_rep_widen:
  assumes "eq_default_st_rep s1 s2"
    and "eq_default_st_rep t1 t2"
  shows "eq_default_st_rep (widen_default_st_rep s1 t1)
      (widen_default_st_rep s2 t2)"
  using assms unfolding widen_default_st_rep_def
  by (rule eq_default_st_rep_map2)
```

Widening and narrowing are lifted to a named constant first and only then installed as the class operation, unlike (⊔) which is lifted inside its own `instantiation`. The detour is forced: the `widening` and `narrowing` classes carry axioms, so their `instance` proofs need the lookup equation, and inside an `instantiation` block the class operation is not yet the same term as a constant lifted there – a lookup lemma stated on the latter fails to refine a goal about the former, and one proved before `instance` abstracts over the operation instead of fixing it. (⊔) escapes this because the bare `sup` class has no axioms to discharge.

```
lift_definition widen_on_default_st ::
```

```
  "('a::{bounded_semilattice_sup_bot, warrowing}) default_st =>
   'a default_st => 'a default_st"
  is widen_default_st_rep
  by (rule eq_default_st_rep_widen)

lemma default_st_get_widen_on [simp]:
  "(widen_on_default_st s t)⟨loc⟩ =
     s⟨loc⟩ ▽ t⟨loc⟩"
  by transfer (rule default_st_rep_get_widen)

instantiation default_st :: ("{bounded_semilattice_sup_bot, warrowing}") widening
begin
definition widen_default_st ::
  "('a::{bounded_semilattice_sup_bot, warrowing}) default_st =>
   'a default_st => 'a default_st"
where
  "widen_default_st s t = widen_on_default_st s t"
instance
proof
  fix a b :: "('a::{bounded_semilattice_sup_bot, warrowing}) default_st"
  show "a ≤ (a ▽ b)"
    by (simp add: le_default_st_iff widen_default_st_def widen_ge1)
  show "b ≤ (a ▽ b)"
    by (simp add: le_default_st_iff widen_default_st_def widen_ge2)
qed
end

lemma default_st_get_widen [simp]:
  "(s ▽ t)⟨loc⟩ = s⟨loc⟩ ▽ t⟨loc⟩"
  by (simp add: widen_default_st_def)

definition narrow_default_st_rep ::
  "('a::{bounded_semilattice_sup_bot, warrowing}) default_st_rep =>
   'a default_st_rep => 'a default_st_rep"
where
  "narrow_default_st_rep s t = map2_default_st_rep (△) s t"

lemma default_st_rep_get_narrow [simp]:
  "default_st_rep_get (narrow_default_st_rep s t) loc =
     default_st_rep_get s loc △ default_st_rep_get t loc"
  by (simp add: narrow_default_st_rep_def)

lemma eq_default_st_rep_narrow:
  assumes "eq_default_st_rep s1 s2"
    and "eq_default_st_rep t1 t2"
  shows "eq_default_st_rep (narrow_default_st_rep s1 t1)
      (narrow_default_st_rep s2 t2)"
  using assms unfolding narrow_default_st_rep_def
  by (rule eq_default_st_rep_map2)

lift_definition narrow_on_default_st ::
  "('a::{bounded_semilattice_sup_bot, warrowing}) default_st =>
   'a default_st => 'a default_st"
  is narrow_default_st_rep
  by (rule eq_default_st_rep_narrow)

lemma default_st_get_narrow_on [simp]:
  "(narrow_on_default_st s t)⟨loc⟩ =
     s⟨loc⟩ △ t⟨loc⟩"
  by transfer (rule default_st_rep_get_narrow)

instantiation default_st :: ("{bounded_semilattice_sup_bot, warrowing}") narrowing
begin
definition narrow_default_st ::
  "('a::{bounded_semilattice_sup_bot, warrowing}) default_st =>
   'a default_st => 'a default_st"
where
  "narrow_default_st s t = narrow_on_default_st s t"
instance
  by standard
     (auto simp add: le_default_st_iff narrow_default_st_def
       intro: narrow_ge narrow_le)
end

lemma default_st_get_narrow [simp]:
  "(s △ t)⟨loc⟩ = s⟨loc⟩ △ t⟨loc⟩"
```

```
  by (simp add: narrow_default_st_def)

instance default_st :: ("{bounded_semilattice_sup_bot, warrowing}") warrowing ..

unbundle no default_st_carrier_syntax

end
```

7.3 Default_St_Transfer

```
theory Default_St_Transfer
  imports Default_St_Algebra "Voblint_Framework.Transfer_Algebra"
begin

unbundle default_st_carrier_syntax
```

Refinement to variable-indexed states

Everything so far is stated about locations. A program, though, talks about variable names, and which names are global is a property of the program, not of the carrier. `location_of` is that classifier applied: it turns a name into the location holding its value, and `default_st_rep_to_fun` maps a whole state to the function `'a abs_state` it represents, the form soundness is stated over.

The rest of the theory is the operations a call and a return need – assignment, formal binding, restriction to one side of the ownership split, frame entry, and the caller/callee combination – each paired with the equation saying that performing it on the carrier and then taking the represented function agrees with performing its counterpart on that function. Those equations are the interface every soundness proof downstream actually uses.

Location classification and projection

```
definition location_of ::
  "(vname => bool) => vname => location" where
  "location_of 𝒢 x =
     (if 𝒢 x then Global_Location x else Local_Location x)"
```

Both tags carry the vname they classify, so a classified location determines its own name and two names collide only when they are equal. Proofs about `location_of` therefore need no case split on `𝒢 x` against `𝒢 y`.

```
lemma location_vname_location_of [simp]:
  "location_vname (location_of 𝒢 x) = x"
  by (simp add: location_of_def)

lemma location_of_eq_iff [simp]:
  "location_of 𝒢 x = location_of 𝒢 y ⟷ x = y"
  by (auto simp: location_of_def)
```

Deliberately no `[simp]` rule matching `location_of 𝒢 x = Local_Location y` against a constructor. Such a rule reads well in isolation, but proofs here derive facts of exactly that shape and then use them as rewrites for `location_of`; a simp rule on the same left-hand side collapses the equation to its classification side and takes the rewrite away.

```
definition default_st_rep_to_fun ::
  "(vname => bool) => ('a::bot) default_st_rep => vname => 'a" where
  "default_st_rep_to_fun 𝒢 s x =
     default_st_rep_get s (location_of 𝒢 x)"

lemma default_st_rep_to_fun_set_location [simp]:
  "default_st_rep_to_fun 𝒢
      (default_st_rep_set s (location_of 𝒢 x) a) =
   (default_st_rep_to_fun 𝒢 s)(x := a)"
proof (rule ext)
  fix y
  show "default_st_rep_to_fun 𝒢
      (default_st_rep_set s (location_of 𝒢 x) a) y =
    ((default_st_rep_to_fun 𝒢 s)(x := a)) y"
    unfolding default_st_rep_to_fun_def    by (cases "x = y"; cases "𝒢 x"; cases "𝒢 y";
       simp_all add: location_of_def)
qed


definition default_st_to_fun ::
  "(vname => bool) => ('a::bot) default_st => vname => 'a"
where
  "default_st_to_fun 𝒢 s x =
     s⟨location_of 𝒢 x⟩"
```

The function a state represents is its `readback`, written $\varrho_{\mathcal{G}}$ `s`; a lifted state reads back pointwise under the lift. The notation joins the carrier notation in the opt-in bundle `default_st_syntax`: vendored theories bind ϱ as a variable, so it must never be global.

```
adhoc_overloading readback == default_st_to_fun
```

```
bundle default_st_syntax
begin
unbundle default_st_carrier_syntax
notation readback ("ϱ_")
end

unbundle default_st_syntax

lemma default_st_to_fun_bot [simp]:
  "ϱ_𝒢 (bot :: ('a::order_bot) default_st) = bot"
  by (rule ext) (simp add: default_st_to_fun_def)
```

The state a run starts in gives every local one value and every global another: C initializes a declared global to zero and leaves a local unconstrained. Each domain's C-initial state is this construction at its own abstraction of zero and its whole-value element, so the equation for its function is one lemma rather than one per domain. The two empty dictionaries are abstracted directly rather than lifted, because a domain whose value type is itself a typedef would otherwise descend through both quotients.

```
definition initial_default_st :: "'a::bot => 'a => 'a default_st" where
  "initial_default_st local_value global_value =
     ⟪(local_value, []), (global_value, [])⟫"

lemma default_st_get_initial [simp]:
  "ϱ_𝒢 (initial_default_st local_value global_value) x =
   (if 𝒢 x then global_value else local_value)"
  unfolding default_st_to_fun_def initial_default_st_def
  by (auto simp: location_of_def split: if_splits)

lemma default_st_to_fun_initial:
  "ϱ_𝒢 (initial_default_st local_value global_value) =
   (λx. if 𝒢 x then global_value else local_value)"
  by (rule ext) simp
```

The quotient-level projection agrees with the raw one at `s`'s own chosen representative – so any raw-level fact about `default_st_rep_to_fun` transports directly to `readback` through `rep_default_st`, with no separate quotient-respectfulness argument needed: `rep_default_st` is a genuine function of `s`, so anything defined through it is automatically well-defined on the quotient.

```
lemma default_st_to_fun_rep:
  "ϱ_𝒢 s = default_st_rep_to_fun 𝒢 (rep_default_st s)"
  unfolding default_st_to_fun_def default_st_rep_to_fun_def
  by (rule ext) (simp add: default_st_get_rep)

lemma default_st_to_fun_mono:
  assumes "s ≤ t"
  shows "ϱ_𝒢 s ≤ ϱ_𝒢 t"
  using assms
  unfolding default_st_to_fun_def le_fun_def
  by (simp add: le_default_st_iff)
```

Each quotient-level refinement equation is its raw counterpart transported along the quotient: unfolding the projection leaves a goal in `default_st_get`, `transfer` lowers it to `default_st_rep_get`, and the raw lemma – read at the same unfolded shape – closes it. An operation returning the quotient has no `rep_eq` to unfold instead, since its representative is fixed only up to `eq_default_st_rep`.

Assignment and formal binding

```
definition combine_assign_default_st_rep ::
  "(vname => bool) => vname option => 'a => ('a::bot) default_st_rep
   => 'a default_st_rep"
where
  "combine_assign_default_st_rep 𝒢 dst v s =
     (case dst of None => s
      | Some x => default_st_rep_set s (location_of 𝒢 x) v)"

lemma eq_default_st_rep_combine_assign:
  assumes "eq_default_st_rep s t"
  shows "eq_default_st_rep (combine_assign_default_st_rep 𝒢 dst v s)
      (combine_assign_default_st_rep 𝒢 dst v t)"
  by (unfold combine_assign_default_st_rep_def; cases dst;
      simp_all add: assms eq_default_st_rep_set)

lift_definition combine_assign_default_st ::
```

```
  "(vname => bool) => vname option => 'a => ('a::bot) default_st
   => 'a default_st"
  is combine_assign_default_st_rep
  by (rule eq_default_st_rep_combine_assign)

lemma default_st_get_combine_assign [simp]:
  "(combine_assign_default_st 𝒢 dst v s)⟨loc⟩ =
     (case dst of
        None => s⟨loc⟩
      | Some x => if location_of 𝒢 x = loc then v else s⟨loc⟩)"
  by transfer (auto simp add:combine_assign_default_st_rep_def split:option.splits)

lemma default_st_rep_to_fun_combine_assign [simp]:
  "default_st_rep_to_fun 𝒢
      (combine_assign_default_st_rep 𝒢 dst v s) =
   combine_assign dst v (default_st_rep_to_fun 𝒢 s)"
  by (cases dst)
     (simp_all add: combine_assign_default_st_rep_def)

lemma default_st_to_fun_combine_assign [simp]:
  "ϱ𝒢
      (combine_assign_default_st 𝒢 dst v s) =
   combine_assign dst v (ϱ𝒢 s)"
  unfolding default_st_to_fun_def
  by transfer
     (rule default_st_rep_to_fun_combine_assign[unfolded default_st_rep_to_fun_def])

definition bind_formals_default_st_rep ::
  "(vname => bool) => vname list => 'a list => ('a::bot) default_st_rep
   => 'a default_st_rep"
where
  "bind_formals_default_st_rep 𝒢 xs avs s =
     fold (λ(x, a) t. default_st_rep_set t (location_of 𝒢 x) a)
       (zip xs avs) s"

lemma eq_default_st_rep_fold_set:
  "eq_default_st_rep s t ⟹
     eq_default_st_rep
       (fold (λ(x, a) t. default_st_rep_set t (location_of 𝒢 x) a) ps s)
       (fold (λ(x, a) t. default_st_rep_set t (location_of 𝒢 x) a) ps t)"
proof (induction ps arbitrary: s t)
  case Nil
  then show ?case by simp
next
  case (Cons p ps)
  then show ?case
    by (cases p) (simp_all add: eq_default_st_rep_set)
qed

lemma eq_default_st_rep_bind_formals:
  assumes "eq_default_st_rep s t"
  shows "eq_default_st_rep (bind_formals_default_st_rep 𝒢 xs avs s)
      (bind_formals_default_st_rep 𝒢 xs avs t)"
  unfolding bind_formals_default_st_rep_def
  using assms by (rule eq_default_st_rep_fold_set)

lift_definition bind_formals_default_st ::
  "(vname => bool) => vname list => 'a list => ('a::bot) default_st
   => 'a default_st"
  is bind_formals_default_st_rep
  by (rule eq_default_st_rep_bind_formals)
```

A single-formal call binds exactly one location, so its reduction is a plain `default_st_set`. An instance with a one-argument procedure call cites this directly instead of unfolding `bind_formals_default_st`'s fold.

```
lemma bind_formals_default_st_singleton:
  "bind_formals_default_st 𝒢 [x] [a] s = s⟨location_of 𝒢 x := a⟩"
  by transfer (simp add: bind_formals_default_st_rep_def eq_default_st_rep_def)

lemma default_st_rep_to_fun_fold_set:
  "default_st_rep_to_fun 𝒢
      (fold (λ(x, a) t. default_st_rep_set t (location_of 𝒢 x) a) ps s) =
   fold (λ(x, a) t. t(x := a)) ps
      (default_st_rep_to_fun 𝒢 s)"
  by (induction ps arbitrary: s)
     (simp_all split: prod.splits)
```

```
lemma default_st_rep_to_fun_bind_formals [simp]:
  "default_st_rep_to_fun 𝒢
      (bind_formals_default_st_rep 𝒢 xs avs s) =
   bind_formals xs avs (default_st_rep_to_fun 𝒢 s)"
  unfolding bind_formals_default_st_rep_def
  by (rule default_st_rep_to_fun_fold_set)

lemma default_st_to_fun_bind_formals [simp]:
  "ϱ𝒢
      (bind_formals_default_st 𝒢 xs avs s) =
   bind_formals xs avs (ϱ𝒢 s)"
  unfolding default_st_to_fun_def
  by transfer
     (rule default_st_rep_to_fun_bind_formals[unfolded default_st_rep_to_fun_def])

lemma default_st_to_fun_set [simp]:
  "ϱ𝒢 s⟨location_of 𝒢 x := a⟩ =
   (ϱ𝒢 s)(x := a)"
proof (rule ext)
  fix y
  show "ϱ𝒢 s⟨location_of 𝒢 x := a⟩ y =
    ((ϱ𝒢 s)(x := a)) y"
    unfolding default_st_to_fun_def
    by (cases "x = y"; cases "𝒢 x"; cases "𝒢 y";
        simp_all add: location_of_def)
qed
```

Ownership restriction and combination

Each dictionary holds one partition, so the ownership operations below move whole dictionaries: a restriction keeps one and empties the other to `bot`, and the combination takes the caller's local dictionary and the callee's global one.

```
definition restrict_local_default_st_rep ::
  "('a::bot) default_st_rep => 'a default_st_rep" where
  "restrict_local_default_st_rep s = (fst s, (bot, []))"

definition restrict_global_default_st_rep ::
  "('a::bot) default_st_rep => 'a default_st_rep" where
  "restrict_global_default_st_rep s = ((bot, []), snd s)"


lemma default_st_rep_get_restrict_local:
  "default_st_rep_get (restrict_local_default_st_rep s) loc =
     (case loc of
        Local_Location x => default_st_rep_get s loc
      | Global_Location x => bot)"
  by (cases s; cases loc) (simp_all add: restrict_local_default_st_rep_def)

lemma default_st_rep_get_restrict_global:
  "default_st_rep_get (restrict_global_default_st_rep s) loc =
     (case loc of
        Local_Location x => bot
      | Global_Location x => default_st_rep_get s loc)"
  by (cases s; cases loc) (simp_all add: restrict_global_default_st_rep_def)

lemma eq_default_st_rep_restrict_local:
  assumes "eq_default_st_rep s t"
  shows "eq_default_st_rep (restrict_local_default_st_rep s)
      (restrict_local_default_st_rep t)"
  by (rule eq_default_st_repI)
     (simp add: default_st_rep_get_restrict_local eq_default_st_repD[OF assms]
      split: location.split)

lemma eq_default_st_rep_restrict_global:
  assumes "eq_default_st_rep s t"
  shows "eq_default_st_rep (restrict_global_default_st_rep s)
      (restrict_global_default_st_rep t)"
  by (rule eq_default_st_repI)
     (simp add: default_st_rep_get_restrict_global eq_default_st_repD[OF assms]
      split: location.split)

lift_definition restrict_local_default_st ::
  "('a::bot) default_st => 'a default_st"
  is restrict_local_default_st_rep
  by (rule eq_default_st_rep_restrict_local)
```

```
lift_definition restrict_global_default_st ::
  "('a::bot) default_st => 'a default_st"
  is restrict_global_default_st_rep
  by (rule eq_default_st_rep_restrict_global)
```

`restrict_local_default_st`/`restrict_global_default_st` preserve the caller's semantic default over the (potentially infinite) location space: the kept side carries over its input's own dictionary verbatim, whose default need not be $\bot$, and only the dropped side is replaced by the empty dictionary over $\bot$. A scope-parametric projection over a bounded materialized support would disagree with this pair outside its bound, so the pair is stated over the full location space and preserves the default by construction.

```
lemma default_st_get_restrict_local [simp]:
  "(restrict_local_default_st s)⟨loc⟩ =
     (case loc of
        Local_Location x => s⟨loc⟩
      | Global_Location x => bot)"
  by transfer (rule default_st_rep_get_restrict_local)

lemma default_st_get_restrict_global [simp]:
  "(restrict_global_default_st s)⟨loc⟩ =
     (case loc of
        Local_Location x => bot
      | Global_Location x => s⟨loc⟩)"
  by transfer (rule default_st_rep_get_restrict_global)

lemma default_st_to_fun_restrict_local [simp]:
  "ϱ_𝒢 (restrict_local_default_st s) x =
     (if 𝒢 x then bot else ϱ_𝒢 s x)"
  unfolding default_st_to_fun_def location_of_def
  by (cases "𝒢 x") simp_all

lemma default_st_to_fun_restrict_global [simp]:
  "ϱ_𝒢 (restrict_global_default_st s) x =
     (if 𝒢 x then ϱ_𝒢 s x else bot)"
  unfolding default_st_to_fun_def location_of_def
  by (cases "𝒢 x") simp_all

definition combine_default_st_rep ::
  "('a::bot) default_st_rep => 'a default_st_rep => 'a default_st_rep"
where
  "combine_default_st_rep sc se = (fst sc, snd se)"

lemma default_st_rep_get_combine [simp]:
  "default_st_rep_get (combine_default_st_rep sc se) loc =
   (case loc of
      Local_Location x => default_st_rep_get sc loc
    | Global_Location x => default_st_rep_get se loc)"
  by (cases sc; cases se; cases loc) (simp_all add: combine_default_st_rep_def)

lemma eq_default_st_rep_combine:
  assumes "eq_default_st_rep sc1 sc2"
    and "eq_default_st_rep se1 se2"
  shows "eq_default_st_rep (combine_default_st_rep sc1 se1)
      (combine_default_st_rep sc2 se2)"
  by (rule eq_default_st_repI)
     (simp add: eq_default_st_repD[OF assms(1)] eq_default_st_repD[OF assms(2)]
      split: location.split)

lift_definition combine_default_st ::
  "('a::bot) default_st => 'a default_st => 'a default_st"
  is combine_default_st_rep
  by (rule eq_default_st_rep_combine)

lemma default_st_get_combine [simp]:
  "(combine_default_st sc se)⟨loc⟩ =
     (case loc of
        Local_Location x => sc⟨loc⟩
      | Global_Location x => se⟨loc⟩)"
  by transfer (rule default_st_rep_get_combine)


lemma default_st_rep_to_fun_combine [simp]:
  "default_st_rep_to_fun 𝒢 (combine_default_st_rep sc se) =
   combine_env 𝒢 (default_st_rep_to_fun 𝒢 sc)
     (default_st_rep_to_fun 𝒢 se)"
```

```
proof (rule ext)
  fix x
  show "default_st_rep_to_fun 𝒢 (combine_default_st_rep sc se) x =
      combine_env 𝒢 (default_st_rep_to_fun 𝒢 sc)
        (default_st_rep_to_fun 𝒢 se) x"
    unfolding default_st_rep_to_fun_def combine_env_def location_of_def
    by (cases "𝒢 x") simp_all
qed

lemma default_st_to_fun_combine [simp]:
  "ϱ𝒢 (combine_default_st sc se) =
   combine_env 𝒢 (ϱ𝒢 sc)
     (ϱ𝒢 se)"
proof (rule ext)
  fix x
  show "ϱ𝒢 (combine_default_st sc se) x =
      combine_env 𝒢 (ϱ𝒢 sc)
        (ϱ𝒢 se) x"
    unfolding default_st_to_fun_def combine_env_def location_of_def
    by (cases "𝒢 x") simp_all
qed


lemma default_st_to_fun_sup [simp]:
  "ϱ𝒢 (s ⊔ t) =
   ϱ𝒢 s ⊔ ϱ𝒢 t"
  by (rule ext) (simp add: default_st_to_fun_def sup_fun_def)
```

Frame entry

Entering a frame sets every local to `top_val` and keeps the globals; the lemmas carry this through the quotient and the readback.

```
definition enter_frame_D_default_st_rep ::
  "'a => ('a::bot) default_st_rep => 'a default_st_rep"
where
  "enter_frame_D_default_st_rep top_val s = ((top_val, []), snd s)"

lemma default_st_rep_get_enter_frame_D [simp]:
  "default_st_rep_get (enter_frame_D_default_st_rep top_val s) loc =
   (case loc of
      Local_Location x => top_val
    | Global_Location x => default_st_rep_get s loc)"
  by (cases s; cases loc) (simp_all add: enter_frame_D_default_st_rep_def)

lemma eq_default_st_rep_enter_frame_D:
  assumes "eq_default_st_rep s t"
  shows "eq_default_st_rep (enter_frame_D_default_st_rep top_val s)
      (enter_frame_D_default_st_rep top_val t)"
  by (rule eq_default_st_repI)
     (simp add: eq_default_st_repD[OF assms] split: location.split)

lift_definition enter_frame_D_default_st ::
  "'a => ('a::bot) default_st => 'a default_st"
  is enter_frame_D_default_st_rep
  by (rule eq_default_st_rep_enter_frame_D)

lemma default_st_get_enter_frame_D [simp]:
  "(enter_frame_D_default_st top_val s)⟨loc⟩ =
     (case loc of
        Local_Location x => top_val
      | Global_Location x => s⟨loc⟩)"
  by transfer (rule default_st_rep_get_enter_frame_D)

lemma default_st_rep_to_fun_enter_frame [simp]:
  "default_st_rep_to_fun 𝒢 (enter_frame_D_default_st_rep top_val s) =
   enter_frame 𝒢 top_val (default_st_rep_to_fun 𝒢 s)"
proof (rule ext)
  fix x
  show "default_st_rep_to_fun 𝒢 (enter_frame_D_default_st_rep top_val s) x =
      enter_frame 𝒢 top_val (default_st_rep_to_fun 𝒢 s) x"
    unfolding enter_frame_def default_st_rep_to_fun_def location_of_def
    by (cases "𝒢 x") simp_all
qed

lemma default_st_to_fun_enter_frame [simp]:
```

```
  "ϱ𝒢
      (enter_frame_D_default_st top_val s) =
   enter_frame 𝒢 top_val (ϱ𝒢 s)"
  unfolding default_st_to_fun_def
  by transfer
     (rule default_st_rep_to_fun_enter_frame[unfolded default_st_rep_to_fun_def])
```

Executable restriction and combination laws

Algebraic identities of the carrier itself: no abstract state, no concretization, no classifier. They say that combining and then projecting recovers the projected side, and that the two projections recombine by join. Executable trees use them to split and rebuild a routed state.

```
lemma combine_default_st_eq_restrict_sup:
  "combine_default_st A B =
     restrict_local_default_st A ⊔ restrict_global_default_st B"
  by (rule default_st_eqI) (simp split: location.split)

lemma restrict_local_default_st_combine [simp]:
  "restrict_local_default_st (combine_default_st A B) =
     restrict_local_default_st A"
  by (rule default_st_eqI) (simp split: location.split)

lemma restrict_global_default_st_combine [simp]:
  "restrict_global_default_st (combine_default_st A B) =
     restrict_global_default_st B"
  by (rule default_st_eqI) (simp split: location.split)

lemma restrict_local_default_st_split [simp]:
  "restrict_local_default_st (restrict_local_default_st A ⊔
      restrict_global_default_st B) = restrict_local_default_st A"
  by (rule default_st_eqI) (simp split: location.split)

lemma restrict_global_default_st_split [simp]:
  "restrict_global_default_st (restrict_local_default_st A ⊔
      restrict_global_default_st B) = restrict_global_default_st B"
  by (rule default_st_eqI) (simp split: location.split)
```

Combining reads only the local half of its first argument and the global half of its second, so a state recombines from its own two projections.

```
lemma combine_default_st_restrict_split [simp]:
  "combine_default_st (restrict_local_default_st x) (restrict_global_default_st x) = x"
  by (rule default_st_eqI) (simp split: location.split)

lemma combine_default_st_restrict_local_left [simp]:
  "combine_default_st (restrict_local_default_st x) y = combine_default_st x y"
  by (rule default_st_eqI) (simp split: location.split)

lemma combine_default_st_self_restrict_global [simp]:
  "combine_default_st x (restrict_global_default_st x) = x"
  by (rule default_st_eqI) (simp split: location.split)
```

The stores a carrier state describes

A carrier state denotes the stores its function denotes. This is the one concretization the carrier offers to semantic statements; the represented function itself stays a refinement device, used to relate carrier operations to their counterparts on `'a abs_state`. Only the classifier $\mathcal{G}$ is needed besides the state, because it decides which location holds each name's value. A context that fixes $\mathcal{G}$ may register `default_st_gamma 𝒢` under γ `_` with `adhoc_overloading`.

```
definition default_st_gamma ::
  "(vname => bool) => ('a::numeric_domain) default_st => store set" where
  "default_st_gamma 𝒢 s = γ (ϱ𝒢 s)"

lemma default_st_gamma_mono:
  "s ≤ t ⟹ default_st_gamma 𝒢 s ⊆ default_st_gamma 𝒢 t"
  unfolding default_st_gamma_def by (intro gamma_state_mono default_st_to_fun_mono)

lemma default_st_gamma_bot [simp]: "default_st_gamma 𝒢 bot = {}"
  by (simp add: default_st_gamma_def)

lemma default_st_gamma_initial:
  "default_st_gamma 𝒢 (initial_default_st local_value global_value) =
   γ (λx. if 𝒢 x then global_value else local_value)"
```

```
  by (simp add: default_st_gamma_def default_st_to_fun_initial)

unbundle no default_st_syntax

end
```

7.4 Default_St_Reachability

```
theory Default_St_Reachability
  imports Default_St_Transfer "Voblint_Domain.Nonrelational_Reachability"
begin

unbundle default_st_syntax
```

A lifted state reads back pointwise under the lift: `Bot` stays `Bot`, and a `Lifted` state becomes `Lifted` of the function it represents. This is how a solved local unknown, stored as an `'a default_st lifted`, enters the result boundary. The readback performs no bottom test. `canonicalize_lift` collapses a witness-bottom payload to `Bot` before it, and coverage stays a property of the solve: a covered key whose value is `Bot` remains a key.

```
adhoc_overloading readback == "λG. map_lift (default_st_to_fun G)"
```

Executable dead-code detection

A nonrelational state denotes the product of its variables' value sets, so a single bottom component makes the whole state denote nothing. Deciding that by inspecting every variable is not executable – there are infinitely many names – so this theory builds a finite test and proves it says exactly what the infinite one does.

The finite test has three witness sources: the local default, which covers every name no entry mentions; an explicitly stored entry that is bottom in the dictionary the classifier selects for its name; and a scan of the program's declared globals, which covers the side where no fresh-name argument is available. The middle one needs the classifier, since an entry stored in the other dictionary says nothing about the function the state represents. Above that sits the lifted view, where a whole unreachable state collapses to `Bot` and an update that writes a bottom value collapses to it incrementally, without re-testing the rest of the state.

Raw witness test

Bottom detection is the one carrier operation that consults $\mathcal{G}$, and the reason is the mismatch between two readings of the same state. The quotient `'a default_st` identifies two states exactly when their lookups agree at *every* location, whereas `default_st_rep_to_fun` reads each name in the one dictionary $\mathcal{G}$ selects for it. An entry for a name in the other dictionary is therefore visible to equality and invisible to the concretization, and a test that must agree with `is_empty_state` on the represented function has to skip it. Every other operation on the carrier – lookup, update, order, join, widening, narrowing – treats all locations alike and needs no classifier.

The test below is sufficient but not yet exact, and takes three disjuncts: the local default already bottom, which covers every name no entry mentions; or a local entry, for a name $\mathcal{G}$ leaves local, that is bottom; or a global entry, for a name $\mathcal{G}$ calls global, that is bottom. The global default is deliberately not checked. A real program's classifier calls only finitely many names global, so no argument independent of the entry list can produce a fresh global escaping it, the way a fresh local always exists. Only the entry branch can safely observe a global; the exact test in the next subsection closes that gap by enumerating the declared globals outright.

```
fun default_st_rep_is_bot ::
  "(vname => bool) => ('a::executable_domain) default_st_rep => bool" where
  "default_st_rep_is_bot G ((dl, ls), (dg, gs)) ⟷
     is_empty dl ∨
     (∃x ∈ set (map fst ls). ¬ G x ∧ is_empty (default_dict_get (dl, ls) x)) ∨
     (∃x ∈ set (map fst gs). G x ∧ is_empty (default_dict_get (dg, gs) x))"
```

Soundness needs $\mathcal{G}$ to leave infinitely many vnames local, so that some local witness always escapes any given (finite) entry list – true for any `declared_global` of a real program, which only ever declares finitely many globals while `vname` is infinite.

```
lemma default_st_rep_is_bot_sound:
  assumes bot: "default_st_rep_is_bot G s"
    and infinite_local: "infinite {x. ¬ G x}"
  shows "is_empty_state (default_st_rep_to_fun G s)"
proof -
  obtain l g where s_eq: "s = (l, g)" by (cases s)
  obtain dl ls where l_eq: "l = (dl, ls)" by (cases l)
  obtain dg gs where g_eq: "g = (dg, gs)" by (cases g)
  have local_lookup: "default_st_rep_to_fun G s x = default_dict_get l x"
    if "¬ G x" for x
    using that unfolding default_st_rep_to_fun_def location_of_def s_eq by simp
  have global_lookup: "default_st_rep_to_fun G s x = default_dict_get g x"
    if "G x" for x
    using that unfolding default_st_rep_to_fun_def location_of_def s_eq by simp
  from bot consider
      (local_default) "is_empty dl"
```

```
    | (local_entry) x where "¬ 𝒢 x" "is_empty (default_dict_get l x)"
    | (global_entry) x where "𝒢 x" "is_empty (default_dict_get g x)"
    unfolding s_eq l_eq g_eq by auto
  then show ?thesis
  proof cases
    case local_default
    from infinite_local obtain x :: vname
      where x_local: "x ∈ {x. ¬ 𝒢 x}"
        and fresh: "x ∉ set (map fst ls)"
      by (rule obtain_fresh_vname)
    from fresh have "map_of ls x = None"
      by (simp add: map_of_resolved_none_iff)
    then have "default_dict_get l x = dl"
      unfolding l_eq by simp
    with local_default x_local have "is_empty (default_st_rep_to_fun 𝒢 s x)"
      by (simp add: local_lookup)
    then show ?thesis by (rule is_empty_stateI)
  next
    case (local_entry x)
    then have "is_empty (default_st_rep_to_fun 𝒢 s x)"
      by (simp add: local_lookup)
    then show ?thesis by (rule is_empty_stateI)
  next
    case (global_entry x)
    then have "is_empty (default_st_rep_to_fun 𝒢 s x)"
      by (simp add: global_lookup)
    then show ?thesis by (rule is_empty_stateI)
  qed
qed
```

`default_st_rep_is_bot` deliberately leaves the global default unchecked because an opaque classifier gives no way to enumerate the (finite) globally classified vnames. A concrete program's classifier is always backed by an explicit finite list (`declared_global_vars`), and enumerating exactly that list closes the gap: every globally classified vname is checked directly, so the global branch needs no entry witness. This makes the combined check an exact (not merely sound) characterization of `is_empty_state`, with no side condition on $\mathcal{G}$ beyond `globals` actually listing its true set – unlike ⟦`default_st_rep_is_bot ?𝒢 ?s; infinite {x. ¬ ?𝒢 x}`⟧ ⟹ `is_empty_state (default_st_rep_to_fun ?𝒢 ?s)`, which still needs infinitely many locals to exist.

Exact test for the declared globals

In these names `rep` marks a test on representations and `for` a test exact for an explicit list of declared globals. The classifier-only test `default_st_rep_is_bot` has no quotient counterpart: it ignores the global default, so extensionally equal representatives can disagree on it.

```
definition default_st_rep_is_bot_for ::
  "vname list => (vname => bool) => ('a::executable_domain) default_st_rep => bool" where
  "default_st_rep_is_bot_for globals 𝒢 s =
     ((∃x ∈ set globals. is_empty (default_st_rep_get s (location_of 𝒢 x))) ∨
      default_st_rep_is_bot 𝒢 s)"

lemma default_st_rep_is_bot_for_iff:
  fixes s :: "'a::executable_domain default_st_rep"
  assumes globals: "⋀x. 𝒢 x = (x ∈ set globals)"
  shows "default_st_rep_is_bot_for globals 𝒢 s ⟷ is_empty_state (default_st_rep_to_fun 𝒢 s)"
proof -
  have infinite_local: "infinite {x::vname. ¬ 𝒢 x}"
  proof
    assume fin: "finite {x::vname. ¬ 𝒢 x}"
    have "finite (UNIV :: vname set)"
    proof -
      have univ: "(UNIV :: vname set) = {x. ¬ 𝒢 x} ∪ set globals"
        using globals by auto
      have "finite ({x. ¬ 𝒢 x} ∪ set globals)" using fin by simp
      then show ?thesis unfolding univ .
    qed
    then show False using infinite_literal by simp
  qed
  show ?thesis
  proof
    assume "default_st_rep_is_bot_for globals 𝒢 s"
    then show "is_empty_state (default_st_rep_to_fun 𝒢 s)"
      unfolding default_st_rep_is_bot_for_def
    proof (elim disjE)
      assume "∃x ∈ set globals. is_empty (default_st_rep_get s (location_of 𝒢 x))"
```

```
      then obtain x where "is_empty (default_st_rep_get s (location_of 𝒢 x))" by blast
      then have "is_empty (default_st_rep_to_fun 𝒢 s x)"
        unfolding default_st_rep_to_fun_def by simp
      then show ?thesis by (rule is_empty_stateI)
    next
      assume bot: "default_st_rep_is_bot 𝒢 s"
      show ?thesis by (rule default_st_rep_is_bot_sound[OF bot infinite_local])
    qed
  next
    assume "is_empty_state (default_st_rep_to_fun 𝒢 s)"
    then obtain x where x: "is_empty (default_st_rep_to_fun 𝒢 s x)" by (rule is_empty_stateE)
    show "default_st_rep_is_bot_for globals 𝒢 s"
    proof (cases "𝒢 x")
      case True
      then have "x ∈ set globals" using globals by simp
      moreover have "default_st_rep_to_fun 𝒢 s x = default_st_rep_get s (location_of 𝒢 x)"
        unfolding default_st_rep_to_fun_def by (rule refl)
      ultimately have "∃y ∈ set globals. is_empty (default_st_rep_get s (location_of 𝒢 y))"
        using x by auto
      then show ?thesis unfolding default_st_rep_is_bot_for_def by (rule disjI1)
    next
      case False
      note local_x = False
      obtain l g where s_eq: "s = (l, g)" by (cases s)
      obtain dl ls where l_eq: "l = (dl, ls)" by (cases l)
      obtain dg gs where g_eq: "g = (dg, gs)" by (cases g)
      have empty: "is_empty (default_dict_get l x)"
        using x local_x unfolding default_st_rep_to_fun_def location_of_def s_eq by simp
      have "default_st_rep_is_bot 𝒢 s"
      proof (cases "x ∈ set (map fst ls)")
        case True
        with empty local_x
        have "∃y ∈ set (map fst ls). ¬ 𝒢 y ∧ is_empty (default_dict_get (dl, ls) y)"
          unfolding l_eq by blast
        then show ?thesis unfolding s_eq l_eq g_eq by simp
      next
        case False
        then have "map_of ls x = None"
          by (simp add: map_of_resolved_none_iff)
        with empty have "is_empty dl" unfolding l_eq by simp
        then show ?thesis unfolding s_eq l_eq g_eq by simp
      qed
      then show ?thesis unfolding default_st_rep_is_bot_for_def by (rule disjI2)
    qed
  qed
qed
```

Lifting the exact test to the quotient

`default_st_rep_is_bot_for` takes $\mathcal{G}$ as a free parameter, which makes it `eq_default_st_rep`-respectful only when $\mathcal{G}$ agrees with `globals`: a mismatched pair can pick out a witness through one representative's dictionary entries that another, extensionally equal, representative encodes via its defaults instead. Every real caller supplies the matching pair, so fixing $\mathcal{G}$ to λ`x. x ∈ set globals` internally loses nothing and restores unconditional respectfulness – which is what makes `default_st_is_bot_for` below liftable to the quotient at all.

With that pair fixed, respectfulness is a corollary of exactness rather than a separate argument: extensionally equal representatives project to the same state, and `default_st_rep_is_bot_for_iff` says the test is exactly `is_empty_state` of that projection.

```
lemma eq_default_st_rep_is_bot_for:
  assumes eq: "eq_default_st_rep s t"
  shows "default_st_rep_is_bot_for globals (λx. x ∈ set globals) s
       = default_st_rep_is_bot_for globals (λx. x ∈ set globals) t"
proof -
  have iff: "default_st_rep_is_bot_for globals (λx. x ∈ set globals) u
      = is_empty_state (default_st_rep_to_fun (λx. x ∈ set globals) u)"
    for u :: "'a default_st_rep"
    by (rule default_st_rep_is_bot_for_iff) simp
  have "default_st_rep_to_fun (λx. x ∈ set globals) s
        = default_st_rep_to_fun (λx. x ∈ set globals) t"
    unfolding default_st_rep_to_fun_def
    by (rule ext) (rule eq_default_st_repD[OF eq])
  then show ?thesis unfolding iff by simp
qed
```

The exact check transported to the quotient through `rep_default_st`. This is the predicate the executable side

trees are built against: unlike `is_empty_state` composed with `readback`, it has a genuine `[code]` equation, because it never quantifies over all of `vname`.

It takes `globals` alone and rebuilds $\mathcal{G}$ from it internally, rather than accepting a caller-supplied classifier. That is what makes `eq_default_st_rep_is_bot_for` apply unconditionally, and so what makes the definition liftable at all. Nothing is lost: every real caller already recomputes the same $\mathcal{G}$ from the same list at the call site.

```
lift_definition default_st_is_bot_for ::
  "vname list => ('a::executable_domain) default_st => bool"
  is "λglobals s. default_st_rep_is_bot_for globals (λx. x ∈ set globals) s"
  by (rule eq_default_st_rep_is_bot_for)

lemma default_st_is_bot_for_iff:
  fixes s :: "'a::executable_domain default_st"
  assumes globals: "⋀x. 𝒢 x = (x ∈ set globals)"
  shows "default_st_is_bot_for globals s ⟷ is_empty_state (ϱ_𝒢 s)"
proof -
  have gs_eq: "(λx. x ∈ set globals) = 𝒢"
    using globals by (simp add: fun_eq_iff)
  show ?thesis
    unfolding default_st_is_bot_for.rep_eq gs_eq
    by (simp add: default_st_rep_is_bot_for_iff[OF globals]
      default_st_to_fun_rep)
qed
```

The same test read through the carrier's concretization: a state passes it exactly when it describes no store.

```
lemma default_st_is_bot_for_gamma_iff:
  fixes s :: "'a::numeric_domain default_st"
  assumes globals: "⋀x. 𝒢 x = (x ∈ set globals)"
  shows "default_st_is_bot_for globals s ⟷ default_st_gamma 𝒢 s = {}"
  by (simp add: default_st_is_bot_for_iff[OF globals] default_st_gamma_def
      is_empty_state_iff_gamma_state_empty)

lemma exact_emptiness_default_st_is_bot_for:
  assumes globals: "⋀x. 𝒢 x = (x ∈ set globals)"
  shows "exact_emptiness (default_st_is_bot_for globals)
          (default_st_gamma 𝒢 :: 'a::numeric_domain default_st ⇒ store set)"
  by (rule exact_emptinessI) (rule default_st_is_bot_for_gamma_iff[OF globals])
```

The stores a lifted carrier state describes

Reachability adds no new concretization: a lifted carrier state is read through `gamma_lift` over `default_st_gamma`, as a lifted pointwise state is read through `gamma_lift` over γ. The lemma below relates the two through the represented function.

```
lemma gamma_lift_default_st_gamma_to_fun:
  "gamma_lift (default_st_gamma 𝒢) = (λd. γ⊥ (map_lift ϱ_𝒢 d))"
  by (rule ext) (simp add: gamma_lift_def default_st_gamma_def split: lifted.split)
```

Incremental dead-code tracking

`live_default_st` states that no location of the readback is empty, the condition under which a transfer result counts as reachable.

```
definition live_default_st ::
  "(vname => bool) => ('a::executable_domain) default_st => bool"
where
  "live_default_st 𝒢 s = (~ is_empty_state (ϱ_𝒢 s))"
```

`is_empty_state` on $\varrho_{\mathcal{G}}$ `s` is an infinite existential over `vname`, not executable on a quotient value. `default_st_set_lift` tracks it incrementally instead: given a `live_default_st` input and the single freshly computed element, the result is witness-bottom iff that element is `is_empty` – every other location is provably unchanged ($\varrho_{?\mathcal{G}}$ `?s⟨location_of ?𝒢 ?x := ?a⟩ = (`$\varrho_{?\mathcal{G}}$ `?s)(?x := ?a)`), so it cannot newly become bottom. This mirrors Goblint's per-analysis `Deadcode` raise while staying generic: it lives at the shared update primitive, not in each domain's own transfer code.

```
definition default_st_set_lift ::
  "('a::executable_domain) default_st lifted => location => 'a => 'a default_st lifted"
where
  "default_st_set_lift x loc a = do {
     s <- x;
     if is_empty a then Bot else Lifted s⟨loc := a⟩
   }"
```

```
lemma default_st_set_lift_Bot [simp]:
  "default_st_set_lift Bot loc a = Bot"
  unfolding default_st_set_lift_def by simp

lemma default_st_set_lift_Lifted:
  "default_st_set_lift (Lifted s) loc a =
     (if is_empty a then Bot else Lifted s⟨loc := a⟩)"
  unfolding default_st_set_lift_def by simp
```

The completeness theorem the location-scoped check relies on: from a live input, the lifted update exactly tracks the spec-level normalized result. Only the freshly written variable's element needs checking – every other variable's `abs_state` component provably survives unchanged ($\varrho_{?\mathcal{G}}$ `?s⟨location_of ?𝒢 ?x := ?a⟩ = (`$\varrho_{?\mathcal{G}}$ `?s)(?x := ?a)`), so it cannot be the source of a new witness-bottom.

```
lemma default_st_set_lift_correct:
  fixes s :: "'a::executable_domain default_st"
  assumes live: "live_default_st 𝒢 s"
  shows "map_lift ϱ𝒢
           (default_st_set_lift (Lifted s) (location_of 𝒢 x) a) =
         normalize_lift is_empty_state ((ϱ𝒢 s)(x := a))"
proof -
  from live have live': "¬ is_empty_state (ϱ𝒢 s)"
    unfolding live_default_st_def by simp
  have upd: "is_empty_state ((ϱ𝒢 s)(x := a)) = is_empty a"
    by (rule is_empty_state_fun_upd_iff[OF live'])
  show ?thesis
    by (cases "is_empty a")
      (simp_all add: default_st_set_lift_Lifted upd)
qed

unbundle no default_st_syntax

end
```

7.5 Default_St_Restriction_Refinement

```
theory Default_St_Restriction_Refinement
  imports Default_St_Reachability "Voblint_Framework.State_Restriction"
begin

unbundle default_st_syntax
```

Refinement between executable and abstract split states

The readback $\varrho_{\mathcal{G}}$ maps an executable `'a default_st` to the abstract `'a abs_state` it represents. It is a homomorphism for the local/global projections and for the routed combine, so an executable step and its abstract counterpart agree on the represented functions.

The represented function of ownership projections

```
lemma default_st_to_fun_restrict_local_for [simp]:
  "ϱ_𝒢 (restrict_local_default_st s) = restrict_local_for 𝒢 (ϱ_𝒢 s)"
  unfolding restrict_local_for_def
  by (rule ext) simp

lemma default_st_to_fun_restrict_global_for [simp]:
  "ϱ_𝒢 (restrict_global_default_st s) = restrict_global_for 𝒢 (ϱ_𝒢 s)"
  unfolding restrict_global_for_def
  by (rule ext) simp
```

The represented function of joins

The lifted readback commutes with joins and is monotone. The join law is stated for the lifted carrier itself: the generic `map_lift` form matches a schematic function and sends the simplifier into higher-order search.

```
lemma map_lift_default_st_to_fun_sup [simp]:
  "ϱ_𝒢 (a ⊔ b :: _ default_st lifted) = ϱ_𝒢 a ⊔ ϱ_𝒢 b"
  by (cases a; cases b) simp_all

lemma map_lift_default_st_to_fun_mono:
  assumes "x ≤ (y :: _ default_st lifted)"
  shows "ϱ_𝒢 x ≤ ϱ_𝒢 y"
  using assms by (cases x; cases y; simp add: default_st_to_fun_mono)

end
```

7.6 Exec_DG_State

```
theory Exec_DG_State
  imports Default_St_Restriction_Refinement "Voblint_Framework.DG_Constraint_Programs"
begin
```

The executable carrier and the functions it represents

The verified solver uses the executable association-list carrier `'a default_st`, while soundness is stated over function-valued abstract states. This theory is the bottom of the bridge: the D/G product's lattice structure and `dg_state_to_fun`, which lifts `readback` to that product: the pair of functions a D/G state represents, parametric in the classifier. Everything here is carrier-level – no specification, no transfer, no equation shape – so a domain's executable mirror is related to its abstract state once, here, and the specification layers above never restate it. D/G lattice operations are componentwise, so the product inherits the order, join, bottom, equality, and widening operations the solver requires.

Classifier-parametric represented function

The function a local or side state represents is `readback`, generic in the classifier: an executable state is written with a declaration-driven classifier, so its function needs the same classifier or it consults the wrong slot. `dg_state_to_fun` applies it to both components.

```
definition dg_state_to_fun ::
  "(vname => bool) =>
   (('a::bot) default_st, ('b::bot) default_st) dg_state => ('a abs_state, 'b abs_state) dg_state"
where
  "dg_state_to_fun 𝒢 d =
    DG (ϱ𝒢 (dg_local d)) (ϱ𝒢 (dg_global d))"

adhoc_overloading readback == dg_state_to_fun

lemma dg_state_to_fun_simps [simp]:
  "dg_local (ϱ𝒢 d) = ϱ𝒢 (dg_local d)"
  "dg_global (ϱ𝒢 d) = ϱ𝒢 (dg_global d)"
  "ϱ𝒢 (DG a b) = DG (ϱ𝒢 a) (ϱ𝒢 b)"
  by (simp_all add: dg_state_to_fun_def)

lemma dg_state_to_fun_bot [simp]:
  "ϱ𝒢 (bot :: ('a::bounded_semilattice_sup_bot default_st,
                         'b::bounded_semilattice_sup_bot default_st) dg_state) = bot"
  by (simp add: bot_dg_state_def)
end
```

7.7 Ownership_Split_Exec

```
theory Ownership_Split_Exec
  imports
    Exec_DG_State
    "Voblint_Framework.DG_Ownership_Split_Spec"
    "Voblint_Framework.Routed_Context"
begin

unbundle default_st_syntax
```

The ownership-splitting analysis at the executable carrier

`ownership_split_transfer_gen` asks only that a carrier can merge a local and a global half and project each back out. The association-list carrier answers with `combine_default_st` and the two `restrict_…_default_st` projections, so the executable analysis is that transfer at those three arguments – not a second definition of what the analysis does.

```
definition ownership_split_transfer_st ::
  "('x,'k,unit,'a::bounded_semilattice_sup_bot default_st,'a default_st) man_transfer
   ⇒ ('x,'k,unit,'a default_st,'a default_st) man_transfer"
where
  "ownership_split_transfer_st =
     ownership_split_transfer_gen combine_default_st restrict_global_default_st restrict_local_default_st"
```

Entry is the same wrapping one step up in arity: it answers a list of caller-continuation/callee-entry pairs rather than one successor value, so the executable carrier supplies the same three operations to `ownership_split_enter_transfer_gen`.

```
definition ownership_split_enter_transfer_st ::
  "('x,'k,unit,'a::bounded_semilattice_sup_bot default_st,'a default_st) man_enter_transfer
   ⇒ ('x,'k,unit,'a default_st,'a default_st) man_enter_transfer"
where
  "ownership_split_enter_transfer_st =
     ownership_split_enter_transfer_gen combine_default_st restrict_global_default_st
       restrict_local_default_st"
```

The callee exit reaches the wrapped stage merged against the same shared fact, like the caller continuation. The return slot is read off it at the classifier's own location for `ret_var`; that name is local, so the merge leaves it at the callee's own value.

```
definition ownership_split_combine_transfer_st ::
  "(vname ⇒ bool) ⇒ call_info
   ⇒ ('x,'k,unit,'a::bounded_semilattice_sup_bot default_st,'a default_st) man_combine_transfer"
where
  "ownership_split_combine_transfer_st 𝒢 ci =
     ownership_split_combine_transfer_gen combine_default_st restrict_global_default_st
       restrict_local_default_st
       (local_combine_transfer
          (λenv de. combine_assign_default_st 𝒢 (ci_dst ci)
                      de⟨location_of 𝒢 ret_var⟩ env))"

definition ownership_split_dg_spec_st_for ::
  "(vname ⇒ bool)
   ⇒ (edge_action ⇒ 'a::bounded_semilattice_sup_bot default_st ⇒ 'a default_st)
   ⇒ (call_info ⇒ 'a default_st ⇒ 'a default_st)
   ⇒ ('x,'k,unit,'a default_st,'a default_st) dg_spec"
where
  "ownership_split_dg_spec_st_for 𝒢 tf_st enter_st = local_dg_spec_template⦇
     dgs_skip := ownership_split_transfer_st (local_transfer (tf_st EA_Nop)),
     dgs_assign := (λx e. ownership_split_transfer_st (local_transfer (tf_st (EA_Assign x e)))),
     dgs_special := (λsc x. ownership_split_transfer_st (local_transfer (tf_st (EA_Special sc x)))),
     dgs_branch := (λb pol. ownership_split_transfer_st
                      (local_transfer (tf_st (if pol then EA_Assume b else EA_AssumeNot b)))),
     dgs_body := (λp. ownership_split_transfer_st (local_transfer (tf_st (EA_Body p)))),
     dgs_return := (λe p. ownership_split_transfer_st (local_transfer (tf_st (EA_Ret e p)))),
     dgs_enter := (λci. ownership_split_enter_transfer_st
                         (local_enter_transfer (λd. [(d, enter_st ci d)]))),
     dgs_event := (λev. case ev of Check_Event l bc
                     ⇒ ownership_split_transfer_st (local_transfer (tf_st (EA_Check l bc)))),
     dgs_combine_assign := ownership_split_combine_transfer_st 𝒢 ⦈"

unbundle no default_st_syntax

end
```

7.8 DG_Local_State_Exec

```
theory DG_Local_State_Exec
  imports
    Ownership_Split_Exec
begin

unbundle default_st_syntax
```

Executable Base-style DG construction

Executable mirror of `lifted_state_dg_spec`: a component over `'a default_st lifted` instead of `'a abs_state lifted`, with `tf_st` a bare `edge_action ⇒ _` dispatcher – the executable side dispatches on the action rather than naming one operation per edge kind – and `enter_st` its enter counterpart. Local-only means it reads no global and publishes none, so its compiled equations carry no `QueryG` and no `Side` – exactly as on the mathematical side.

One field differs from the mathematical construction: the env stage of `combine` is not the identity here. `combine_assign_default_` (unlike $\texttt{combine}^\sharp$) does not itself select locals-from-caller/globals-from-callee; that selection is what `combine_default_st` computes, so the env stage must compute it explicitly before the assign stage writes the return value.

The caller half of `enter` is the identity for the same reason as in the abstract Base record: the carrier relates no two locations, so a call has nothing in it to invalidate. What the env stage then merges is the raw call-site value against the callee exit.

```
definition combine_env_st_lifted ::
  "'a::bounded_semilattice_sup_bot default_st lifted ⇒ 'a default_st lifted
   ⇒ 'a default_st lifted"
where
  "combine_env_st_lifted dc de =
     (case dc of Bot ⇒ Bot | Lifted x ⇒
        (case de of Bot ⇒ Bot | Lifted y ⇒ Lifted (combine_default_st x y)))"
```

The executable analysis is a component whose edge transfers are the one dispatcher `tf_st`, lifted over the unreachable state. Its handler answers nothing.

```
definition exec_local_spec ::
  "(vname ⇒ bool) ⇒ ('a::bounded_semilattice_sup_bot default_st ⇒ bool)
   ⇒ (edge_action ⇒ 'a default_st ⇒ 'a default_st)
   ⇒ (call_info ⇒ 'a default_st ⇒ 'a default_st)
   ⇒ 'a default_st lifted local_spec" where
  "exec_local_spec 𝒢 empty_pred tf_st enter_st = make_local_spec
     (λ_ _. ⊤)
     (λ_ a. transfer_lift empty_pred (tf_st a))
     (λ_ ci p. [(fst p, transfer_lift empty_pred (enter_st ci) (fst p))])
     (λ_ _ ci. combine_env_st_lifted)
     (λ_ ci. transfer_lift2 empty_pred
        (λenv0 de0. combine_assign_default_st 𝒢 (ci_dst ci)
             de0⟨location_of 𝒢 ret_var⟩ env0))"

lemma ls_step_exec_local_spec [simp]:
  "ls_step (exec_local_spec 𝒢 empty_pred tf_st enter_st) A a = transfer_lift empty_pred (tf_st a)"
  by (simp add: exec_local_spec_def)

lemma closed_step_exec_local_spec [simp]:
  "⟦a⟧♯exec_local_spec 𝒢 empty_pred tf_st enter_st
     = transfer_lift empty_pred (tf_st a)"
  by (simp add: closed_step_def fun_eq_iff)

lemma ls_enter_exec_local_spec [simp]:
  "ls_enter (exec_local_spec 𝒢 empty_pred tf_st enter_st) A ci (d, d)
   = [(d, transfer_lift empty_pred (enter_st ci) d)]"
  by (simp add: exec_local_spec_def)

definition exec_dg_spec ::
  "(vname ⇒ bool)
   ⇒ ('a::bounded_semilattice_sup_bot default_st ⇒ bool)
   ⇒ (edge_action ⇒ 'a default_st ⇒ 'a default_st)
   ⇒ (call_info ⇒ 'a default_st ⇒ 'a default_st)
   ⇒ ('x,'k,unit,'a default_st lifted,'g::bounded_semilattice_sup_bot) dg_spec"
where
  "exec_dg_spec 𝒢 empty_pred tf_st enter_st
     = dg_spec_of (exec_local_spec 𝒢 empty_pred tf_st enter_st)"
```

Basic equations

Unfolding equations for each field of `exec_dg_spec`: step, entry and combine are the executable transfers lifted over reachability.

```
lemma dg_spec_step_exec_dg_spec:
  "dg_spec_step (exec_dg_spec 𝒢 empty_pred tf_st enter_st) a
     = local_transfer (transfer_lift empty_pred (tf_st a))"
  by (simp add: exec_dg_spec_def)

lemma dg_spec_wf_exec_dg_spec [intro, simp]:
  "dg_spec_wf (exec_dg_spec 𝒢 empty_pred tf_st enter_st)"
  by (simp add: exec_dg_spec_def)

lemma dgs_enter_exec_dg_spec:
  "enter♯ (exec_dg_spec 𝒢 empty_pred tf_st enter_st) ci
     = local_enter_transfer (λd. [(d, transfer_lift empty_pred (enter_st ci) d)])"
  by (simp add: exec_dg_spec_def)

lemma dg_spec_combine_transfer_exec_dg_spec:
  "dg_spec_combine_transfer (exec_dg_spec 𝒢 empty_pred tf_st enter_st) ci
     = local_combine_transfer
         (λdc de. transfer_lift2 empty_pred
            (λenv0 de0. combine_assign_default_st 𝒢 (ci_dst ci)
                 de0⟨location_of 𝒢 ret_var⟩ env0)
            (combine_env_st_lifted dc de) de)"
  by (simp add: exec_dg_spec_def exec_local_spec_def)

unbundle no default_st_syntax

end
```

7.9 DG_Local_State_Exec_Refinement

```
theory DG_Local_State_Exec_Refinement
  imports
    DG_Local_State_Exec
    "Voblint_Framework.DG_Reader_Transport"
begin

unbundle default_st_syntax
```

The mathematical run an executable D/G run represents

The executable construction above computes on association-list states; the soundness statements are about function-valued ones. This theory is the bridge: it fixes a domain whose executable transfer agrees with its abstract transfer on the represented functions (`dg_domain_exec`) and derives, from those three agreements alone, that a whole D/G specification is sound at the executable carrier – with no separate abstract-carrier run to compare against.

The function one return combine represents

The one fact about the executable state representation this theory needs that no lifter supplies: merging caller against callee and then writing the return value represents the mathematical $\texttt{combine}^\sharp$ of the two represented functions. Everything else transports through `transfer_lift`/`transfer_lift2` naturality.

```
lemma default_st_to_fun_combine_assign:
  "ϱ_𝒢
     (combine_assign_default_st 𝒢 dst y⟨location_of 𝒢 ret_var⟩
        (combine_default_st x y))
   = combine♯ 𝒢 dst (ϱ_𝒢 x) (ϱ_𝒢 y)"
  unfolding default_st_to_fun_def
  by (auto simp add: combine_collect_abs_def fun_eq_iff location_of_def
      split: option.splits)
```

Routed-domain compatibility, independent of any routing context

Every routed domain instance needs the same shapes – the compiled edge, enter and combine trees commuting under `readback`, plus `dg_reader_commute_gen` at that same reader – from its own executable/abstract transfer-commute facts, citing this theory's packaging theorems verbatim. Nothing in that derivation is domain-specific beyond the two primitive commute facts a domain's own executable-transfer soundness development already proves (`sign_tf.tf_st_for_commute`, `ivl_tf.tf_st_for_commute`, ...): this locale states the derivation once, so a domain interprets it instead of restating it. The locale is deliberately free of any routing context (`route`, `Seed`, `Global`, a solver): those are context-owned and solver-owned respectively, not domain-owned.

`dg_reader_commute_gen`'s instance at the same reader on both sides needs no domain fact at all: `readback` is already carrier-polymorphic and `sup`-homomorphic (`default_st_to_fun_sup`, `Voblint_Exec.Default_St_Transfer`), so this is a free-standing fact, not part of the `dg_domain_exec` locale below – keeping it outside means citing it never drags in that locale's `empty_pred`/transfer obligations.

```
lemma dg_reader_commute_gen_lifted_for:
  "dg_reader_commute_gen
     (ϱ_𝒢 :: _ default_st lifted ⇒ _) (ϱ_𝒢 :: _ default_st lifted ⇒ _)"
  by unfold_locales (simp_all add: map_lift_sup)
```

`dg_domain_exec` relates an executable transfer on `default_st` to the abstract state specification: on live states the step, the entry transfer and the emptiness test agree with their abstract counterparts after readback through `default_st_to_fun`.

```
locale dg_domain_exec =
  fixes 𝒢 :: "vname ⇒ bool"
    and empty_pred :: "'a::numeric_domain default_st ⇒ bool"
    and tf_st :: "edge_action ⇒ 'a default_st ⇒ 'a default_st"
    and enter_st :: "call_info ⇒ 'a default_st ⇒ 'a default_st"
    and sk :: "'a abs_state ⇒ 'a abs_state"
    and asn :: "vname ⇒ exp ⇒ 'a abs_state ⇒ 'a abs_state"
    and sp :: "special_call ⇒ vname ⇒ 'a abs_state ⇒ 'a abs_state"
    and br :: "exp ⇒ bool ⇒ 'a abs_state ⇒ 'a abs_state"
    and bd :: "pname ⇒ 'a abs_state ⇒ 'a abs_state"
    and rt :: "exp option ⇒ pname ⇒ 'a abs_state ⇒ 'a abs_state"
    and en :: "call_info ⇒ 'a abs_state ⇒ 'a abs_state"
    and ev :: "analysis_event ⇒ 'a abs_state ⇒ 'a abs_state"
  assumes tf_st_commute:
```

```
    "⋀a s. live_default_st 𝒢 s ⟹
       ϱ𝒢 (tf_st a s)
         = local_spec_step sk asn sp br bd rt ev a (ϱ𝒢 s)"
  and enter_st_commute:
    "⋀ci s. ϱ𝒢 (enter_st ci s)
                = en ci (ϱ𝒢 s)"
  and empty_pred_exact:
    "⋀s. empty_pred s = is_empty_state (ϱ𝒢 s)"
begin
```

With $\mathcal{G}$ fixed, γ `s` is the carrier's concretization, for a plain and for a lifted carrier state alike.

```
adhoc_overloading gamma_S == "default_st_gamma 𝒢"
adhoc_overloading gamma_S == "gamma_lift (default_st_gamma 𝒢)"

abbreviation reader :: "'a default_st lifted ⇒ 'a abs_state lifted" where
  "reader ≡ map_lift ϱ𝒢"
```

Each field's equation on the represented function, once. These are the only inputs the tree commutes below take: a local-only transfer compiles to a single answer, so its transport is exactly the pure equation on the function it wraps.

Unlike `enter_st_commute`/`combine`'s field equations, `tf_st_commute` only holds on live inputs (a domain's `branch_st`-style raw result need not match its abstract `branch`'s on a dead one), so this cannot go through the blanket `transfer_lift_commute`. `normalized_lift` supplies liveness directly instead: production only ever feeds `tf_st` a `d` that is itself `Bot` or the (always-normalized, `transfer_lift_normalized`) result of a prior step.

```
lemma step_lift_commute:
  assumes norm: "normalized_lift empty_pred d"
  shows "reader (transfer_lift empty_pred (tf_st a) d)
     = transfer_lift is_empty_state (local_spec_step sk asn sp br bd rt ev a) (reader d)"
using norm proof (cases d)
  case Bot
  then show ?thesis by (simp add: transfer_lift_def)
next
  case (Lifted s)
  with norm have "live_default_st 𝒢 s"
    by (simp add: live_default_st_def empty_pred_exact)
  then show ?thesis
    unfolding Lifted
    by (simp add: transfer_lift_def normalize_lift_def tf_st_commute empty_pred_exact)
qed

lemma enter_lift_commute:
  "reader (transfer_lift empty_pred (enter_st ci) d)
     = transfer_lift is_empty_state (en ci) (reader d)"
proof (rule transfer_lift_commute)
  show "⋀s. ϱ𝒢 (enter_st ci s)
              = en ci (ϱ𝒢 s)"
    by (simp add: enter_st_commute)
  show "⋀s. empty_pred s = is_empty_state (ϱ𝒢 s)"
    by (rule empty_pred_exact)
qed
```

The env stage merges into the assign stage: both collapse on the same two `Bot` cases, so the composed combine is a single `transfer_lift2` and transports by the same naturality lemma as every other field.

```
lemma transfer_lift2_combine_env_st_lifted:
  "transfer_lift2 empty_pred g (combine_env_st_lifted dc de) de
     = transfer_lift2 empty_pred (λx y. g (combine_default_st x y) y) dc de"
  by (cases dc; cases de) (simp_all add: combine_env_st_lifted_def transfer_lift2_def)

lemma combine_lift_commute:
  "reader (transfer_lift2 empty_pred
            (λenv0 de0. combine_assign_default_st 𝒢 dst
                 de0⟨location_of 𝒢 ret_var⟩ env0)
            (combine_env_st_lifted dc de) de)
     = transfer_lift2 is_empty_state (combine♯ 𝒢 dst) (reader dc) (reader de)"
  unfolding transfer_lift2_combine_env_st_lifted
proof (rule transfer_lift2_commute)
  show "⋀x y. ϱ𝒢
      (combine_assign_default_st 𝒢 dst y⟨location_of 𝒢 ret_var⟩
         (combine_default_st x y))
       = combine♯ 𝒢 dst (ϱ𝒢 x) (ϱ𝒢 y)"
    by (rule default_st_to_fun_combine_assign)
  show "⋀s. empty_pred s = is_empty_state (ϱ𝒢 s)"
```

```
    by (rule empty_pred_exact)
qed
```

Tree-level transport of the two specifications

The routed spine's transport hypotheses are commutes of compiled sub-trees. Both specifications are local-only, so each such commute reduces to the corresponding field equation above.

```
abbreviation spec_st :: "('x,'k,unit,'a default_st lifted,'a default_st lifted) dg_spec" where
  "spec_st ≡ exec_dg_spec 𝒢 empty_pred tf_st enter_st"

abbreviation spec_abs :: "('x,'k,unit,'a abs_state lifted,'a abs_state lifted) dg_spec" where
  "spec_abs ≡ lifted_state_dg_spec 𝒢 is_empty_state sk asn sp br bd rt en ev"

lemma Henter_lifted_for:
  "dg_reader_commute_gen.dg_enter_st_commute reader reader σ_st
     (enter♯ spec_st ci (mk_dg_man d (λ_. gk)))
     (enter♯ spec_abs ci (mk_dg_man (reader d) (λ_. gk)))"
  unfolding dgs_enter_exec_dg_spec dgs_enter_lifted_state_dg_spec
  by (rule dg_reader_commute_gen.dg_enter_st_commute_local_enter_transfer
        [OF dg_reader_commute_gen_lifted_for])
     (simp add: enter_lift_commute)

lemma Hcomb_lifted_for:
  "dg_reader_commute_gen.dg_tree_st_commute reader reader σ_st
     (sp_compile_with (λx. DG x bot)
        (dg_spec_combine_transfer spec_st ci (mk_dg_man d (λ_. gk)) de))
     (sp_compile_with (λx. DG x bot)
        (dg_spec_combine_transfer spec_abs ci (mk_dg_man (reader d) (λ_. gk)) (reader de)))"
  unfolding dg_spec_combine_transfer_exec_dg_spec
    dg_spec_combine_transfer_lifted_state_dg_spec
  by (rule dg_reader_commute_gen.dg_tree_st_commute_local_combine_transfer
        [OF dg_reader_commute_gen_lifted_for,
         where F = "transfer_lift2 is_empty_state (combine♯ 𝒢 (ci_dst ci))"])
     (rule combine_lift_commute)
```

Soundness at the executable carrier, pulled back through its functions

The framework is carrier-agnostic, so nothing forces it to be instantiated at `'a abs_state lifted`: with the carrier's own concretization `default_st_gamma`, the executable Base-style spec is itself a `analysis_contract`, and the field equations above are all that the proof needs. An instance that interprets the routed spine at `spec_st` with this concretization feeds it the solver's own table and never transports a solved system between carriers.

```
definition gamma_exec :: "'a default_st lifted ⇒ 'a default_st lifted ⇒ store set" where
  "gamma_exec d g = γ d"

lemma gamma_exec_Bot [simp]: "gamma_exec Bot g = {}"
  by (simp add: gamma_exec_def)
```

Entry is not part of `analysis_contract`, so a routed instance needs it separately. This is the same fact the collapse below proves for its own entry obligation, exported once because every routed instance over this carrier discharges its entry coverage from it.

```
lemma entered_st:
  assumes tf_sound: "sound_nonrelational_transfer 𝒢 sk asn sp br bd rt en ev"
    and s: "s ∈ γ d"
  shows "call_enter 𝒢 (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s
           ∈ γ (transfer_lift empty_pred (enter_st ci) d)"
  using s unfolding gamma_lift_default_st_gamma_to_fun enter_lift_commute
  by (intro transfer_lift_sound_mem[OF _ is_empty_state_gamma_state_empty])
     (simp add: call_enter_CallEdge
       sound_nonrelational_transfer.tf_sound_enter_entry_for[OF tf_sound])

theorem entry_pairs_cover_st:
  assumes tf_sound: "sound_nonrelational_transfer 𝒢 sk asn sp br bd rt en ev"
    and sin: "s ∈ γ d"
  shows "entry_pairs_cover (λd'. γ d') s
           (call_enter 𝒢 (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s)
           [(d, transfer_lift empty_pred (enter_st ci) d)]"
  by (rule entry_pairs_coverI
        [where cont = d and entry = "transfer_lift empty_pred (enter_st ci) d"])
     (simp_all add: sin entered_st[OF tf_sound sin])
```

The executable analysis as a component, sound for the carrier's concretization. The proof maps each state to its function through `reader`; the contract below is its one-line consequence.

```
theorem exec_local_spec_sound:
  assumes tf_sound: "sound_nonrelational_transfer 𝒢 sk asn sp br bd rt en ev"
  shows "sound_local_spec 𝒢 (λd. γ d) (exec_local_spec 𝒢 empty_pred tf_st enter_st)"
proof -
  have step: "edge_collect a (γ⊥ (reader d)) ⊆ γ⊥ (reader (transfer_lift empty_pred (tf_st a) d))"
    for a d
  proof (cases "normalized_lift empty_pred d")
    case True
    show ?thesis
      unfolding step_lift_commute[OF True]
      by (rule transfer_lift_sound_collect
            [OF sound_nonrelational_transfer.step_sound_for[OF tf_sound]
                edge_collect_empty_set is_empty_state_gamma_state_empty])
  next
    case False
    then obtain s where "d = Lifted s" "empty_pred s" by (cases d) simp_all
    then show ?thesis by (simp add: empty_pred_exact is_empty_state_gamma_state_empty)
  qed
  have comb: "combine_collect 𝒢 (ci_dst ci) s t ∈ γ⊥ (reader (transfer_lift2 empty_pred
        (λenv0 de0. combine_assign_default_st 𝒢 (ci_dst ci)
           de0⟨location_of 𝒢 ret_var⟩ env0)
        (combine_env_st_lifted dc de) de))"
    if "s ∈ γ⊥ (reader dc)" "t ∈ γ⊥ (reader de)" for s t dc de ci
    unfolding combine_lift_commute
    by (rule
      transfer_lift2_sound_mem[OF combine_collect_sound is_empty_state_gamma_state_empty that])
  have mono: "∀x y. x ≤ y ⟶ γ⊥ (reader x) ⊆ γ⊥ (reader y)"
    by (meson gamma_lift_mono gamma_state_mono map_lift_default_st_to_fun_mono)
  show ?thesis
    unfolding sound_local_spec_def gamma_lift_default_st_gamma_to_fun
    using mono subset_trans[OF edge_collect_mono[OF Int_lower1] step]
      entered_st[OF tf_sound, unfolded gamma_lift_default_st_gamma_to_fun] comb
    by (auto simp: exec_local_spec_def)
qed

theorem analysis_contract_st:
  assumes tf_sound: "sound_nonrelational_transfer 𝒢 sk asn sp br bd rt en ev"
  shows "analysis_contract spec_st (λd e. gamma_exec d (e ())) 𝒢"
  unfolding exec_dg_spec_def gamma_exec_def[abs_def]
  by (rule dg_spec_of_contract[OF exec_local_spec_sound[OF tf_sound]])

end

unbundle no default_st_syntax

end
```

7.10 Routed_Exec_Refinement

```
theory Routed_Exec_Refinement
  imports
    DG_Local_State_Exec_Refinement
begin
```

Routed execution, once for every domain and context policy

`dg_domain_exec` already reduces a domain's obligation to the routed spine to three primitive commute facts. What it deliberately does not carry is the *routing* layer: the equation system a routed analysis actually solves, and the executable-to-abstract transport of its post-solution. Each domain, at each context policy, re-derived that layer, and because `routed_call_program` mentions no domain constant, those derivations differ only in the domain carrier, the routing function and a name prefix.

This locale is that layer, stated once. It adds the seed-key pair the routed generator needs — kept as parameters rather than a fixed datatype, so a domain keeps its own key type and this locale stays independent of how that type is eventually shared — together with the executable and abstract routing functions and their agreement, and derives the commute and transport facts every routed instance needs. A context-insensitive instance passes `route_unit` on both sides, where agreement is free; a call-string instance passes its own routing function on each side and discharges the agreement from that function's own laws.

Deliberately absent: the equation-system, solved-table and result **definition**s themselves. They must stay concrete per-domain constants because they carry `[code]` equations, and `dg_domain_exec`'s own `empty_pred_exact` is not dischargeable without fixing a concrete global set, so no locale carrying it can be interpreted globally. The domain keeps its definitions; what it stops re-proving is everything below.

Also deliberately absent: the solver. The generated equation system is solver-independent, and `TD_side_upd_rule` already supplies `partial_post_solution` for every update rule on the menu, so a solver choice is an argument at the use site, never a parameter of the domain capability.

The buffered generator at any component

The buffered generator a domain actually solves, reconciled with the unbuffered one the framework is stated over, for every component run as a specification. Both reshaping hooks are the identity: the buffered generator only asks a hook to hoist what it publishes at the buffered key `buffer_key`, and a component publishes nothing there, since its edge transfers, its entry and its return all read and write the local unknown only. Its queries do not change that: its channel is a pure function of the local value, so an asking transfer is still a local one.

```
theorem pp_dg_spec_of:
  assumes S: "S = dg_spec_of c"
    and ne: "⋀p ctx. seed p ctx ≠ buffer_key"
    and pp: "part_post_solution
     (routed_node_rhs_buffered intra_predecessor_addr_list call_site_list (λ_. buffer_key) route_st
        (λctx' src a. dg_spec_edge_program S a src (λ_. buffer_key))
        (routed_call_program S (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot))
        (routed_entry_seed_programs seed)
        g bot0 d₀ g₀)
     x0 sigma_st vars"
  shows "part_post_solution
     (routed_node_rhs intra_predecessor_addr_list call_site_list (λ_. buffer_key) route_st
        (λctx' src a. dg_spec_edge_program S a src (λ_. buffer_key))
        (routed_call_program S (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot))
        (routed_entry_seed_programs seed)
        g bot0 d₀ g₀)
     x0 sigma_st vars"
proof -
  have wf: "dg_spec_wf S" by (simp add: S)
  have intra_free: "sides_of_program (dg_spec_edge_program S a src (λ_. buffer_key)) τ z = bot"
    for a src τ z
    by (simp add: S dg_spec_edge_program_def)
  have cmb_free: "sides_of_program
      (routed_call_program S (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot) route' ctx' ca cc ex)
      τ (Inr buffer_key) = bot" for route' ctx' ca cc ex τ
    by (rule routed_call_program_side_free_at_buffer_key[OF wf])
       (auto simp: S local_transfer_def local_combine_transfer_def ne
         dest!: enter_runs_local_pub_bot)
  show ?thesis
  proof (rule part_post_solution_routed_node_rhs_buffered
      [where cmb_c =
        "routed_call_program S (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot)"
         and it_c = "λctx' src a. dg_spec_edge_program S a src (λ_. buffer_key)"])
```

```
    show "⋀c' w. ∀p ∈ set (routed_contribution_programs intra_predecessor_addr_list
            call_site_list route_st (λctx' src a. dg_spec_edge_program S a src (λ_. buffer_key))
            (routed_call_program S (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot))
            (routed_entry_seed_programs seed) g c' w). sp_wf p"
      by (rule routed_contribution_programs_wf)
         (auto intro: sp_wf_dg_spec_edge_program[OF wf] sp_wf_routed_call_program[OF wf])
    then show "⋀c' w. ∀p ∈ set (routed_contribution_programs intra_predecessor_addr_list
            call_site_list route_st (λctx' src a. dg_spec_edge_program S a src (λ_. buffer_key))
            (routed_call_program S (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot))
            (routed_entry_seed_programs seed) g c' w). sp_wf p" .
    show "⋀c' src a τ. dg_local (sides_of_program (dg_spec_edge_program S a src (λ_. buffer_key)) τ
            (Inr ((λ_. buffer_key) c'))) = bot"
      by (simp add: intra_free bot_dg_state_def)
    show "⋀c' src a τ. dg_global (traverse_program (dg_spec_edge_program S a src (λ_. buffer_key)) τ)
          = dg_global (sides_of_program (dg_spec_edge_program S a src (λ_. buffer_key)) τ
              (Inr ((λ_. buffer_key) c')))"
      by (simp add: S dg_spec_edge_program_def bot_dg_state_def)
    show "⋀c' src a τ. sides_of_program (dg_spec_edge_program S a src (λ_. buffer_key)) τ
            (Inr ((λ_. buffer_key) c')) = bot"
      by (rule intra_free)
    show "⋀c' ca cc ex τ. dg_local (sides_of_program
            (routed_call_program S (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot) route_st c' ca cc ex)
            τ (Inr ((λ_. buffer_key) c'))) = bot"
      by (simp add: cmb_free bot_dg_state_def)
    show "⋀c' ca cc ex τ. dg_global (traverse_program
            (routed_call_program S (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot) route_st c' ca cc ex) τ)
          = dg_global (sides_of_program
              (routed_call_program S (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot) route_st c' ca cc ex)
              τ (Inr ((λ_. buffer_key) c')))"
      by (simp add: routed_call_program_global_free[OF wf] cmb_free bot_dg_state_def)
    show "⋀c' ca cc ex τ. sides_of_program
            (routed_call_program S (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot) route_st c' ca cc ex)
            τ (Inr ((λ_. buffer_key) c')) = bot"
      by (rule cmb_free)
    show "⋀c' w τ z x. x ∈ set (routed_entry_seed_programs seed route_st c' w)
          ⟹ sides_of_program x τ z = bot"
      by (rule routed_entry_seed_programs_free)
    show "⋀c' w τ x. x ∈ set (routed_entry_seed_programs seed route_st c' w)
          ⟹ dg_global (traverse_program x τ) = bot"
      by (rule routed_entry_seed_programs_local_only)
  qed (simp_all add: pp)
qed
```

`routed_domain_exec` adds call routing to `dg_domain_exec`: seed keys never collide with the analysis global, and routing and callee resolution on the executable state agree with their abstract versions after readback.

```
locale routed_domain_exec =
  dg_domain_exec 𝒢 empty_pred tf_st enter_st sk asn sp br bd rt en ev
  for 𝒢 :: "vname ⇒ bool"
    and empty_pred :: "'a::numeric_domain default_st ⇒ bool"
    and tf_st :: "edge_action ⇒ 'a default_st ⇒ 'a default_st"
    and enter_st :: "call_info ⇒ 'a default_st ⇒ 'a default_st"
    and sk :: "'a abs_state ⇒ 'a abs_state"
    and asn :: "vname ⇒ exp ⇒ 'a abs_state ⇒ 'a abs_state"
    and sp :: "special_call ⇒ vname ⇒ 'a abs_state ⇒ 'a abs_state"
    and br :: "exp ⇒ bool ⇒ 'a abs_state ⇒ 'a abs_state"
    and bd :: "pname ⇒ 'a abs_state ⇒ 'a abs_state"
    and rt :: "exp option ⇒ pname ⇒ 'a abs_state ⇒ 'a abs_state"
    and en :: "call_info ⇒ 'a abs_state ⇒ 'a abs_state"
    and ev :: "analysis_event ⇒ 'a abs_state ⇒ 'a abs_state" +
  fixes buffer_key :: 'k
    and seed :: "pp ⇒ 'c ⇒ 'k"
    and route_st :: "pp ⇒ 'c ⇒ 'a default_st lifted ⇒ call_action ⇒ 'c"
    and route_abs :: "pp ⇒ 'c ⇒ 'a abs_state lifted ⇒ call_action ⇒ 'c"
    and resolve_st :: "cfg ⇒ pp ⇒ pp ⇒ call_action ⇒ 'a default_st lifted ⇒ pname list"
    and resolve_abs :: "cfg ⇒ pp ⇒ pp ⇒ call_action ⇒ 'a abs_state lifted ⇒ pname list"
  assumes seed_ne_buffer_key [simp]: "⋀p ctx. seed p ctx ≠ buffer_key"
      and route_agree: "⋀u c' d ca. route_st u c' d ca
                          = route_abs u c' (readback 𝒢 d) ca"
      and resolve_agree: "⋀g w cc ca d. resolve_st g w cc ca d
                          = resolve_abs g w cc ca (readback 𝒢 d)"
begin
```

The routed combine tree commutes with the executable-to-abstract reader. `spec_st` and `spec_abs` come from `dg_domain_exec`; nothing here mentions a domain constant beyond them, so one proof serves every instance. The caller continuation needs no hypothesis: `dg_spec_combine_transfer` already runs it inside the combine sub-tree.

```
lemma dg_prog_st_commute_routed_call_program:
  "dg_reader_commute_gen.dg_prog_st_commute
     (map_lift (default_st_to_fun 𝒢)) (map_lift (default_st_to_fun 𝒢)) env
     (routed_call_program spec_st (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot) route_st ctx ca cc ex)
     (routed_call_program spec_abs (λ_. buffer_key) seed (resolve_abs g) (λd. d = Bot)
        route_abs ctx ca cc ex)"
  by (rule dg_reader_commute_gen.dg_prog_st_commute_routed_call_program
        [where Floc = "map_lift (default_st_to_fun 𝒢)"
           and Fglob = "map_lift (default_st_to_fun 𝒢)"])
     (rule dg_reader_commute_gen_lifted_for seed_ne_buffer_key
           dg_spec_wf_exec_dg_spec
           dg_spec_wf_lifted_state_dg_spec
           Henter_lifted_for Hcomb_lifted_for
           route_agree map_lift_eq_Bot_iff resolve_agree)+
```

The routed extra-goal list commutes elementwise, for the same reason.

The buffered generator a domain actually solves, reconciled with the unbuffered one the framework is stated over — at the executable spec, before publication.

Both reshaping hooks are the identity here. The buffered generator only ever asks a hook to hoist what it publishes at the buffered key `buffer_key`; this spec is local-only, so its intra tree and its routed combine publish nothing there, and each tree is already its own contribution analogue. That is a property of the trees — read off `routed_call_program_side_free_at_buffer_key` and the local-only compile-down facts — not of any analysis family: a spec whose transfers do publish at `buffer_key` would have to supply reshaped hooks instead, with the same generic bridge unchanged. The routed seed survives untouched, because a seed key is never `buffer_key` and the bridge requires off-key sides to be preserved, not removed.

An instance that interprets the spine at `spec_st` hands this post-solution to `dg_context_activation` directly.

```
abbreviation intra_st :: "'c ⇒ pp × 'c + 'k ⇒ edge_action
   ⇒ (pp × 'c, 'k, ('a default_st lifted, 'a default_st lifted) dg_state,
        ('a default_st lifted, 'a default_st lifted) dg_state) strategy_program"
where
  "intra_st ctx' src a ≡ dg_spec_edge_program spec_st a src (λ_. buffer_key)"

abbreviation cmb_st :: "cfg ⇒ (pp ⇒ 'c ⇒ 'a default_st lifted ⇒ call_action ⇒ 'c)
   ⇒ 'c ⇒ call_action ⇒ pp ⇒ pp
   ⇒ (pp × 'c, 'k, ('a default_st lifted, 'a default_st lifted) dg_state,
        ('a default_st lifted, 'a default_st lifted) dg_state) strategy_program"
where
  "cmb_st g ≡ routed_call_program spec_st (λ_. buffer_key) seed (resolve_st g) (λd. d = Bot)"

theorem pp_st:
  assumes pp: "part_post_solution
     (routed_node_rhs_buffered intra_predecessor_addr_list call_site_list (λ_. buffer_key) route_st
        intra_st (cmb_st g)
        (routed_entry_seed_programs seed)
        g bot0 d₀ g₀)
     x0 sigma_st vars"
  shows "part_post_solution
     (routed_node_rhs intra_predecessor_addr_list call_site_list (λ_. buffer_key) route_st
        intra_st (cmb_st g)
        (routed_entry_seed_programs seed)
        g bot0 d₀ g₀)
     x0 sigma_st vars"
  by (rule pp_dg_spec_of[where S = spec_st])
     (rule exec_dg_spec_def, rule seed_ne_buffer_key, rule pp)

end

end
```

Chapter 8

Voblint_Routing

8.1 Compiled_Routed_Equations

```
theory Compiled_Routed_Equations
  imports
    "Voblint_Framework.Routed_Context"
    "Voblint_Compile.Compile_Wellformed"
    "HOL-Library.RBT"
begin
```

Executable routed equations for a compiled program

A routed configuration chooses the key a node buffers its own global part at, the key each global name is read and published at, a seed-key constructor, a routing function, a D/G specification, an initial local state and an initial global state. The remaining equation-system wiring is fixed: predecessor enumeration, edge execution, static call resolution, seed publication, and bottom initialization of every other unknown.

A context-insensitive analysis is this constructor at the unit context: `Analysis_Global ()` as the only global key, `Activation_Seed` as the seed constructor and `route_unit` as the routing function. There is no second unit-context generator.

Solver policy is deliberately outside this constructor. Always-join, per-origin, and warrowing instances can solve the same equations without hiding that choice here.

The enumerations the equations read are stated over the edge relations and answer every query by re-enumerating a whole relation. The code equation at the end answers them from indexes built once per graph instead, so a solver step costs a lookup rather than a scan and sort of every edge.

```
definition compiled_routed_eqs_for ::
    "'K
      ⇒ ('N ⇒ 'K)
      ⇒ (pp ⇒ 'C ⇒ 'K)
      ⇒ (pp ⇒ 'C ⇒ 'D ⇒ call_action ⇒ 'C)
      ⇒ (pp × 'C, 'K, 'N,
           'D::bounded_semilattice_sup_bot, 'G::bounded_semilattice_sup_bot) dg_spec
      ⇒ cfg
      ⇒ 'D
      ⇒ 'G
      ⇒ (pp × 'C, 'K, ('D, 'G) dg_state) eqsT"
where
  "compiled_routed_eqs_for global global_of seed route S g initial initial_global =
     routed_node_rhs_buffered intra_predecessor_addr_list call_site_list
       (λ_. global) route
       (λctx' src a. dg_spec_edge_program S a src global_of)
       (routed_call_program S global_of seed (static_resolve g) (λd. d = bot))
       (routed_entry_seed_programs seed)
       g bot initial initial_global"
```

Per-node edge indexes

`group_by_key key_of f xs` files each element `f` accepts under its key, keeping list order within a key. A lookup is therefore the order-preserving filter of `xs` at that key, which is what lets it replace an enumeration inside a fold that observes order.

```
definition group_lookup :: "('k::linorder, 'y list) rbt ⇒ 'k ⇒ 'y list" where
  "group_lookup m k = (case RBT.lookup m k of None ⇒ [] | Some ys ⇒ ys)"

definition group_by_key ::
    "('x ⇒ 'k::linorder) ⇒ ('x ⇒ 'y option) ⇒ 'x list ⇒ ('k, 'y list) rbt" where
  "group_by_key key_of f xs =
     foldr (λx m. case f x of
                None ⇒ m
              | Some y ⇒ RBT.insert (key_of x) (y # group_lookup m (key_of x)) m)
       xs RBT.empty"

lemma group_lookup_group_by_key:
  "group_lookup (group_by_key key_of f xs) k
     = List.map_filter (λx. if key_of x = k then f x else None) xs"
  by (induction xs) (auto simp: group_by_key_def group_lookup_def split: option.splits)

definition intra_predecessor_index ::
    "cfg ⇒ (cfg_node, (cfg_node × edge_action) list) rbt" where
  "intra_predecessor_index g =
     group_by_key (λ(u, a, w). w) (λ(u, a, w). Some (u, a)) (cfg_intra_list g)"

lemma group_lookup_intra_predecessor_index:
```

```
  "group_lookup (intra_predecessor_index g) v = intra_predecessor_list g v"
proof -
  have "List.map_filter
         (λx. if (λ(u, a, w). w) x = v then (λ(u, a, w). Some (u, a)) x else None) es
         = map (λ(u, a, w). (u, a)) (filter (λ(u, a, w). w = v) es)"
    for es :: "(cfg_node × edge_action × cfg_node) list"
    by (induction es) auto
  then show ?thesis
    unfolding intra_predecessor_index_def intra_predecessor_list_def group_lookup_group_by_key
    by simp
qed

lemma map_group_lookup_intra_predecessor_index:
  "map (λ(u, a). (Inl (u, ctx), a)) (group_lookup (intra_predecessor_index g) v)
     = intra_predecessor_addr_list g v ctx"
  by (simp add: intra_predecessor_addr_list_def group_lookup_intra_predecessor_index)

definition call_target_index ::
    "cfg ⇒ (cfg_node, (cfg_node × call_action × pname) list) rbt" where
  "call_target_index g =
     group_by_key (λ(c, ca, ce, k). k) (λ(c, ca, ce, k). call_target_at k (c, ca, ce, k))
       (cfg_calls_list g)"

lemma group_lookup_call_target_index:
  "group_lookup (call_target_index g) v = call_target_list g v"
proof -
  have "(if (λ(c, ca, ce, k). k) e = v
        then (λ(c, ca, ce, k). call_target_at k (c, ca, ce, k)) e else None)
         = call_target_at v e" for e :: "cfg_node × call_action × cfg_node × cfg_node"
  proof -
    obtain c ca ce k where e: "e = (c, ca, ce, k)" by (cases e) auto
    show ?thesis by (cases ce) (auto simp: e)
  qed
  then show ?thesis
    unfolding call_target_index_def call_target_list_def group_lookup_group_by_key
    by simp
qed
```

The indexes are let-bound outside the equation system, so a solver holding the system evaluates them once. The three selectors restate `intra_predecessor_addr_list`, `call_site_list` and `static_resolve` over a lookup in place of the enumeration. They ignore their graph argument, which is sound only because the generator applies every selector at the one graph it is given.

```
lemma compiled_routed_eqs_for_code [code]:
  "compiled_routed_eqs_for global global_of seed route S g initial initial_global =
     (let preds = intra_predecessor_index g; targets = call_target_index g
      in routed_node_rhs_buffered
           (λ_ v ctx. map (λ(u, a). (Inl (u, ctx), a)) (group_lookup preds v))
           (λ_ v. remdups (map (λ(c, ca, p). (c, ca)) (group_lookup targets v)))
           (λ_. global) route
           (λctx' src a. dg_spec_edge_program S a src global_of)
           (routed_call_program S global_of seed
              (λv cc ca d. map (λ(c, a, p). p)
                 (filter (λ(c, a, p). c = cc ∧ a = ca) (group_lookup targets v)))
              (λd. d = bot))
           (routed_entry_seed_programs seed)
           g bot initial initial_global)"
proof -
  note preds = map_group_lookup_intra_predecessor_index
  have sites: "remdups (map (λ(c, ca, p). (c, ca)) (group_lookup (call_target_index g) v))
     = call_site_list g v" for v
    by (simp add: call_site_list_def group_lookup_call_target_index)
  have resolve: "(λv cc ca d. map (λ(c, a, p). p)
                   (filter (λ(c, a, p). c = cc ∧ a = ca) (group_lookup (call_target_index g) v)))
     = static_resolve g"
    by (intro ext) (simp add: static_resolve_def static_targets_def group_lookup_call_target_index)
  show ?thesis
    unfolding compiled_routed_eqs_for_def routed_node_rhs_buffered_def
      routed_contribution_programs_def Let_def
    by (simp only: preds sites resolve)
qed

end
```

8.2 Call_String_Routed_Context

```
theory Call_String_Routed_Context
  imports
    Compiled_Routed_Equations
    "Voblint_Framework.Call_String_Context"
begin
```

Telling activations apart by how they were called

A call string is the last `k` call sites on the stack. Keying an analysis by one means a procedure entered from two places is analysed twice, separately, instead of once at the join of both — and entered from more than `k` levels deep, the oldest sites drop off and those activations merge again.

This theory says that policy is a legal routing for the generic spine, and pays once, for every `k` and every domain, the part of that bill which depends only on the policy and on the shape of a compiled program:

- `finC` holds for every `compile_prog` output (`compile_prog_finite`);
- `calls_unique` is call-source uniqueness for every `compile_prog` output (`compile_prog_calls_source_unique`);
- `seed_ne_buffer_key` is datatype distinctness for `call_string_gk`;
- `routing_adequate`'s routing conjunct is `cs_route` and `cs_context` being the same closed term (`cs_route_context_agree`), independently of the entered value, so every context the relation admits is the one `cs_route` computes on any alternative;
- `routing_total` is immediate: the graph of a function admits exactly one context per call.

`call_fwd` and `comb_fwd` remain assumptions, and deliberately so: each says the solved variable set covers a particular routed callee entry or return continuation, which is a property of the program together with what the solver actually explored. An instance supplies them; no generic argument can.

```
definition call_string_eqs_for ::
    "nat
      ⇒ (pp × call_string, call_string_gk, unit,
           'D::bounded_semilattice_sup_bot, 'G::bounded_semilattice_sup_bot) dg_spec
      ⇒ cfg
      ⇒ 'D
      ⇒ (pp × call_string, call_string_gk, ('D, 'G) dg_state) eqsT"
where
  "call_string_eqs_for k S g initial =
     compiled_routed_eqs_for Call_String_Context.Global
       (λ_. Call_String_Context.Global) Call_String_Context.Seed (cs_route k) S g initial bot"
```

The constructor fixes the call-string routing, global and seed keys, compiled-program resolver, and bottom initialization. A domain supplies only its specification, graph, and initial local state.

```
locale call_string_routed_context =
  dg_context_activation S "λd e. γ_DG d (e ())" 𝒢 "compile_prog Pi ps" Global "λ_. Global"
    "cs_route k"
    "routed_call_program S (λ_. Global) Seed (static_resolve (compile_prog Pi ps)) is_bot"
    "routed_entry_seed_programs Seed"
    bot0 d₀ g₀ sigma vars x0 sg γ_M
  for S :: "(pp × cfg_node list, call_string_gk, unit, 'D::bounded_semilattice_sup_bot,
              'G::bounded_semilattice_sup_bot) dg_spec"
    and γ_DG :: "'D ⇒ 'G ⇒ store set"
    and 𝒢 :: "vname ⇒ bool"
    and Pi :: proc_table and ps :: "pname list"
    and k :: nat
    and bot0 d₀ g₀ sigma vars x0 sg
    and is_bot :: "'D ⇒ bool"
    and γ_M :: "'M ⇒ store set" +
  assumes is_bot_bot: "is_bot bot"
    and is_bot_sound: "⋀d gv. is_bot d ⟹ γ_DG d gv = {}"
    and enter_complete:
    "⋀u ctx dst pars args p cont s.
       (u, ctx) ∈ vars
       ⟹ (u, CallEdge dst pars args, FunctionEntry p, cont)
            ∈ calls (compile_prog Pi ps)
       ⟹ s ∈ γ_DG (dg_local (sigma (Inl (u, ctx)))) (dg_global (sigma (Inr Global)))
       ⟹ ∃pairs pub deps.
            enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
              (mk_dg_man (dg_local (sigma (Inl (u, ctx)))) (λ_. Global)) sigma pairs pub
```

```
          ∧ enter_deps (enter♯ S (call_info_of (CallEdge dst pars args) p))
              (mk_dg_man (dg_local (sigma (Inl (u, ctx)))) (λ_. Global)) sigma pairs deps
          ∧ entry_pairs_cover (λd. γ_DG d (dg_global (sigma (Inr Global)))) s
              (call_enter G (CallEdge dst pars args) s) pairs"
    and call_fwd:
    "⋀u ctx dst pars args p cont.
       (u, ctx) ∈ vars
       ⟹ (u, CallEdge dst pars args, FunctionEntry p, cont)
            ∈ calls (compile_prog Pi ps)
       ⟹ (FunctionEntry p,
             cs_route k u ctx (dg_local (sigma (Inl (u, ctx)))) (CallEdge dst pars args))
            ∈ vars"
    and comb_fwd:
    "⋀cl c1 dst pars args p cont.
       (cl, c1) ∈ vars
       ⟹ (cl, CallEdge dst pars args, FunctionEntry p, cont)
            ∈ calls (compile_prog Pi ps)
       ⟹ (cont, c1) ∈ vars"
begin

sublocale routed: routed_context S "λd e. γ_DG d (e ())" G "compile_prog Pi ps" Global
  "λ_. Global"
  "cs_route k" bot0 d_0 g_0 sigma vars x0 sg Seed
  "static_resolve (compile_prog Pi ps)" is_bot γ_M "context_policy_of_fun (cs_context k)"
proof unfold_locales
  show "finite (calls (compile_prog Pi ps))" by (simp add: compile_prog_finite)
next
  show "calls_source_unique (compile_prog Pi ps)"
    unfolding calls_source_unique_def using compile_prog_calls_source_unique by blast
next
  show "⋀p ctx. Seed p ctx ≠ Global" by simp
next
  show "is_bot bot" by (rule is_bot_bot)
next
  show "⋀d gv. is_bot d ⟹ γ_DG d gv = {}" by (rule is_bot_sound)
next
  fix u ctx dst pars args p cont s
  assume "(u, ctx) ∈ vars"
    and "(u, CallEdge dst pars args, FunctionEntry p, cont)
           ∈ calls (compile_prog Pi ps)"
    and "s ∈ gamma_at u ctx"
  then show "p ∈ set (static_resolve (compile_prog Pi ps) cont u
                       (CallEdge dst pars args) (dg_local (sigma (Inl (u, ctx)))))"
    by (simp add: compile_prog_finite)
next
  fix u ctx dst pars args p cont s ctx'
  assume covV: "(u, ctx) ∈ vars"
    and ce: "(u, CallEdge dst pars args, FunctionEntry p, cont)
               ∈ calls (compile_prog Pi ps)"
    and sin: "s ∈ gamma_at u ctx"
    and Rc: "ctx' ∈ context_policy_of_fun (cs_context k) u ctx (call_info_of (CallEdge dst pars args) p)
               s (call_enter G (CallEdge dst pars args) s)"
  have ctx': "ctx' = cs_context k u ctx (call_enter G (CallEdge dst pars args) s)"
    using Rc by simp
  obtain pairs pub deps
    where R: "enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
                (man_at u ctx) sigma pairs pub"
      and D: "enter_deps (enter♯ S (call_info_of (CallEdge dst pars args) p))
                (man_at u ctx) sigma pairs deps"
      and P: "entry_pairs_cover (λd. γ_DG d (dg_global (sigma (Inr Global)))) s
                (call_enter G (CallEdge dst pars args) s) pairs"
    using enter_complete[OF covV ce sin[simplified]] by blast
  from P obtain cont' entry
    where mem: "(cont', entry) ∈ set pairs"
      and ccov: "s ∈ γ_DG cont' (dg_global (sigma (Inr Global)))"
      and ecov: "call_enter G (CallEdge dst pars args) s
                   ∈ γ_DG entry (dg_global (sigma (Inr Global)))"
    by (rule entry_pairs_coverE)
  have req: "cs_route k u ctx entry (CallEdge dst pars args) = ctx'"
    unfolding ctx' by (rule cs_route_context_agree)
  have covE: "(FunctionEntry p, ctx') ∈ vars"
    using call_fwd[OF covV ce] req by (simp add: cs_route_def)
  show "∃pairs pub deps cont' entry.
          enter_runs (enter♯ S (call_info_of (CallEdge dst pars args) p))
            (man_at u ctx) sigma pairs pub
          ∧ enter_deps (enter♯ S (call_info_of (CallEdge dst pars args) p))
```

```
            (man_at u ctx) sigma pairs deps
          ∧ (cont', entry) ∈ set pairs
          ∧ s ∈ γ_DG cont' (genv (λ_. Global) sigma ())
          ∧ call_enter 𝒢 (CallEdge dst pars args) s
              ∈ γ_DG entry (genv (λ_. Global) sigma ())
          ∧ cs_route k u ctx entry (CallEdge dst pars args) = ctx'
          ∧ (FunctionEntry p, ctx') ∈ vars"
    using R D mem ccov ecov req covE by auto
next
  fix u ctx dst pars args p cont s
  show "context_policy_of_fun (cs_context k) u ctx
                (call_info_of (CallEdge dst pars args) p) s
                (call_enter 𝒢 (CallEdge dst pars args) s) ≠ {}"
    by simp
next
  fix cl c1 dst pars args p cont
  assume "(cl, c1) ∈ vars"
    and "(cl, CallEdge dst pars args, FunctionEntry p, cont)
          ∈ calls (compile_prog Pi ps)"
  then show "(cont, c1) ∈ vars" by (rule comb_fwd)
next
  show "⋀p ctx v. Seed p ctx ≠ Global" by simp
qed

end

end
```

8.3 Entry_State_Routed_Context

```
theory Entry_State_Routed_Context
  imports "Voblint_Framework.Routed_Context" "Voblint_Compile.Compile_Wellformed"
begin
```

Routing when the entry operation is a pure list of alternatives

A call can be answered with several possible entry states rather than one. This theory covers the case where computing that list is a *pure* step: it reads no solver unknown and publishes nothing to a global, so the entry run has an empty dependency set and a bottom publication.

`alts ci d` is that list — the alternatives the specification answers for call `ci` at caller state `d` — and `enter_pure` is the assumption that the specification's own $\texttt{enter}^\sharp$ is exactly that list handed to its continuation. What the locale restricts is therefore the entry operation, not the route; hence the name.

`route` itself stays unconstrained: it may read the call site, the caller's context and the entered state, or ignore any of them. Entry-state routing is its main application and the reason the parameter is there — `formals_route_lifted_gen` projects the formals out of an abstract state, an executable instance projects them out of its own quotient state, and one policy spelled two ways is what a parameter is for — but nothing here forces a route to look at the entered value at all. Unlike `Call_String_Routed_Context`'s `call_string_routed_context`, no single seed-key datatype is shared across instances: each is keyed by its own carrier's value list, so `buffer_key` and `seed` stay genuine locale parameters too.

Everything the routed locale asks about `adm` reduces to two facts about `alts`: its alternatives cover every concrete call at a covered call site (`enter_cover`), and the solver visited the callee entry at every context a live alternative routes to (`call_fwd`). A one-alternative specification instantiates `alts ci d = [(d, en ci d)]`; one that splits a call into several alternatives, possibly overlapping, instantiates it with the longer list and nothing else changes.

Two of the routed obligations are facts about `compile_prog` alone and are discharged here once: `finC` (`compile_prog_finite`) and `calls_unique` (`compile_prog_calls_source_unique`). `seed_ne_buffer_key`, `call_fwd` and `comb_fwd` remain per-instance assumptions: the first is datatype distinctness of the instance's seed key, the other two say the solved variable set covers a routed callee entry or return continuation, a property of the program together with what the solver explored.

```
locale pure_entry_routed_context =
  dg_context_activation S "λd e. γ_DG d (e ())" 𝒢 "compile_prog Pi ps" buffer_key
    "λ_. buffer_key" route
    "routed_call_program S (λ_. buffer_key) seed (static_resolve (compile_prog Pi ps)) is_bot"
    "routed_entry_seed_programs seed"
    bot0 d₀ g₀ sigma vars x0 sg γ_M
  for S :: "(pp × 'c, 'k, unit, 'D::bounded_semilattice_sup_bot,
             'G::bounded_semilattice_sup_bot) dg_spec"
    and γ_DG :: "'D ⇒ 'G ⇒ store set"
    and 𝒢 :: "vname ⇒ bool"
    and Pi :: proc_table and ps :: "pname list"
    and buffer_key :: 'k
    and route :: "pp ⇒ 'c ⇒ 'D ⇒ call_action ⇒ 'c"
    and bot0 d₀ g₀ sigma vars x0 sg
    and seed :: "pp ⇒ 'c ⇒ 'k"
    and is_bot :: "'D ⇒ bool"
    and γ_M :: "'M ⇒ store set"
    and alts :: "call_info ⇒ 'D ⇒ 'D enter_result list" +
  assumes seed_ne_buffer_key[simp]: "⋀p ctx. seed p ctx ≠ buffer_key"
    and is_bot_bot: "is_bot bot"
    and is_bot_sound: "⋀d gv. is_bot d ⟹ γ_DG d gv = {}"
    and enter_pure: "⋀ci. enter♯ S ci = local_enter_transfer (alts ci)"
    and enter_cover:
    "⋀u ctx dst pars args p cont s.
       (u, ctx) ∈ vars
       ⟹ (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls (compile_prog Pi ps)
       ⟹ s ∈ γ_DG (dg_local (sigma (Inl (u, ctx)))) (dg_global (sigma (Inr buffer_key)))
       ⟹ entry_pairs_cover (λd. γ_DG d (dg_global (sigma (Inr buffer_key)))) s
            (call_enter 𝒢 (CallEdge dst pars args) s)
            (alts (call_info_of (CallEdge dst pars args) p) (dg_local (sigma (Inl (u, ctx)))))"
    and call_fwd:
    "⋀u ctx dst pars args p cont cont' entry.
       (u, ctx) ∈ vars
       ⟹ (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls (compile_prog Pi ps)
       ⟹ (cont', entry) ∈ set (alts (call_info_of (CallEdge dst pars args) p)
                                    (dg_local (sigma (Inl (u, ctx)))))
       ⟹ ¬ is_bot entry
       ⟹ (FunctionEntry p, route u ctx entry (CallEdge dst pars args)) ∈ vars"
    and comb_fwd:
```

```
    "⋀cl c1 dst pars args p cont.
       (cl, c1) ∈ vars
       ⟹ (cl, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls (compile_prog Pi ps)
       ⟹ (cont, c1) ∈ vars"
begin
```

The context relation this instance keys its collecting semantics by.

```
abbreviation entry_context_rel :: "'c context_policy" where
  "entry_context_rel ≡ routed_entry_context_rel alts (λd e. γ_DG d (e ())) sigma (λ_. buffer_key) route"

sublocale routed: routed_context S "λd e. γ_DG d (e ())" G "compile_prog Pi ps" buffer_key
  "λ_. buffer_key"
  route bot0 d₀ g₀ sigma vars x0 sg seed
  "static_resolve (compile_prog Pi ps)" is_bot γ_M entry_context_rel
proof unfold_locales
  show "finite (calls (compile_prog Pi ps))" by (simp add: compile_prog_finite)
next
  show "calls_source_unique (compile_prog Pi ps)"
    unfolding calls_source_unique_def using compile_prog_calls_source_unique by blast
next
  show "⋀p ctx. seed p ctx ≠ buffer_key" by (rule seed_ne_buffer_key)
next
  show "is_bot bot" by (rule is_bot_bot)
next
  show "⋀d gv. is_bot d ⟹ γ_DG d gv = {}" by (rule is_bot_sound)
next
  fix u ctx dst pars args p cont s
  assume "(u, ctx) ∈ vars"
    and "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls (compile_prog Pi ps)"
    and "s ∈ gamma_at u ctx"
  then show "p ∈ set (static_resolve (compile_prog Pi ps) cont u
                       (CallEdge dst pars args) (dg_local (sigma (Inl (u, ctx)))))"
    by (simp add: compile_prog_finite)
next
  fix u ctx dst pars args p cont s ctx'
  assume covV: "(u, ctx) ∈ vars"
    and ce: "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls (compile_prog Pi ps)"
    and "s ∈ gamma_at u ctx"
    and Rc: "ctx' ∈ entry_context_rel u ctx (call_info_of (CallEdge dst pars args) p) s
               (call_enter G (CallEdge dst pars args) s)"
  let ?ci = "call_info_of (CallEdge dst pars args) p"
  let ?d = "dg_local (sigma (Inl (u, ctx)))"
  from Rc obtain cont' entry
    where mem: "(cont', entry) ∈ set (alts ?ci ?d)"
      and ccov: "s ∈ γ_DG cont' (dg_global (sigma (Inr buffer_key)))"
      and ecov: "call_enter G (CallEdge dst pars args) s
                   ∈ γ_DG entry (dg_global (sigma (Inr buffer_key)))"
      and req0: "ctx' = route u ctx entry (CallEdge (ci_dst ?ci) (ci_formals ?ci) (ci_args ?ci))"
    using Rc by (auto elim: routed_entry_context_relE)
  have req: "route u ctx entry (CallEdge dst pars args) = ctx'" using req0 by simp
  have not_bot: "¬ is_bot entry"
    using ecov is_bot_sound by fastforce
  have Rr: "enter_runs (enter♯ S ?ci) (mk_dg_man ?d (λ_. buffer_key)) sigma (alts ?ci ?d) bot"
    unfolding enter_pure by (rule enter_runs_local_enter_transfer_mk_dg_man)
  have D: "enter_deps (enter♯ S ?ci) (mk_dg_man ?d (λ_. buffer_key)) sigma (alts ?ci ?d) {}"
    unfolding enter_pure by (rule enter_deps_local_enter_transfer_mk_dg_man)
  show "∃pairs pub deps cont' entry.
            enter_runs (enter♯ S ?ci) (mk_dg_man ?d (λ_. buffer_key)) sigma pairs pub
          ∧ enter_deps (enter♯ S ?ci) (mk_dg_man ?d (λ_. buffer_key)) sigma pairs deps
          ∧ (cont', entry) ∈ set pairs
          ∧ s ∈ γ_DG cont' (genv (λ_. buffer_key) sigma ())
          ∧ call_enter G (CallEdge dst pars args) s
              ∈ γ_DG entry (genv (λ_. buffer_key) sigma ())
          ∧ route u ctx entry (CallEdge dst pars args) = ctx'
          ∧ (FunctionEntry p, ctx') ∈ vars"
    using Rr D mem ccov ecov req call_fwd[OF covV ce mem not_bot] by auto
next
  fix u ctx dst pars args p cont s
  assume covV: "(u, ctx) ∈ vars"
    and ce: "(u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls (compile_prog Pi ps)"
    and sin: "s ∈ gamma_at u ctx"
  show "entry_context_rel u ctx (call_info_of (CallEdge dst pars args) p) s
              (call_enter G (CallEdge dst pars args) s) ≠ {}"
  proof (rule routed_entry_context_rel_total)
    show "entry_pairs_cover (λd. γ_DG d (genv (λ_. buffer_key) sigma ())) s
            (call_enter G (CallEdge dst pars args) s)
```

```
            (alts (call_info_of (CallEdge dst pars args) p) (dg_local (sigma (Inl (u, ctx)))))"
      using enter_cover[OF covV ce sin[simplified]] by simp
  qed
next
  fix cl c1 dst pars args p cont
  assume "(cl, c1) ∈ vars"
    and "(cl, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls (compile_prog Pi ps)"
  then show "(cont, c1) ∈ vars" by (rule comb_fwd)
next
  show "⋀p ctx v. seed p ctx ≠ buffer_key" by simp
qed

end

end
```

8.4 Context_Space_Finite

```
theory Context_Space_Finite
  imports "Voblint_Framework.Call_String_Context" "Voblint_Compile.Compile_Wellformed"
begin
```

Which context policies can have a finite solved key set

A published result table is well formed only if its key set is finite (`wf_solved_table`, in the framework's result layer). A key is a `(node, context)` pair, so one way to get there is to exhibit a fixed finite set the keys are drawn from and appeal to `finite_subset`. This theory builds that set, one per context policy.

What it builds is a candidate space, not a reachability or a termination result. Nothing below says which keys a solver run actually produces, and nothing below bounds how often the value at one key is raised – an interval unknown can ascend forever inside a one-element key space. Both halves of the containment are therefore hypotheses of every result here, not conclusions:

- `fst ‘ vars ⊆ cfg_nodes (compile_prog Pi ps)` — the keys are nodes of the program the solver was given. There is no theorem here connecting the vendored solver's returned key set to the unknowns its equation system mentions.
- `snd ‘ vars` lies in the policy's context space — routing never left it. For call strings that is a length-and-alphabet condition, and truncation alone does not establish it: a starting context whose elements are not nodes of this program stays short under `cs_route` without ever entering the space.

What differs between policies is how cheaply the second hypothesis can be met. Call strings bound the length for free: `cs_route` truncates every context to length at most `k` (`cs_route_length`, `Voblint_Framework.Call_String_Context`), and a compiled program's CFG has finitely many nodes (`cfg_nodes_finite`, `Voblint_CFG.CFG_Def`), so with the standard library's own `finite_lists_length_le` the space is finite for every `k` and every domain alike. The monovariant policy makes the hypothesis vacuous: its context type is `unit`, so there is no context space to bound and the node half is the whole story.

An entry-state context is a domain value rather than a bounded-length list over a finite alphabet, so it needs a separate argument. A finite value domain together with a fixed formals count supplies one – `sign list` at the callee's arity is a finite set – but an infinite-height domain such as `ivl` does not, and an ordinary widening policy bounds values rather than the context identities keyed by them. Bounding those needs a policy that controls key creation or merges keys (a gas budget, a tabulation cap), with real precision consequences that nothing here specifies. This development stops at the two policies whose context space needs no such decision.

The concrete analyses reach a finite key set by an unrelated route: they discharge their own obligation from the vendored solver's `finite_stabl_solve`, which yields a finite stable set from a terminating solve. Neither argument implies the other – a bounded candidate space does not make a solve terminate, and a terminating solve says nothing about the space its keys were drawn from.

```
lemma call_strings_bounded_finite:
  assumes "finite A"
  shows "finite {cs::call_string. set cs ⊆ A ∧ length cs ≤ k}"
  using assms by (rule finite_lists_length_le)

theorem compiled_call_strings_finite:
  "finite {cs::call_string. set cs ⊆ cfg_nodes (compile_prog Pi ps) ∧ length cs ≤ k}"
  using cfg_nodes_finite[OF compile_prog_finite[THEN conjunct1] compile_prog_finite[THEN conjunct2]]
  by (rule call_strings_bounded_finite)
```

The practical corollary a `call_string_routed_context` instance can cite directly: every `(node, call-string)` pair a `k`-bounded call-string analysis over a compiled program could ever solve for lies in a fixed finite set, so `vars` — whatever a particular solver run actually populates it with — is finite because it lies in a finite superset, not merely because the run happened to terminate.

```
theorem compiled_call_string_vars_finite:
  assumes nodes: "fst ‘ vars ⊆ cfg_nodes (compile_prog Pi ps)"
    and ctxs: "snd ‘ vars ⊆ {cs::call_string. set cs ⊆ cfg_nodes (compile_prog Pi ps)
                               ∧ length cs ≤ k}"
  shows "finite vars"
proof -
  have "vars ⊆ cfg_nodes (compile_prog Pi ps)
         × {cs::call_string. set cs ⊆ cfg_nodes (compile_prog Pi ps) ∧ length cs ≤ k}"
    using nodes ctxs by(auto; force)
  moreover have "finite (cfg_nodes (compile_prog Pi ps)
         × {cs::call_string. set cs ⊆ cfg_nodes (compile_prog Pi ps) ∧ length cs ≤ k})"
    using cfg_nodes_finite compile_prog_finite compiled_call_strings_finite
```

```
    by (blast intro: finite_cartesian_product)
  ultimately show ?thesis by (rule finite_subset)
qed
```

The seed-key space follows the same shape: one `Global` slot plus one `Seed` slot per (callee-entry node, call-string) pair. `seed_pp` ranges over `pp`, which is `cfg_node` (`Voblint_CFG.CFG_Def`) – a call-string seed is keyed by the callee's `FunctionEntry` node, not the raw procedure name – so this reuses `cfg_nodes_finite` again rather than needing a separate finiteness fact about `ps`.

```
theorem compiled_call_string_gk_finite:
  "finite ({Global} ∪ (⋃p ∈ cfg_nodes (compile_prog Pi ps). Seed p ‘
        {cs::call_string. set cs ⊆ cfg_nodes (compile_prog Pi ps) ∧ length cs ≤ k}))"
  using compiled_call_strings_finite[of Pi ps k]
    cfg_nodes_finite[OF compile_prog_finite[of Pi ps, THEN conjunct1]
      compile_prog_finite[of Pi ps, THEN conjunct2]]
  by auto
```

The monovariant policy

The same result for the monovariant `route_unit` policy, where it needs less: a context is `unit`, so the context factor is a singleton and the node bound is the entire hypothesis. Stated over `fst ‘ vars` rather than a product, since with one context there is nothing for a product to say.

```
theorem compiled_unit_vars_finite:
  fixes vars :: "(pp × unit) set"
  assumes "fst ‘ vars ⊆ cfg_nodes (compile_prog Pi ps)"
  shows "finite vars"
proof -
  have "vars ⊆ cfg_nodes (compile_prog Pi ps) × (UNIV :: unit set)"
    using assms by auto
  moreover have "finite (cfg_nodes (compile_prog Pi ps) × (UNIV :: unit set))"
    using cfg_nodes_finite compile_prog_finite by force
  ultimately show ?thesis by (rule finite_subset)
qed

end
```

Chapter 9

Voblint_Result

9.1 Source_Activation_Sound

```
theory Source_Activation_Sound
  imports "Voblint_Framework.Activation_Backbone" "Voblint_Compile.Source_To_Trace"
begin
```

End-to-end source-level activation soundness

A reachable store of a compiled source program is bounded by the abstract analysis at the stable activation context of the trace that produced it. The composition joins the stack-faithful source bridge `source_store_in_activation_collect` with `activation_collect_sound`.

The cap-consuming theorems read the abstract map through a supplied concretization γ_M instead of fixing it to `'a::numeric_domain abs_state`/γ `_`: the cap is the only thing they touch, so a reachability-lifted map fits as readily as a raw one. The backbone corollaries below read it at γ, which is what `activation_collect_sound` produces.

```
theorem source_sound_from_collecting_cap:
  fixes sg :: "pp × 'c + 'g ⇒ 'M"
    and γM :: "'M ⇒ store set"
    and adm :: "'c context_policy" and c0 :: 'c
  assumes wf: "wf_compile_input 𝒢 Pi ps"
    and s0: "s0 ∈ S"
    and run: "𝒢, Pi ⊢ (main_body Pi, s0, []) →p* (residual, s, frs)"
    and has_ctx: "⋀t. t ∈ 𝒯𝒢,compile_prog Pi ps,S
                  ⟹ ∃c. activation_context_rel 𝒢 adm c0 (compile_prog Pi ps) t c"
    and cap: "⋀v ctx. 𝒜𝒢,adm,c0,compile_prog Pi ps,S v ctx
                      ⊆ γM (sg (Inl (v, ctx)))"
  shows "∃v stk t c. Pi, compile_prog Pi ps, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
                  ∧ activation_context_rel 𝒢 adm c0 (compile_prog Pi ps) t c
                  ∧ s ∈ γM (sg (Inl (v, c)))"
proof -
  let ?g = "compile_prog Pi ps"
  from source_store_in_activation_collect
         [where adm=adm and c0=c0,
          OF wf s0 run has_ctx]
  obtain v stk t c where m:
    "Pi, compile_prog Pi ps, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)"
    and ck: "activation_context_rel 𝒢 adm c0 ?g t c"
    and mem: "s ∈ 𝒜𝒢,adm,c0,?g,S v c"
    by blast
  have "s ∈ γM (sg (Inl (v, c)))"
    using cap[of v c] mem by blast
  then show ?thesis using m ck by blast
qed
```

The witness-free specialisation of the composition at top-level program points: an empty source frame stack lands at the fixed seed context, no `activation_trace` witness exposed.

```
theorem source_sound_toplevel_from_collecting_cap:
  fixes sg :: "pp × 'c + 'g ⇒ 'M"
    and γM :: "'M ⇒ store set"
    and adm :: "'c context_policy" and c0 :: 'c
  assumes wf: "wf_compile_input 𝒢 Pi ps"
    and s0: "s0 ∈ S"
    and run: "𝒢, Pi ⊢ (main_body Pi, s0, []) →p* (residual, s, [])"
    and cap: "⋀v ctx. 𝒜𝒢,adm,c0,compile_prog Pi ps,S v ctx
                      ⊆ γM (sg (Inl (v, ctx)))"
  shows "∃v. Pi, compile_prog Pi ps, prog_main_name ⊢ (residual, s, []) ≈ (v, s, [])
           ∧ s ∈ γM (sg (Inl (v, c0)))"
proof -
  let ?g = "compile_prog Pi ps"
  from source_toplevel_in_activation_collect
         [where adm=adm and c0=c0, OF wf s0 run]
  obtain v where m: "Pi, compile_prog Pi ps, prog_main_name ⊢ (residual, s, []) ≈ (v, s, [])"
    and mem: "s ∈ 𝒜𝒢,adm,c0,?g,S v c0" by blast
  have "s ∈ γM (sg (Inl (v, c0)))" using cap[of v c0] mem by blast
  then show ?thesis using m by blast
qed
```

Backbone corollaries: discharge the five coverage obligations to build the cap

Given entry, edge, call, combine and totality obligations on the abstract map, `activation_coverage` builds the cap and the composition above yields source-level soundness. The node-collect variants restate it per CFG node.

```
theorem source_activation_sound:
  fixes sg :: "pp × 'c + 'g ⇒ 'a::numeric_domain abs_state"
    and adm :: "'c context_policy" and c₀ :: 'c
  assumes wf: "wf_compile_input 𝒢 Pi ps"
    and s0: "s0 ∈ S"
    and run: "𝒢, Pi ⊢ (main_body Pi, s0, []) →ₚ* (residual, s, frs)"
    and ENTRY_G: "⋀x. x ∈ S ⟹ x ∈ γ (sg (Inl (cfg_entry (compile_prog Pi ps), c₀)))"
    and EDGE: "⋀u a v c x x'. (u, a, v) ∈ intra (compile_prog Pi ps)
        ⟹ x ∈ γ (sg (Inl (u, c))) ⟹ x' ∈ edge_step a x
        ⟹ x' ∈ γ (sg (Inl (v, c)))"
    and CALL: "⋀u dst pars args p cont c c' x.
        (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls (compile_prog Pi ps)
        ⟹ x ∈ γ (sg (Inl (u, c)))
        ⟹ c' ∈ adm u c (call_info_of (CallEdge dst pars args) p) x
              (call_enter 𝒢 (CallEdge dst pars args) x)
        ⟹ call_enter 𝒢 (CallEdge dst pars args) x ∈ γ (sg (Inl (FunctionEntry p, c')))"
    and COMB: "⋀cl dst pars args p cont c1 c' p' x t es.
        (cl, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls (compile_prog Pi ps)
        ⟹ x ∈ γ (sg (Inl (cl, c1)))
        ⟹ admits_call_context 𝒢 (compile_prog Pi ps) adm cl c1 p' x es c'
        ⟹ t ∈ γ (sg (Inl (FunctionResult p, c')))
        ⟹ combine_collect 𝒢 dst x t ∈ γ (sg (Inl (cont, c1)))"
    and TOTAL: "call_context_total_on (λv c. γ (sg (Inl (v, c)))) adm 𝒢 (compile_prog Pi ps)"
  shows "∃v stk t c. Pi, compile_prog Pi ps, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
                  ∧ activation_context_rel 𝒢 adm c₀ (compile_prog Pi ps) t c
                  ∧ s ∈ γ (sg (Inl (v, c)))"
proof -
  interpret G: activation_coverage "compile_prog Pi ps" S "λv c. γ (sg (Inl (v, c)))" adm c₀ 𝒢
    by (standard; blast intro: ENTRY_G EDGE CALL COMB TOTAL)
  have cap: "⋀v ctx. 𝒜𝒢,adm,c₀,compile_prog Pi ps,S v ctx
                    ⊆ γ (sg (Inl (v, ctx)))"
    by (rule activation_collect_sound[OF G.activation_coverage_axioms])
  have has_ctx: "⋀t. t ∈ 𝒯𝒢,compile_prog Pi ps,S
                  ⟹ ∃c. activation_context_rel 𝒢 adm c₀ (compile_prog Pi ps) t c"
    using G.valid_activation_trace_has_context by blast
  show ?thesis
    by (rule source_sound_from_collecting_cap[where γ_M=gamma_state,
          OF wf s0 run has_ctx cap])
qed
```

Monovariant source bridge into the trace collecting

The context-forgetting specialisation projects the activation collector at trivial routing directly to `node_collect` via `activation_collect_le_node_collect`.

```
theorem source_reaches_node_collect:
  assumes wf: "wf_compile_input 𝒢 Pi ps"
    and s0: "s0 ∈ S"
    and run: "𝒢, Pi ⊢ (main_body Pi, s0, []) →ₚ* (residual, s, frs)"
  shows "∃v stk. Pi, compile_prog Pi ps, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
                ∧ s ∈ 𝒞𝒢,compile_prog Pi ps,S v"
proof -
  let ?g = "compile_prog Pi ps"
  from source_store_in_activation_collect_of_fun
         [where f = "λ_ _ _. ()" and c₀ = "()",
          OF wf s0 run]
  obtain v stk t c where m:
    "Pi, compile_prog Pi ps, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)"
    and "activation_context_of (λ_ _ _. ()) () t = c"
    and mem: "s ∈ 𝒜𝒢,context_policy_of_fun (λ_ _ _. ()),(),?g,S v c"
    by blast
  have "s ∈ 𝒞𝒢,?g,S v"
    using mem by (rule subsetD[OF activation_collect_le_node_collect])
  then show ?thesis using m by blast
qed
```

The monovariant counterpart of `source_sound_from_collecting_cap`: the cap is stated on `node_collect` alone, so there is no context index for a caller to instantiate. This is the shape a whole-program analysis result supplies, whose per-node bound has already forgotten the context.

```
theorem source_sound_from_node_collect_cap:
  fixes G :: "pp ⇒ store set"
  assumes wf: "wf_compile_input 𝒢 Pi ps"
    and s0: "s0 ∈ S"
    and run: "𝒢, Pi ⊢ (main_body Pi, s0, []) →ₚ* (residual, s, frs)"
```

```
    and cap: "⋀v. C_{G,compile_prog Pi ps,S} v ⊆ G v"
  shows "∃v stk. Pi, compile_prog Pi ps, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
                ∧ s ∈ G v"
proof -
  from source_reaches_node_collect[OF wf s0 run]
  obtain v stk where m: "Pi, compile_prog Pi ps, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)"
    and mem: "s ∈ C_{G,compile_prog Pi ps,S} v" by blast
  have "s ∈ G v" using cap[of v] mem by blast
  then show ?thesis using m by blast
qed
```

Completed runs reach a procedure result

A source run that has completed at the top level (residual `SKIP`, empty frame stack) is collected at the `FunctionResult` of the activation it completed. Two structural facts do the work, both from the CFG layer: the located node reaches its fragment exit along store-preserving intra flow (`compile_control_at_SKIP_exit_path`), and `compiled_at` supplies the fall-through `EA_Ret None p` edge from that exit into `FunctionResult p`.

The witness is kept: the completing activation is the *root* one (empty frame stack, so `caller_of` is `None`), which is what pins its procedure — see `source_completes_node_collect_exit`. Projecting into `node_collect` first would lose that: the collection at a `FunctionResult` deliberately merges every activation of that procedure.

```
theorem source_completes_valid_activation_trace_result:
  assumes wf: "wf_compile_input G Pi ps"
    and s0: "s0 ∈ S"
    and run: "G, Pi ⊢ (main_body Pi, s0, []) →_p* (SKIP (), s, [])"
  shows "∃t p. t ∈ T_{G,compile_prog Pi ps,S}
              ∧ caller_of t = None
              ∧ fst (hd (path_of t)) = cfg_entry (compile_prog Pi ps)
              ∧ sink_node t = FunctionResult p
              ∧ sink_store t = s"
proof -
  let ?g = "compile_prog Pi ps"
  from source_run_has_activation_trace[OF wf s0 run]
  obtain v stk t where sim: "Pi, ?g, prog_main_name ⊢ (SKIP (), s, []) ≈ (v, s, stk)"
    and rep: "activation_trace_repr G ?g S (v, s, stk) t" by blast
  have stk0: "stk = []" using sim by blast
  from rep stk0 have tv: "t ∈ T_{G,?g,S}" and sn: "sink_node t = v"
    and ss: "sink_store t = s" and sr: "stack_repr ?g [] t"
    by (auto simp: activation_trace_repr_def)
  have cof: "caller_of t = None" using stack_repr_Nil_iff[OF sr] by simp
  have hd_t: "fst (hd (path_of t)) = cfg_entry ?g"
    by (rule valid_activation_trace_caller_None_entry[OF tv cof])
  — the located node of a completed activation is its procedure's result
  from sim obtain p body_p k n where
    ca: "control_at Pi p body_p k n (SKIP ()) v" and cat: "compiled_at Pi ?g p body_p k n"
    by blast
  from cat obtain n' en E K where
    cc: "compile Pi p body_p k n = (n', en, E, K)" and Esub: "E ⊆ intra ?g"
    and ret: "(k, EA_Ret None p, FunctionResult p) ∈ intra ?g"
    by blast
  — the whole extension is intra flow, so it stays inside this same (root) activation
  have path_to_ret: "intra_path ?g (sink_node t, sink_store t) (FunctionResult p, s)"
  proof -
    have a: "intra_path ?g (sink_node t, sink_store t) (k, s)"
      using compile_control_at_SKIP_exit_path[OF ca cc Esub] sn ss by simp
    have b: "intra_path ?g (k, s) (FunctionResult p, s)"
      by (rule intra_path_single[OF ret]) simp
    from a b show ?thesis by (rule star_trans)

  qed
  from valid_activation_trace_intra_path_extend[OF path_to_ret tv]
  obtain t' where t'v: "t' ∈ T_{G,?g,S}" and t'n: "sink_node t' = FunctionResult p"
    and t's: "sink_store t' = s" and t'c: "caller_of t' = caller_of t"
    and t'h: "fst (hd (path_of t')) = fst (hd (path_of t))" by blast
  show ?thesis
  proof (intro exI conjI)
    show "t' ∈ T_{G,?g,S}" by (rule t'v)
    show "caller_of t' = None" using t'c cof by simp
    show "fst (hd (path_of t')) = cfg_entry ?g" using t'h hd_t by simp
    show "sink_node t' = FunctionResult p" by (rule t'n)
    show "sink_store t' = s" by (rule t's)
  qed
qed
```

Whole-program completion. The completing activation is the root one, so it entered at `FunctionEntry prog_main_name`; activation procedure locality (`valid_activation_trace_entry_result_eq`) then forces its result node to be `FunctionResult prog_main_name = cfg_exit`. No assumption about the *number* of declared procedures is needed: a program may call as many as it likes.

```
corollary source_completes_node_collect_exit:
  assumes wf: "wf_compile_input 𝒢 Pi ps"
    and s0: "s0 ∈ S"
    and run: "𝒢, Pi ⊢ (main_body Pi, s0, []) →ₚ* (SKIP (), s, [])"
  shows "s ∈ 𝒞𝒢,compile_prog Pi ps,S
              (cfg_exit (compile_prog Pi ps))"
proof -
  let ?g = "compile_prog Pi ps"
  from source_completes_valid_activation_trace_result[OF wf s0 run]
  obtain t p where tv: "t ∈ 𝒯𝒢,?g,S" and hd_t: "fst (hd (path_of t)) = cfg_entry ?g"
    and sn: "sink_node t = FunctionResult p" and ss: "sink_store t = s" by blast
  have "fst (hd (path_of t)) = FunctionEntry prog_main_name"
    using hd_t by (simp add: compile_prog_def Let_def split: prod.splits)
  from valid_activation_trace_entry_result_eq[OF wf tv this sn] have "prog_main_name = p" .
  with sn ss tv show ?thesis
    using node_collect_I by fastforce
qed

end
```

9.2 DG_Result_Construction

```
theory DG_Result_Construction
  imports
    "Voblint_Framework.DG_Analysis_Adapter"
    "Voblint_VIMP.VIMP_Program"
    "Voblint_Exec.Exec_DG_State"
begin
```

What a solved D/G system publishes

Every analysis, at every context policy and every solver discipline, turns the solver's answer – a covered key set and a map from unknowns to `dg_state`s over its carrier – into an `solved_table` table of the locals. This theory states that construction once, over an arbitrary solved pair, so an analysis's result table is one application rather than a rewritten body.

Publishing a local unknown means two steps in sequence. `canonicalize_lift` collapses a stored `Lifted` payload that describes no store to `Bot`, so a dead point reads as dead; the publication map `rd` then turns the carrier into the value the table publishes. Coverage is separate from deadness: a key the solver never visited is absent from the table, and `lookup_table` answers `Bot` for it without any claim about the program.

`sol` is the already-solved pair, not the solve function. Passing the pair keeps one solve per table in generated code – the argument is evaluated once and the per-point closure captures it – which is why no analysis needs a separate `[code]` equation with an explicit `let`.

What a point publishes is read off its local unknown together with the solved global environment, each name read at its key `Inr (gk n)`: `rc` recombines the local half with that environment into the state the point describes. An analysis that keeps every variable in the local half passes a recombination that ignores the global.

```
definition dg_result_for ::
    "('s ⇒ 'v) ⇒ ('s ⇒ bool) ⇒ ('s lifted ⇒ ('n ⇒ 'g) ⇒ 's lifted) ⇒ ('n ⇒ 'k)
     ⇒ (pp × 'c) set × (pp × 'c + 'k ⇒ ('s lifted, 'g) dg_state)
     ⇒ ('c, 'v) solved_table" where
  "dg_result_for rd emp rc gk sol =
     Solved_Table (fst sol)
       (λv ctx. map_lift rd (canonicalize_lift emp
          (rc (dg_local (snd sol (Inl (v, ctx)))) (genv gk (snd sol)))))"

lemma covered_keys_dg_result_for [simp]:
  "covered_keys (dg_result_for rd emp rc gk sol) = fst sol"
  unfolding dg_result_for_def by simp
```

The one fact every soundness bridge needs about the table: a covered key reads back the normalized recombined state, an uncovered one answers `Bot`.

```
lemma lookup_table_dg_result_for [simp]:
  "lookup_table (dg_result_for rd emp rc gk sol) v ctx
     = (if (v, ctx) ∈ fst sol
        then map_lift rd (canonicalize_lift emp
               (rc (dg_local (snd sol (Inl (v, ctx)))) (genv gk (snd sol))))
        else Bot)"
  unfolding dg_result_for_def lookup_table_def by simp
```

Normalizing before publication agrees with normalizing after it whenever the two emptiness tests agree, so a soundness bridge stated after publication applies to the table built before it.

```
lemma map_lift_canonicalize_lift:
  assumes "⋀s. emp s = empty_V (rd s)"
  shows "map_lift rd (canonicalize_lift emp d) = canonicalize_lift empty_V (map_lift rd d)"
  by (cases d) (simp_all add: assms normalize_lift_def)
```

Collapsing a payload that denotes nothing to `Bot` does not change what the value denotes.

```
lemma gamma_lift_canonicalize_lift:
  assumes "⋀v. empty_V v ⟹ gm v = {}"
  shows "gamma_lift gm (canonicalize_lift empty_V x) = gamma_lift gm x"
  by (cases x) (simp_all add: normalize_lift_def assms)

end
```

9.3 Analysis_Surface

```
theory Analysis_Surface
  imports
    "Voblint_Framework.Solved_Table"
    "Voblint_Framework.Check_Report"
    "Voblint_Compile.Compile_Invariants"
begin
```

The surface a solved table publishes

What every domain, at every solver discipline and context policy, ends up publishing about a whole program: the state its solved table (an `solved_table`) holds at a point under a context, and the check report read off those states. `analysis_surface` states both once, generically in the domain `'a` and the context type `'c`, so a domain's public API is one interpretation rather than a rewritten copy per domain, solver discipline and policy. A context-insensitive run reads it at the one context `()`.

An interpretation publishes the complete surface for a solved table. Solver disciplines therefore share one construction and cannot accidentally expose only a subset of the result and report operations.

$\bot$ at an `Bot` point is the reading every existing report function already makes: nothing reaches the point, so no store does, and $\bot$ is the abstract state whose concretisation is empty.

`bot_state` is a parameter rather than the `order_bot` class operation, and that is a code-generation requirement, not a stylistic choice. A sort constraint here would make the surface's code equation polymorphic in `'a::order_bot`, and generating code for it at an abstract state — a function type `String.literal ⇒ 'b` — would demand an executable $\bot$ arity for that function type, which needs the domain to be `enum`. `String.literal` is not. Taking the element as a parameter keeps the equation free of sorts; every interpretation still passes $\bot$, where the concrete instance is executable exactly as it was before.

```
locale analysis_surface =
  fixes table :: "imp_prog ⇒ ('c, 'a) solved_table"
    and bot_state :: 'a
    and classify :: "exp ⇒ 'a ⇒ check_result"
begin

definition state_at :: "imp_prog ⇒ 'c ⇒ pp ⇒ 'a" where
  "state_at p ctx v =
     (case lookup_table (table p) v ctx of Bot ⇒ bot_state | Lifted st ⇒ st)"

definition report :: "imp_prog ⇒ 'c ⇒ check_report_entry list" where
  "report p ctx = classify_checks (prog_cfg p) (state_at p ctx) classify"
```

The same table read with its `Bot` entries kept visible: `True` marks a point whose entry in `table` is `Bot`, which a report consumer suppresses instead of printing the bottom state's vacuous verdict. Exact for the same reason `state_at` is – it is the `lookup_table` case split itself, not a second test on the raw unknown (`fst_reach_state_at` states exactly that).

The flag is a fact about the table and nothing more. This locale assumes nothing whatever about `table`, so it cannot and does not say that such a point is one no solver covered, or one no concrete run reaches. A caller wanting either reading owes it from its own soundness theorem about the table it supplies.

```
definition reach_state_at :: "imp_prog ⇒ 'c ⇒ pp ⇒ bool × 'a" where
  "reach_state_at p ctx v =
     (case lookup_table (table p) v ctx of
        Bot ⇒ (True, bot_state) | Lifted st ⇒ (False, st))"
```

The two readings agree on the state: `reach_state_at` adds a column and changes no verdict, so a report that shows reachability classifies exactly what `report` does. Without this the two definitions are only visibly similar, and every caller reproves it by unfolding both.

```
lemma state_at_Bot [simp]:
  "lookup_table (table p) v ctx = Bot ⟹ state_at p ctx v = bot_state"
  by (simp add: state_at_def)

lemma state_at_Lifted [simp]:
  "lookup_table (table p) v ctx = Lifted st ⟹ state_at p ctx v = st"
  by (simp add: state_at_def)

lemma reach_state_at_Bot [simp]:
  "lookup_table (table p) v ctx = Bot ⟹ reach_state_at p ctx v = (True, bot_state)"
  by (simp add: reach_state_at_def)
```

```
lemma reach_state_at_Lifted [simp]:
  "lookup_table (table p) v ctx = Lifted st ⟹ reach_state_at p ctx v = (False, st)"
  by (simp add: reach_state_at_def)

lemma snd_reach_state_at [simp]:
  "snd (reach_state_at p ctx v) = state_at p ctx v"
  by (simp add: reach_state_at_def state_at_def split: lifted.splits)

lemma fst_reach_state_at [simp]:
  "fst (reach_state_at p ctx v) = (lookup_table (table p) v ctx = Bot)"
  by (simp add: reach_state_at_def split: lifted.splits)

definition report_with_state ::
  "imp_prog ⇒ 'c ⇒ (pp × exp × check_result × bool × 'a) list" where
  "report_with_state p ctx =
     classify_checks_with_state (prog_cfg p) (reach_state_at p ctx)
       (λc (_, s). classify c s)"

end
```

Register the generic surface equations outside the locale so code generation can call them directly with the table, bottom state and classifier as parameters.

```
declare
  analysis_surface.state_at_def [code]
  analysis_surface.report_def [code]
  analysis_surface.reach_state_at_def [code]
  analysis_surface.report_with_state_def [code]

end
```

9.4 DG_Analysis

```
theory DG_Analysis
  imports
    DG_Result_Construction
    Analysis_Surface
    "Voblint_Framework.Routed_Analysis_Sound"
    "Voblint_Exec.Routed_Exec_Refinement"
    Source_Activation_Sound
    "Voblint_Routing.Compiled_Routed_Equations"
    "Voblint_Routing.Entry_State_Routed_Context"
    "Voblint_Solver.TD_Solver_Bridge"
    "Voblint_Framework.DG_Keyed_Split_Spec"
begin
```

One context-sensitive analysis, assembled from its choices

A concrete analysis picks a domain implementation, an initial state, a way of telling activations apart, a solver and a classifier; everything between those choices and the published table is the same work at every domain and at every context policy. This theory does that work once.

The construction half is `dg_pipeline`, which carries no correctness assumptions, so its defining equations are unconditional and can be declared to the code generator. The correctness half is `dg_analysis`, which adds the domain and solver contracts and derives, for one program, the routed soundness statement every policy shares.

Vocabulary. A *routed* equation system publishes a callee's entry state at a global seed key instead of reading the callee entry directly; a *context* is the value the routing function attaches to a program point so that one point can be solved at several unknowns; *covered* keys are the unknowns the solver actually visited, which is not the same as the unknowns holding a bottom state.

What this covers, and what it does not. It is the assembly for executable non-relational analyses whose local unknown carries a whole reachability-lifted store and whose entry operation answers with a pure list of alternatives. The context type, the global-key type, the seed constructor, the routing function and the root context are all parameters; the carrier and the entry shape are not.

The formals route, once for every domain

Keying a callee on the abstract values its formals hold on entry, at the executable carrier. `formals_route_lifted_gen` is the same decision one layer up, on the state a solved table hands out; the equation below is what lets a routed instance discharge its routing-agreement obligation without a domain-specific lemma. The routed generator enters the callee frame before it routes, so the route itself only projects the formals out of the state it is handed.

```
definition exec_formals_route ::
    "(vname ⇒ bool) ⇒ pp ⇒ 'a list
       ⇒ ('a::bot) default_st lifted ⇒ call_action ⇒ 'a list" where
  "exec_formals_route 𝒢 u ctx d ca =
     (case ca of CallEdge dst pars args ⇒
        formals_context pars
          (ϱ𝒢 (case d of Bot ⇒ bot | Lifted d0 ⇒ d0)))"
```

The same routing decision taken from a solved *result* instead of from the solver's carrier. A rendering that draws one node per activation has to know which callee context a call edge enters, and the table it draws from holds `'a abs_state` rather than `'a default_st` — so the projection `exec_formals_route` performs is unavailable to it, and the entered state has to be recomputed from the caller's. `enter` is the domain's own entry transfer and is the only domain-specific value involved; emptiness is `is_empty`, the `executable_domain` operation every domain carries.

The witness search runs over the formals rather than over the whole state, and it has to: `is_empty_state` quantifies over every `vname`, so its code equation would demand an `enum` instance for `String.literal` and this constant would not generate code at all. Restricting to the formals is also exact rather than merely cheaper — procedure entry resets every non-global variable to top and leaves every global at the caller's own value, so no name outside the formals can witness emptiness that the caller did not already have.

`None` means the call routes nowhere, because the entered state represents no concrete store. It is not a sentinel context: an empty formal list is a legitimate context in its own right — a zero-formal callee's own entry is genuinely keyed at `[]` — so no `'a list` value could carry that meaning without colliding with a real one.

```
definition callee_ctx_of ::
    "((vname ⇒ bool) ⇒ vname list ⇒ exp list ⇒ 'a abs_state ⇒ 'a abs_state)
       ⇒ (vname ⇒ bool) ⇒ call_action ⇒ ('a::executable_domain) abs_state
       ⇒ 'a list option" where
  "callee_ctx_of enter 𝒢 ca st =
```

```
    (case ca of CallEdge dst pars args ⇒
       (let entered = enter 𝒢 pars args st
        in if list_ex (λx. is_empty (entered x)) pars then None
           else Some (formals_context pars entered)))"

lemma exec_formals_route_commute:
  "formals_route_lifted_gen u ctx (ϱ_𝒢 d) ca
     = exec_formals_route 𝒢 u ctx d ca"
  by (cases d; cases ca)
     (simp_all add: formals_route_lifted_gen_def formals_route_lifted_def
        exec_formals_route_def default_st_to_fun_def)
```

The construction

The pipeline runs one component, `comp` $\mathcal{G}$ `p`: a single analysis, or the combination of several. `init_st` is the unlifted entry state; the pipeline lifts it, because a reachability-lifted carrier is what the routed generator solves over. `empty p` is the carrier's executable emptiness test and `rd` $\mathcal{G}$ reads a carrier state back as the value the result table publishes. Nothing here fixes what the carrier or the published value is. `bot_state` is a parameter rather than the `order_bot` operation for the same code-generation reason `analysis_surface` states: a sort constraint here would demand an executable $\bot$ at a function type.

Everything one solve publishes, read back as values: the result table, the solved value of each global name, the seed a call published at a procedure entry under a context, what one edge's local step makes of a point's solved state, and where a call leads when it contributes at all.

```
record ('c, 'n, 'v) solved_run =
  run_table :: "('c, 'v) solved_table"
  run_shared :: "'n ⇒ 'v lifted"
  run_seed :: "pname ⇒ 'c ⇒ 'v lifted"
  run_step :: "pp ⇒ 'c ⇒ edge_action ⇒ 'v lifted"
  run_succ :: "cfg_node ⇒ 'c ⇒ call_action ⇒ pname ⇒ 'c option"
```

`dg_pipeline` only fixes parameters: the component `comp`, the carrier operations `emp`, `rd` and `init_st`, the key a node buffers its own global part at and the key `global_of n` each global name lives at, the context policy `seed`/`route`/ `root_ctx`, the solver `solve` with its domain, the check classifier, and the placement of program state between the two halves of a solver unknown. The soundness assumptions live in the locales built on it.

The placement is six operations. `place_spec` builds the D/G specification the equations run, from the component. `place_cmb` recombines a local value with the solved global environment into the state a point describes, and `place_rl` and `place_rg` project a state onto the part each half keeps. `place_inits` lists the initial value of each global name, which the program entry publishes at that name's key `global_of n`. `place_enter` is the caller state a call enters from: it may describe more than `place_cmb`, so a call reads only the globals it needs. An analysis that keeps all of its state in the local half builds the specification with `dg_spec_of`, recombines by ignoring the environment, keeps every state locally, publishes `Bot`, lists no initial values and enters from the caller's state as it is.

```
locale dg_pipeline =
  fixes comp :: "(vname ⇒ bool) ⇒ imp_prog ⇒ 's::semilattice_sup lifted local_spec"
    and emp :: "imp_prog ⇒ 's ⇒ bool"
    and rd :: "(vname ⇒ bool) ⇒ 's ⇒ 'v"
    and init_st :: 's
    and buffer_key :: 'k
    and global_of :: "'n ⇒ 'k"
    and seed :: "pp ⇒ 'c ⇒ 'k"
    and route :: "(vname ⇒ bool) ⇒ pp ⇒ 'c ⇒ 's lifted ⇒ call_action ⇒ 'c"
    and root_ctx :: 'c
    and solve :: "(pp × 'c, 'k, ('s lifted, 's lifted) dg_state) eqsT
                  ⇒ pp × 'c
                  ⇒ (pp × 'c) set × (pp × 'c + 'k ⇒ ('s lifted, 's lifted) dg_state)"
    and solve_dom :: "(pp × 'c, 'k, ('s lifted, 's lifted) dg_state) eqsT ⇒ pp × 'c ⇒ bool"
    and bot_state :: 'v
    and classify :: "exp ⇒ 'v ⇒ check_result"
    and place_spec :: "imp_prog ⇒ 's lifted local_spec
                       ⇒ (pp × 'c, 'k, 'n, 's lifted, 's lifted) dg_spec"
    and place_cmb :: "imp_prog ⇒ 's lifted ⇒ ('n ⇒ 's lifted) ⇒ 's lifted"
    and place_rl :: "(vname ⇒ bool) ⇒ 's lifted ⇒ 's lifted"
    and place_rg :: "(vname ⇒ bool) ⇒ 's lifted ⇒ 's lifted"
    and place_inits :: "(vname ⇒ bool) ⇒ imp_prog ⇒ ('n × 's lifted) list"
    and place_enter :: "imp_prog ⇒ call_info ⇒ 's lifted ⇒ ('n ⇒ 's lifted) ⇒ 's lifted"
begin

definition analysis_spec :: "(vname ⇒ bool) ⇒ imp_prog
    ⇒ (pp × 'c, 'k, 'n, 's lifted, 's lifted) dg_spec" where
```

```
  "analysis_spec 𝒢 p = place_spec p (comp 𝒢 p)"
```

The state the component enters a callee with, from a whole caller state. The pipeline supports components whose entry answers one alternative, the caller's state paired with this one.

```
definition comp_entry :: "(vname ⇒ bool) ⇒ imp_prog ⇒ call_info ⇒ 's lifted
    ⇒ 's lifted" where
  "comp_entry 𝒢 p ci w = snd (hd (ls_enter (comp 𝒢 p) (ls_channel (comp 𝒢 p) w) ci (w, w)))"
```

The alternative a call enters its callee with, as the solver stores it: the component enters from the caller state `place_enter` recombines, and each half of the answer keeps its local part.

```
definition entry_alt :: "(vname ⇒ bool) ⇒ imp_prog ⇒ call_info ⇒ 's lifted
    ⇒ ('n ⇒ 's lifted) ⇒ 's lifted × 's lifted" where
  "entry_alt 𝒢 p ci d g =
     (let w = place_enter p ci d g in (place_rl 𝒢 w, place_rl 𝒢 (comp_entry 𝒢 p ci w)))"
```

The state a call enters its callee with.

```
definition entry_of :: "(vname ⇒ bool) ⇒ imp_prog ⇒ call_info ⇒ 's lifted
    ⇒ ('n ⇒ 's lifted) ⇒ 's lifted"
  where "entry_of 𝒢 p ci d g = snd (entry_alt 𝒢 p ci d g)"
```

The unknown the solver is asked for. It is the program exit at the root context: a demand-driven solver is started at the answer a caller wants and explores backwards from it, so the root query is the exit and the entry point is reached as one of its dependencies.

```
definition root_query :: "imp_prog ⇒ pp × 'c" where
  "root_query p = (cfg_exit (prog_cfg p), root_ctx)"
```

`root_query`'s own type mentions no domain, so a code equation for it carries a sort hypothesis the code generator cannot see and silently rejects. The two constants that use it inline it instead: their types do mention the domain, so their equations are accepted, and `root_query` never has to be executable.

The initial value of each global name is a side effect like any other: the program entry publishes it, once, at the name's key, as a run's initialization happens before its first statement. A placement that keeps its initial global part at the node's own key passes no such values.

```
definition init_publications :: "(vname ⇒ bool) ⇒ imp_prog
    ⇒ pp × 'c ⇒ ('k × ('s lifted, 's lifted) dg_state) list" where
  "init_publications 𝒢 p x =
     (if x = (cfg_entry (prog_cfg p), root_ctx)
      then map (λ(n, d). (global_of n, DG Bot d)) (place_inits 𝒢 p) else [])"
```

The generated code evaluates this test at every equation evaluation, so the code equation names the entry node directly instead of compiling the program each time.

```
lemma init_publications_code:
  "init_publications 𝒢 p x =
     (if x = (FunctionEntry prog_main_name, root_ctx)
      then map (λ(n, d). (global_of n, DG Bot d)) (place_inits 𝒢 p) else [])"
  by (simp add: init_publications_def prog_cfg_def)
```

The initial global environment a run starts from: what the entry node buffers at its own key, read at every name living there, together with the root's initial publications.

```
definition init_env :: "(vname ⇒ bool) ⇒ imp_prog ⇒ 'n ⇒ 's lifted" where
  "init_env 𝒢 p n =
     (if global_of n = buffer_key then place_rg 𝒢 (Lifted init_st) else Bot)
       ⊔ (⨆(m, d)←filter (λ(m, d). m = n) (place_inits 𝒢 p). d)"

definition equations :: "(vname ⇒ bool) ⇒ imp_prog
    ⇒ (pp × 'c, 'k,
         ('s lifted, 's lifted) dg_state) eqsT" where
  "equations 𝒢 p =
     with_init (init_publications 𝒢 p)
       (compiled_routed_eqs_for buffer_key global_of seed (route 𝒢)
          (analysis_spec 𝒢 p) (prog_cfg p) (Lifted init_st) (place_rg 𝒢 (Lifted init_st)))"

definition solution :: "(vname ⇒ bool) ⇒ imp_prog
    ⇒ (pp × 'c) set
        × (pp × 'c + 'k
             ⇒ ('s lifted, 's lifted) dg_state)" where
  "solution 𝒢 p = solve (equations 𝒢 p) (root_query p)"

definition terminates :: "(vname ⇒ bool) ⇒ imp_prog ⇒ bool" where
```

```
  "terminates 𝒢 p = solve_dom (equations 𝒢 p) (root_query p)"

lemma solution_code: "solution 𝒢 p = solve (equations 𝒢 p) (cfg_exit (prog_cfg p), root_ctx)"
  by (simp add: solution_def root_query_def)

lemma terminates_code:
  "terminates 𝒢 p = solve_dom (equations 𝒢 p) (cfg_exit (prog_cfg p), root_ctx)"
  by (simp add: terminates_def root_query_def)

definition sol_vars :: "(vname ⇒ bool) ⇒ imp_prog ⇒ (pp × 'c) set" where
  "sol_vars 𝒢 p = fst (solution 𝒢 p)"

definition sol_env :: "(vname ⇒ bool) ⇒ imp_prog
    ⇒ pp × 'c + 'k ⇒ ('s lifted, 's lifted) dg_state" where
  "sol_env 𝒢 p = snd (solution 𝒢 p)"

definition reader :: "(vname ⇒ bool) ⇒ imp_prog
    ⇒ pp × 'c + 'k ⇒ 's lifted" where
  "reader 𝒢 p = solved_local_reader (sol_vars 𝒢 p) (sol_env 𝒢 p)"

definition result :: "(vname ⇒ bool) ⇒ imp_prog
    ⇒ ('c, 'v) solved_table" where
  "result 𝒢 p = dg_result_for (rd 𝒢) (emp p) (place_cmb p) global_of (solution 𝒢 p)"
```

The solved global environment every recombination reads.

```
definition sol_global :: "(vname ⇒ bool) ⇒ imp_prog ⇒ 'n ⇒ 's lifted" where
  "sol_global 𝒢 p = genv global_of (sol_env 𝒢 p)"
```

Where a call leads, as a function of the call site and the caller's context alone: this route applied to the state this solve published at that site. It names no concretization, so it belongs here rather than beside the soundness endpoints – which is what lets a registration that cannot export a binder publish it by spelling this out, exactly as it publishes `result`.

```
definition ctx_succ :: "(vname ⇒ bool) ⇒ imp_prog ⇒ cfg_node ⇒ 'c
    ⇒ call_action ⇒ pname ⇒ 'c" where
  "ctx_succ 𝒢 p u ctx ca q =
     route 𝒢 u ctx (entry_of 𝒢 p (call_info_of ca q) (dg_local (sol_env 𝒢 p (Inl (u, ctx))))
                   (sol_global 𝒢 p)) ca"
```

Where a call leads when it contributes at all: `None` when the entered state is bottom, which is exactly when the routed call program drops the alternative, and otherwise the context `ctx_succ` names.

```
definition live_succ :: "(vname ⇒ bool) ⇒ imp_prog ⇒ cfg_node ⇒ 'c
    ⇒ call_action ⇒ pname ⇒ 'c option" where
  "live_succ 𝒢 p u ctx ca q =
     (if entry_of 𝒢 p (call_info_of ca q) (dg_local (sol_env 𝒢 p (Inl (u, ctx))))
         (sol_global 𝒢 p) = Bot
      then None else Some (ctx_succ 𝒢 p u ctx ca q))"
```

The values one solution publishes. The seeds are read exactly as a table entry is, so an unwritten key reads as `Bot`. The shared global is read without that canonicalization: its local half is bottom by construction, which would make every value it holds read as `Bot`.

A step is what one edge's local step makes of a point's solved state: the term that edge contributes to its target's equation, re-evaluated on the solution. A target with several incoming edges stores only their join, so this is the one place the contribution of a single edge can be read. It is the component's own step, run with the answers its handler gives on the recombined state, as the edge transfer runs it, and published with the state it describes.

The successor is `live_succ` read off the same solution, so a caller listing where every call leads does not solve the system again for each call.

The solution is an argument, so a caller holding the executable solver's answer reads it without solving again; `run` below reads the specification's solve.

```
definition solved_run_of :: "(vname ⇒ bool) ⇒ imp_prog
    ⇒ (pp × 'c) set × (pp × 'c + 'k ⇒ ('s lifted, 's lifted) dg_state)
    ⇒ ('c, 'n, 'v) solved_run" where
  "solved_run_of 𝒢 p sol =
     (let c = comp 𝒢 p;
          g = genv global_of (snd sol);
          read = (λd. map_lift (rd 𝒢) (canonicalize_lift (emp p) d))
      in ⦇ run_table = dg_result_for (rd 𝒢) (emp p) (place_cmb p) global_of sol,
           run_shared = (λn. map_lift (rd 𝒢) (g n)),
           run_seed = (λf ctx. read (place_cmb p
                         (dg_local (snd sol (Inr (seed (FunctionEntry f) ctx)))) g)),
           run_step = (λv ctx a. read (let d = place_cmb p (dg_local (snd sol (Inl (v, ctx)))) g
```

```
                                    in ⟦a⟧♯c d)),
          run_succ = (λu ctx ca q.
            let d = entry_of 𝒢 p (call_info_of ca q) (dg_local (snd sol (Inl (u, ctx)))) g
            in if d = Bot then None else Some (route 𝒢 u ctx d ca)) ⦇)"

definition run :: "(vname ⇒ bool) ⇒ imp_prog ⇒ ('c, 'n, 'v) solved_run" where
  "run 𝒢 p = solved_run_of 𝒢 p (solution 𝒢 p)"

lemma run_table_run [simp]: "run_table (run 𝒢 p) = result 𝒢 p"
  by (simp add: run_def solved_run_of_def result_def Let_def)

lemma run_succ_run: "run_succ (run 𝒢 p) = live_succ 𝒢 p"
  by (simp add: run_def solved_run_of_def live_succ_def ctx_succ_def sol_env_def sol_global_def
      Let_def fun_eq_iff)
```

The contextual publication surface: one verdict per context at a check, and the aggregate a caller prints. Both are `classify_checks_ctx` and `classify_checks_verdicts` unchanged – they are generic in the context type already – so nothing here is policy-specific beyond supplying this pipeline's own result table.

```
definition check_projection :: "(vname ⇒ bool) ⇒ imp_prog
    ⇒ (pp × exp × ('c × contextual_verdict) set) list" where
  "check_projection 𝒢 p = classify_checks_ctx (prog_cfg p) (result 𝒢 p) classify"

definition verdict_report :: "(vname ⇒ bool) ⇒ imp_prog
    ⇒ (pp × exp × contextual_verdict) list" where
  "verdict_report 𝒢 p = classify_checks_verdicts (prog_cfg p) (result 𝒢 p) classify"
```

The same table read at one context: the state published there, and the check report off those states. These are `analysis_surface`'s readings of this pipeline's result, so every policy publishes the same surface; a context-insensitive run reads it at `()`.

```
definition state_at :: "(vname ⇒ bool) ⇒ imp_prog ⇒ 'c ⇒ pp ⇒ 'v" where
  "state_at 𝒢 = analysis_surface.state_at (result 𝒢) bot_state"

definition report :: "(vname ⇒ bool) ⇒ imp_prog ⇒ 'c ⇒ check_report_entry list" where
  "report 𝒢 = analysis_surface.report (result 𝒢) bot_state classify"

definition report_with_state :: "(vname ⇒ bool) ⇒ imp_prog
    ⇒ 'c ⇒ (pp × exp × check_result × bool × 'v) list" where
  "report_with_state 𝒢 = analysis_surface.report_with_state (result 𝒢) bot_state classify"

lemma state_at_unfold:
  "state_at 𝒢 p ctx v
     = (case lookup_table (result 𝒢 p) v ctx of Bot ⇒ bot_state | Lifted st ⇒ st)"
  by (simp add: state_at_def analysis_surface.state_at_def)
end
```

Register the exported locale equations for direct calls to the generic pipeline. Attributes inside the locale apply through interpretation; code generation here also needs equations for the constants with their locale parameters left abstract.

```
declare
  dg_pipeline.analysis_spec_def [code]
  dg_pipeline.comp_entry_def [code]
  dg_pipeline.entry_alt_def [code]
  dg_pipeline.entry_of_def [code]
  dg_pipeline.sol_global_def [code]
  dg_pipeline.init_publications_code [code]
  dg_pipeline.equations_def [code]
  dg_pipeline.solution_code [code]
  dg_pipeline.terminates_code [code]
  dg_pipeline.sol_vars_def [code]
  dg_pipeline.sol_env_def [code]
  dg_pipeline.reader_def [code]
  dg_pipeline.result_def [code]
  dg_pipeline.ctx_succ_def [code]
  dg_pipeline.live_succ_def [code]
  dg_pipeline.solved_run_of_def [code]
  dg_pipeline.run_def [code]
  dg_pipeline.check_projection_def [code]
  dg_pipeline.verdict_report_def [code]
  dg_pipeline.state_at_def [code]
  dg_pipeline.report_def [code]
  dg_pipeline.report_with_state_def [code]
```

The contracts

What an instance owes, and nothing more: its component is sound for the concretization its publication map induces, its entry answers one alternative, its two emptiness tests are exact, its seed keys are distinct from the analysis-wide global, its solver is a `certified_solver` (a post-solution over finitely many keys once it terminates), its classifier is correct, and its entry state describes every initial store. The equation system, the solve, the covered keys, the reader, the result table and the report are all fixed by `dg_pipeline` above and appear here only in conclusions. Each obligation is stated at the program's own declared globals, the one set the soundness statement uses.

```
locale dg_analysis =
  dg_pipeline comp emp rd init_st buffer_key global_of seed route root_ctx solve solve_dom
    bot_state classify place_spec place_cmb place_rl place_rg place_inits place_enter
  + certified_solver solve solve_dom solve_c
  for comp :: "(vname ⇒ bool) ⇒ imp_prog ⇒ 's::semilattice_sup lifted local_spec"
    and emp :: "imp_prog ⇒ 's ⇒ bool"
    and rd :: "(vname ⇒ bool) ⇒ 's ⇒ 'v"
    and init_st :: 's
    and buffer_key :: 'k
    and global_of :: "'n ⇒ 'k"
    and seed :: "pp ⇒ 'c ⇒ 'k"
    and route :: "(vname ⇒ bool) ⇒ pp ⇒ 'c ⇒ 's lifted ⇒ call_action ⇒ 'c"
    and root_ctx :: 'c
    and solve solve_dom
    and bot_state :: 'v
    and classify :: "exp ⇒ 'v ⇒ check_result"
    and γV :: "'v ⇒ store set"
    and emptyV :: "'v ⇒ bool"
    and solve_c :: "(pp × 'c, 'k, ('s lifted, 's lifted) dg_state) eqsT
                    ⇒ pp × 'c
                    ⇒ ((pp × 'c) set
                          × (pp × 'c + 'k ⇒ ('s lifted, 's lifted) dg_state)) option"
    and place_spec :: "imp_prog ⇒ 's lifted local_spec
                         ⇒ (pp × 'c, 'k, 'n, 's lifted, 's lifted) dg_spec"
    and place_cmb :: "imp_prog ⇒ 's lifted ⇒ ('n ⇒ 's lifted) ⇒ 's lifted"
    and place_rl :: "(vname ⇒ bool) ⇒ 's lifted ⇒ 's lifted"
    and place_rg :: "(vname ⇒ bool) ⇒ 's lifted ⇒ 's lifted"
    and place_inits :: "(vname ⇒ bool) ⇒ imp_prog ⇒ ('n × 's lifted) list"
    and place_enter :: "imp_prog ⇒ call_info ⇒ 's lifted ⇒ ('n ⇒ 's lifted) ⇒ 's lifted" +
  assumes comp_sound:
     "⋀p. sound_local_spec (declared_global p)
              (λd. gamma_lift γV (map_lift (rd (declared_global p)) d))
              (comp (declared_global p) p)"
    and enter_single:
     "⋀p ci d. ls_enter (comp (declared_global p) p) (ls_channel (comp (declared_global p) p) d)
                   ci (d, d)
                 = [(d, comp_entry (declared_global p) p ci d)]"
    and place_contract:
     "⋀p. analysis_contract (analysis_spec (declared_global p) p)
              (λd e. gamma_lift γV
                       (map_lift (rd (declared_global p)) (place_cmb p d e)))
              (declared_global p)"
    and place_enter_runs:
     "⋀p ci d σ. ∃pub. enter_runs (enter♯ (analysis_spec (declared_global p) p) ci)
        (mk_dg_man d global_of) σ
        [entry_alt (declared_global p) p ci d (genv global_of σ)] pub"
    and place_enter_deps:
     "⋀p ci d σ. ∃deps. enter_deps (enter♯ (analysis_spec (declared_global p) p) ci)
        (mk_dg_man d global_of) σ
        [entry_alt (declared_global p) p ci d (genv global_of σ)] deps"
    and place_entry_sound:
     "⋀p ci d σ pub s.
        enter_runs (enter♯ (analysis_spec (declared_global p) p) ci)
          (mk_dg_man d global_of) σ
          [entry_alt (declared_global p) p ci d (genv global_of σ)] pub
        ⟹ genv global_of pub ≤ genv global_of σ
        ⟹ s ∈ gamma_lift γV (map_lift (rd (declared_global p))
             (place_cmb p d (genv global_of σ)))
        ⟹ s ∈ gamma_lift γV (map_lift (rd (declared_global p))
                 (place_cmb p
                    (fst (entry_alt (declared_global p) p ci d (genv global_of σ)))
                    (genv global_of σ)))
          ∧ call_enter (declared_global p) (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s
              ∈ gamma_lift γV (map_lift (rd (declared_global p))
                  (place_cmb p
```

```
                    (snd (entry_alt (declared_global p) p ci d (genv global_of σ)))
                    (genv global_of σ)))"
    and place_cmb_bot: "⋀p e. place_cmb p Bot e = Bot"
    and place_init:
      "⋀p. gamma_lift γV (map_lift (rd (declared_global p)) (Lifted init_st))
         ⊆ gamma_lift γV (map_lift (rd (declared_global p))
              (place_cmb p (Lifted init_st) (init_env (declared_global p) p)))"
    and empty_rd: "⋀p s. emp p s ⟷ emptyV (rd (declared_global p) s)"
    and emptyV_sound: "sound_emptiness emptyV γV"
    and seed_ne_buffer_key: "⋀v ctx. seed v ctx ≠ buffer_key"
    and seed_ne_global_of: "⋀v ctx n. seed v ctx ≠ global_of n"
    and classify_proved:
      "⋀c d s. classify c d = Check_Proved ⟹ s ∈ γV d ⟹ truthy (⟦c⟧e s)"
    and classify_refuted:
      "⋀c d s. classify c d = Check_Refuted ⟹ s ∈ γV d
         ⟹ ¬ truthy (⟦c⟧e s)"
    and bot_state_empty: "γV bot_state = {}"
    and init_sound:
      "⋀p. cinit_stores (declared_global p) ⊆ γV (rd (declared_global p) init_st)"
begin
```

The generic contract asks the published emptiness test only to be sound: a product of analyses may describe nothing without any one of them saying so. A particular instance may still be exact.

```
lemmas emptyV_soundD = sound_emptinessD [OF emptyV_sound]
```

Termination is a per-program side condition, and this is how a caller decides it: run the solver's own executable termination check on this program's equations. The premise stays `solve_dom` everywhere else, so nothing in the soundness argument depends on the check having been run.

```
lemma terminates_of_solve_c:
  assumes "solve_c (equations 𝒢 p) (root_query p) ≠ None"
  shows "terminates 𝒢 p"
  unfolding terminates_def by (rule dom_of_solve_c[OF assms])
```

An answer of the executable solver is the solve itself, so a run built from that answer is `run`, and its program needs no separate termination premise.

```
lemma solve_c_run:
  assumes "solve_c (equations 𝒢 p) (root_query p) = Some sol"
  shows "terminates 𝒢 p" and "solved_run_of 𝒢 p sol = run 𝒢 p"
  using assms terminates_of_solve_c solve_of_solve_c[OF assms]
  by (simp_all add: run_def solution_def)

lemma vars_finite_of_terminates:
  assumes "terminates 𝒢 p"
  shows "finite (sol_vars 𝒢 p)"
  using solve_fin[OF assms[unfolded terminates_def]]
  by (simp add: sol_vars_def solution_def)
```

The contexts a concrete call is admitted at under an entry-state policy: every context the entered state routes to. This is a top-level constant rather than something read out of a per-program interpretation, because an analysis publishes it – a caller stating a context-indexed collecting fact has to name the relation those contexts are indexed by, and an example checking a routing decision has to name it too.

With one alternative per call it is one context per abstract caller state; it is still a relation and not a function because several concrete callers sharing one abstract state reach the same context.

```
definition admitted_contexts :: "(vname ⇒ bool) ⇒ imp_prog ⇒ 'c context_policy" where
  "admitted_contexts 𝒢 p =
     routed_entry_context_rel
       (λci d. [entry_alt 𝒢 p ci d (sol_global 𝒢 p)])
       (λd e. gamma_lift γV (map_lift (rd 𝒢) (place_cmb p d e)))
       (sol_env 𝒢 p) global_of (route 𝒢)"
```

What the assembly derives, for one program

Fixing one program `p`, this part equates the published table with the solved reader, puts the solver's answer in the routed spine's shape, and states routed soundness for any relation of admitted contexts.

```
context
  fixes p :: imp_prog
begin

abbreviation (input) pgs :: "vname ⇒ bool" where "pgs ≡ declared_global p"
  — input-only, so interpreted facts print declared_global p
```

What a carrier state describes, published and concretized, and what a point's two halves describe once recombined against the solved global.

```
abbreviation cgam :: "'s lifted ⇒ store set" where
  "cgam d ≡ gamma_lift γ_V (map_lift (rd pgs) d)"

abbreviation (input) pgam :: "'s lifted ⇒ ('n ⇒ 's lifted) ⇒ store set" where
  "pgam d e ≡ cgam (place_cmb p d e)"

abbreviation (input) gsol :: "'n ⇒ 's lifted" where
  "gsol ≡ sol_global pgs p"

abbreviation entered :: "call_info ⇒ 's lifted ⇒ 's lifted" where
  "entered ci d ≡ entry_of pgs p ci d gsol"

lemma empty_rd_exact: "emp p s = empty_V (rd pgs s)"
  by (simp add: empty_rd)

lemma dg_spec_wf_analysis_spec [intro, simp]: "dg_spec_wf (analysis_spec pgs p)"
  by (rule analysis_contract.spec_wf[OF place_contract])

lemma pgam_Bot [simp]: "pgam Bot g = {}"
  by (simp add: place_cmb_bot)
```

The published table and the solved reader describe the same stores at every key. A caller states soundness against `lookup_table` of the result table, while the routed endpoints are stated against the reader; this is the equation between them, and it needs no coverage premise. At a covered key it is publication commuting with the normalization; at an uncovered one it is `Bot` against `Bot`, and both describe nothing.

```
lemma gamma_reader_eq_lookup:
  "pgam (reader pgs p (Inl (v, ctx))) gsol
     = gamma_lift γ_V (lookup_table (result pgs p) v ctx)"
proof -
  have gc: "gamma_lift γ_V (canonicalize_lift empty_V x) = gamma_lift γ_V x" for x
    by (rule gamma_lift_canonicalize_lift) (rule empty_V_soundD)
  show ?thesis
  proof (cases "(v, ctx) ∈ sol_vars pgs p")
    case True
    then show ?thesis
      by (simp add: reader_def result_def sol_vars_def sol_env_def sol_global_def gc
          map_lift_canonicalize_lift[of "emp p" "empty_V" "rd pgs", OF empty_rd_exact])
  next
    case False
    then show ?thesis by (simp add: reader_def result_def sol_vars_def bot_lifted_eq)
  qed
qed
```

The solver's answer, in the shape the routed spine consumes An analysis solves the buffered generator – a node with several intra predecessors or several returning calls publishes its analysis-wide contribution once per evaluation rather than once per contribution – while the framework states its soundness over the unbuffered one. `pp_routed` is that reconciliation at this pipeline's own equations, which the placement owes.

```
lemma pp_buffered:
  assumes solves: "terminates pgs p"
  shows "part_post_solution (equations pgs p) (root_query p)
           (sol_env pgs p) (sol_vars pgs p)"
  using solve_pp[OF solves[unfolded terminates_def]]
  by (simp add: sol_env_def sol_vars_def solution_def)

theorem pp_routed:
  assumes solves: "terminates pgs p"
  shows "part_post_solution
     (routed_node_rhs intra_predecessor_addr_list call_site_list (λ_. buffer_key) (route pgs)
        (λctx' src a. dg_spec_edge_program (analysis_spec pgs p) a src global_of)
        (routed_call_program (analysis_spec pgs p) global_of seed
           (static_resolve (prog_cfg p)) (λd. d = Bot))
        (routed_entry_seed_programs seed)
        (prog_cfg p) Bot (Lifted init_st) (place_rg pgs (Lifted init_st)))
     (root_query p) (sol_env pgs p) (sol_vars pgs p)"
  using part_post_solution_with_init[OF pp_buffered[OF solves, unfolded equations_def]]
  unfolding compiled_routed_eqs_for_def bot_lifted_eq
  by (rule pp_routed_of_buffered[OF dg_spec_wf_analysis_spec seed_ne_buffer_key])
```

The run starts below the solved environment once the solve covers the program entry: the entry node publishes the placement's initial global part at its own key, and each listed initial value at its name's key.

```
lemma init_publication_le_sol:
  assumes solves: "terminates pgs p"
    and entry_cov: "(cfg_entry (prog_cfg p), root_ctx) ∈ sol_vars pgs p"
    and mem: "(n, d) ∈ set (place_inits pgs p)"
  shows "d ≤ gsol n"
proof -
  have pp: "part_post_solution (equations pgs p) (root_query p) (sol_env pgs p) (sol_vars pgs p)"
    by (rule pp_buffered[OF solves])
  have "DG Bot d ≤ sol_env pgs p (Inr (global_of n))"
    by (rule post_bounded_with_init_at[OF post_bounded_of_part_post_solution[OF pp,
          unfolded equations_def] entry_cov])
       (use mem in ‹auto simp: init_publications_def›)
  then show ?thesis by (simp add: sol_global_def less_eq_dg_state_def)
qed

lemma foldr_init_le:
  "∀(m, d) ∈ set xs. d ≤ f m
     ⟹ (⨆(m, d)←filter (λ(m, d). m = n) xs. d) ≤ (f n :: 's lifted)"
  by (induction xs) auto

lemma init_env_le_sol:
  assumes solves: "terminates pgs p"
    and entry_cov: "(cfg_entry (prog_cfg p), root_ctx) ∈ sol_vars pgs p"
    and own: "place_rg pgs (Lifted init_st) ≤ dg_global (sol_env pgs p (Inr buffer_key))"
  shows "init_env pgs p ≤ gsol"
proof (rule le_funI)
  fix n
  have "(⨆(m, d)←filter (λ(m, d). m = n) (place_inits pgs p). d) ≤ gsol n"
    by (rule foldr_init_le) (use init_publication_le_sol[OF solves entry_cov] in blast)
  moreover have "(if global_of n = buffer_key then place_rg pgs (Lifted init_st) else Bot)
                   ≤ gsol n"
    using own by (auto simp: sol_global_def)
  ultimately show "init_env pgs p n ≤ gsol n"
    unfolding init_env_def by simp
qed
```

The one entry alternative this carrier answers with The specification's entry answers with a single alternative, the recombined caller state and the callee frame entered from it, each split back by the placement. `entry_cover` is the only fact about it any context policy needs, namely that a concrete call from a described caller lands in the described callee.

What a call publishes on entry is below the solved global, once the node the call returns to is solved: that node's equation runs the call, and a post-solution bounds what every solved equation publishes.

```
lemma enter_pub_le_sol:
  assumes solves: "terminates pgs p"
    and cont: "(k, ctx) ∈ sol_vars pgs p"
    and ce: "(u, ca, FunctionEntry q, k) ∈ calls (prog_cfg p)"
    and R: "enter_runs (enter♯ (analysis_spec pgs p) (call_info_of ca q))
              (mk_dg_man (dg_local (sol_env pgs p (Inl (u, ctx)))) global_of)
              (sol_env pgs p) pairs pub"
  shows "genv global_of pub ≤ gsol"
proof (rule le_funI)
  fix n
  let ?K = "Inr (global_of n)"
  let ?σ = "sol_env pgs p"
  let ?callee = "routed_callee_call_program (analysis_spec pgs p) global_of seed (route pgs)
                   (λd. d = Bot) ctx ca u (dg_local (?σ (Inl (u, ctx)))) q"
  let ?call = "routed_call_program (analysis_spec pgs p) global_of seed
                 (static_resolve (prog_cfg p)) (λd. d = Bot) (route pgs) ctx ca u k"
  let ?contribs = "routed_contribution_programs intra_predecessor_addr_list call_site_list (route pgs)
        (λctx' src a. dg_spec_edge_program (analysis_spec pgs p) a src global_of)
        (routed_call_program (analysis_spec pgs p) global_of seed
           (static_resolve (prog_cfg p)) (λd. d = Bot))
        (routed_entry_seed_programs seed) (prog_cfg p) ctx k"
  let ?rhs = "routed_node_rhs intra_predecessor_addr_list call_site_list (λ_. buffer_key) (route pgs)
        (λctx' src a. dg_spec_edge_program (analysis_spec pgs p) a src global_of)
        (routed_call_program (analysis_spec pgs p) global_of seed
           (static_resolve (prog_cfg p)) (λd. d = Bot))
        (routed_entry_seed_programs seed)
        (prog_cfg p) Bot (Lifted init_st) (place_rg pgs (Lifted init_st))"
  have fin: "finite (calls (prog_cfg p))"
    unfolding prog_cfg_def by (simp add: compile_prog_finite)
  have wf: "∀t ∈ set ?contribs. sp_wf t"
    by (rule routed_contribution_programs_wf) auto
```

```
  have "pub ?K ≤ sides_of_program ?callee ?σ ?K"
    unfolding routed_callee_call_program_def
    by (simp add: sp_compile_bind sp_wf_observes enter_runsD_sides[OF R] sup_fun_def)
  also have "... ≤ sides_of_program ?call ?σ ?K"
    by (rule routed_call_program_sides_ge_at[OF dg_spec_wf_analysis_spec])
       (use ce fin in simp)
  also have "... ≤ sides_of_program (side_rhs_fold_dg
      (if k = cfg_entry (prog_cfg p) then Bot ⊔ Lifted init_st else Bot) ?contribs)
      ?σ ?K"
  proof (rule sides_le_side_rhs_fold_dg[OF wf])
    have "(u, ca) ∈ set (call_site_list (prog_cfg p) k)" using ce fin by auto
    then show "?call ∈ set ?contribs" by (rule routed_contribution_programs_combineI)
  qed
  also have "... ≤ sides_of_rhs (?rhs (k, ctx)) ?σ ?K"
    unfolding routed_node_rhs_def Let_def
    by (cases "k = cfg_entry (prog_cfg p)") (auto simp: Let_def)
  also have "... ≤ ?σ ?K"
    using pp_routed[OF solves] cont by (blast dest: le_funD)
  finally show "genv global_of pub n ≤ gsol n"
    unfolding sol_global_def by (simp add: less_eq_dg_state_def)
qed

lemma entry_cover:
  assumes solves: "terminates pgs p"
    and cont: "(k, ctx) ∈ sol_vars pgs p"
    and ce: "(u, CallEdge dst pars args, FunctionEntry q, k) ∈ calls (prog_cfg p)"
    and sin: "s ∈ pgam (dg_local (sol_env pgs p (Inl (u, ctx)))) gsol"
  shows "entry_pairs_cover (λd. pgam d gsol) s (call_enter pgs (CallEdge dst pars args) s)
           [entry_alt pgs p (call_info_of (CallEdge dst pars args) q)
              (dg_local (sol_env pgs p (Inl (u, ctx)))) gsol]"
proof -
  let ?ci = "call_info_of (CallEdge dst pars args) q"
  let ?d = "dg_local (sol_env pgs p (Inl (u, ctx)))"
  let ?alt = "entry_alt pgs p ?ci ?d gsol"
  obtain pub where R: "enter_runs (enter♯ (analysis_spec pgs p) ?ci)
      (mk_dg_man ?d global_of) (sol_env pgs p)
      [entry_alt pgs p ?ci ?d (genv global_of (sol_env pgs p))] pub"
    using place_enter_runs by blast
  have le: "genv global_of pub ≤ genv global_of (sol_env pgs p)"
    using enter_pub_le_sol[OF solves cont ce R] by (simp add: sol_global_def)
  have "s ∈ pgam (fst ?alt) gsol ∧ call_enter pgs (CallEdge dst pars args) s ∈ pgam (snd ?alt) gsol"
    using place_entry_sound[OF R le sin[unfolded sol_global_def]]
    unfolding sol_global_def by (simp del: declared_global_iff)
  then show ?thesis
    by (intro entry_pairs_coverI[of "fst ?alt" "snd ?alt"]) simp_all
qed
```

The routed soundness statement, at any admitted-context relation Which contexts a concrete call is admitted at is the one thing a context policy still chooses, so `adm` is a parameter here rather than a derived object. What the policy owes about it is exactly two facts: a context `adm` admits is the one this pipeline's route computes on the entered state, at a covered unknown (`cover_R`); and `adm` admits at least one context for every concrete call at a covered call site (`total_R`). Everything else below is fixed by the pipeline.

The specification is sound for the recombined concretization; the placement owes this, so the routed statement below never re-derives it.

```
interpretation dg_base: analysis_contract "analysis_spec pgs p" "λd e. pgam d e" pgs
  by (rule place_contract)

lemma routed_analysis_sound_of_live:
  fixes adm :: "'c context_policy"
  assumes solves: "terminates pgs p"
    and fwd_ok: "⋀u a v ctx. (u, ctx) ∈ sol_vars pgs p
        ⟹ dg_local (sol_env pgs p (Inl (u, ctx))) ≠ Bot
        ⟹ (u, a, v) ∈ intra (prog_cfg p) ⟹ (v, ctx) ∈ sol_vars pgs p"
    and comb_fwd_ok: "⋀cl c1 dst pars args q cont. (cl, c1) ∈ sol_vars pgs p
        ⟹ (cl, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ (cont, c1) ∈ sol_vars pgs p"
    and cover_R: "⋀u ctx dst pars args q cont s ctx'.
        (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ ctx' ∈ adm u ctx (call_info_of (CallEdge dst pars args) q) s
              (call_enter pgs (CallEdge dst pars args) s)
        ⟹ entered (call_info_of (CallEdge dst pars args) q)
              (dg_local (sol_env pgs p (Inl (u, ctx)))) ≠ Bot
```

```
        ⟹ route pgs u ctx
                (entered (call_info_of (CallEdge dst pars args) q)
                   (dg_local (sol_env pgs p (Inl (u, ctx)))))
                (CallEdge dst pars args) = ctx'
            ∧ (FunctionEntry q, ctx') ∈ sol_vars pgs p"
    and total_R: "⋀u ctx dst pars args q cont s.
        (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ s ∈ pgam (dg_local (sol_env pgs p (Inl (u, ctx)))) gsol
        ⟹ adm u ctx (call_info_of (CallEdge dst pars args) q) s
                     (call_enter pgs (CallEdge dst pars args) s) ≠ {}"
  shows "routed_analysis (analysis_spec pgs p) (λd e. pgam d e) pgs (prog_cfg p)
     buffer_key global_of
     (route pgs) Bot (Lifted init_st) (place_rg pgs (Lifted init_st)) (sol_env pgs p)
     (sol_vars pgs p) (root_query p)
     seed (λd. d = Bot) adm (λd e. map_lift (rd pgs) (place_cmb p d e)) γ_V empty_V
     classify"
proof (unfold_locales, goal_cases CmbWf ExtraWf FinE PP SgCov SgUncov Fwd FinC CallsUnique
    SeedUnknown SeedNeGlobal IsBotBot IsBotSound ResolveSound EnterCover EnterTotal CombFwd GammaRd
    EmptyExact ClProved ClRefuted VarsFin)
  case CmbWf show ?case by (rule sp_wf_routed_call_program[OF dg_spec_wf_analysis_spec])
next
  case ExtraWf then show ?case by (rule sp_wf_routed_entry_seed_programs)
next
  case FinE show ?case unfolding prog_cfg_def by (simp add: compile_prog_finite)
next
  case PP show ?case by (rule post_bounded_of_part_post_solution[OF pp_routed[OF solves]])
next
  case (SgCov v c) then show ?case by simp
next
  case (SgUncov v c) then show ?case by (simp add: bot_lifted_eq)
next
  case (Fwd u a v c)
  have "dg_local (sol_env pgs p (Inl (u, c))) ≠ Bot"
    using Fwd(2) by auto
  with Fwd(1,3) show ?case by (blast intro: fwd_ok)
next
  case FinC show ?case unfolding prog_cfg_def by (simp add: compile_prog_finite)
next
  case CallsUnique show ?case
    unfolding calls_source_unique_def prog_cfg_def
    using compile_prog_calls_source_unique by blast
next
  case (SeedUnknown q ctx) show ?case by (rule seed_ne_buffer_key)
next
  case (SeedNeGlobal q ctx v) show ?case by (rule seed_ne_global_of)
next
  case IsBotBot show ?case by simp
next
  case (IsBotSound d g') then show ?case by simp
next
  case (ResolveSound u ctx dst pars args q cont s)
  then show ?case unfolding prog_cfg_def by (simp add: compile_prog_finite)
next
  case (EnterCover u ctx dst pars args q cont s ctx')
  let ?ci = "call_info_of (CallEdge dst pars args) q"
  let ?caller = "dg_local (sol_env pgs p (Inl (u, ctx)))"
  let ?g = "genv global_of (sol_env pgs p)"
  have sin: "s ∈ pgam ?caller gsol"
    using EnterCover(3) by (simp add: sol_global_def)
  have cov: "entry_pairs_cover (λd. pgam d gsol) s
     (call_enter pgs (CallEdge dst pars args) s) [entry_alt pgs p ?ci ?caller gsol]"
    by (rule entry_cover[OF solves comb_fwd_ok[OF EnterCover(1,2)] EnterCover(2) sin])
  obtain c e where ce: "entry_alt pgs p ?ci ?caller gsol = (c, e)"
    by (cases "entry_alt pgs p ?ci ?caller gsol")
  have nbE: "entered ?ci ?caller ≠ Bot"
  proof
    assume "entered ?ci ?caller = Bot"
    with cov ce show False by (simp add: entry_pairs_cover_def entry_of_def place_cmb_bot)
  qed
  have req: "route pgs u ctx (entered ?ci ?caller) (CallEdge dst pars args) = ctx'"
    and covE: "(FunctionEntry q, ctx') ∈ sol_vars pgs p"
    using cover_R[OF EnterCover(1,2,4) nbE] by blast+
  obtain pub where runs: "enter_runs (enter♯ (analysis_spec pgs p) ?ci)
     (mk_dg_man ?caller global_of) (sol_env pgs p)
     [entry_alt pgs p ?ci ?caller ?g] pub"
```

```
    using place_enter_runs by blast
  obtain deps where deps: "enter_deps (enter♯ (analysis_spec pgs p) ?ci)
      (mk_dg_man ?caller global_of) (sol_env pgs p)
      [entry_alt pgs p ?ci ?caller ?g] deps"
    using place_enter_deps by blast
  have ce': "entry_alt pgs p ?ci ?caller ?g = (c, e)"
    using ce by (simp add: sol_global_def)
  have sc: "s ∈ pgam c ?g" and ec: "call_enter pgs (CallEdge dst pars args) s ∈ pgam e ?g"
    using cov ce by (simp_all add: entry_pairs_cover_def sol_global_def)
  have re: "route pgs u ctx e (CallEdge dst pars args) = ctx'"
    using req ce by (simp add: entry_of_def)
  show ?case
    using runs deps sc ec re covE unfolding ce' by fastforce
next
  case (EnterTotal u ctx dst pars args q cont s)
  then show ?case by (intro total_R) (simp_all add: sol_global_def)
next
  case (CombFwd cl c1 dst pars args q cont)
  then show ?case by (rule comb_fwd_ok)
next
  case (GammaRd d g') show ?case by simp
next
  case (EmptyExact v) then show ?case by (rule empty_V_soundD)
next
  case (ClProved c d s) then show ?case by (rule classify_proved)
next
  case (ClRefuted c d s) then show ?case by (rule classify_refuted)
next
  case VarsFin show ?case by (rule vars_finite_of_terminates[OF solves])
qed

lemma routed_analysis_sound_of:
  fixes adm :: "'c context_policy"
  assumes solves: "terminates pgs p"
    and fwd_ok: "⋀u a v ctx. (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, a, v) ∈ intra (prog_cfg p) ⟹ (v, ctx) ∈ sol_vars pgs p"
    and comb_fwd_ok: "⋀cl c1 dst pars args q cont. (cl, c1) ∈ sol_vars pgs p
        ⟹ (cl, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ (cont, c1) ∈ sol_vars pgs p"
    and cover_R: "⋀u ctx dst pars args q cont s ctx'.
        (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ ctx' ∈ adm u ctx (call_info_of (CallEdge dst pars args) q) s
              (call_enter pgs (CallEdge dst pars args) s)
        ⟹ route pgs u ctx
                (entered (call_info_of (CallEdge dst pars args) q)
                   (dg_local (sol_env pgs p (Inl (u, ctx)))))
                (CallEdge dst pars args) = ctx'
            ∧ (FunctionEntry q, ctx') ∈ sol_vars pgs p"
    and total_R: "⋀u ctx dst pars args q cont s.
        (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ s ∈ pgam (dg_local (sol_env pgs p (Inl (u, ctx)))) gsol
        ⟹ adm u ctx (call_info_of (CallEdge dst pars args) q) s
                     (call_enter pgs (CallEdge dst pars args) s) ≠ {}"
  shows "routed_analysis (analysis_spec pgs p) (λd e. pgam d e) pgs (prog_cfg p)
     buffer_key global_of
     (route pgs) Bot (Lifted init_st) (place_rg pgs (Lifted init_st)) (sol_env pgs p)
     (sol_vars pgs p) (root_query p)
     seed (λd. d = Bot) adm (λd e. map_lift (rd pgs) (place_cmb p d e)) γ_V empty_V
     classify"
  by (rule routed_analysis_sound_of_live [where adm = adm, OF solves _ comb_fwd_ok _ total_R])
     (blast intro: fwd_ok dest: cover_R)+
```

The published endpoint, under termination and coverage Under termination and closure of the solved keys, `activation_collect_sound_of` bounds every admitted activation by the solved table's entry at its context.

```
lemma cinit_le_init: "cinit_stores pgs ⊆ pgam (Lifted init_st) (init_env pgs p)"
  using init_sound[of p] place_init[of p] by auto
```

Every activation the activation-trace semantics admits at a context is described by the solved table's entry for that context. The coverage premises are the shape a solver's own reachable set supplies: an edge out of an unknown the solve visited lands on one it also visited.

```
lemma activation_collect_sound_of:
  fixes adm :: "'c context_policy"
```

```
  assumes solves: "terminates pgs p"
    and entry_cov: "(cfg_entry (prog_cfg p), root_ctx) ∈ sol_vars pgs p"
    and fwd_ok: "⋀u a v ctx. (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, a, v) ∈ intra (prog_cfg p) ⟹ (v, ctx) ∈ sol_vars pgs p"
    and comb_fwd_ok: "⋀cl c1 dst pars args q cont. (cl, c1) ∈ sol_vars pgs p
        ⟹ (cl, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ (cont, c1) ∈ sol_vars pgs p"
    and cover_R: "⋀u ctx dst pars args q cont s ctx'.
        (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ ctx' ∈ adm u ctx (call_info_of (CallEdge dst pars args) q) s
             (call_enter pgs (CallEdge dst pars args) s)
        ⟹ route pgs u ctx
             (entered (call_info_of (CallEdge dst pars args) q)
                (dg_local (sol_env pgs p (Inl (u, ctx)))))
             (CallEdge dst pars args) = ctx'
           ∧ (FunctionEntry q, ctx') ∈ sol_vars pgs p"
    and total_R: "⋀u ctx dst pars args q cont s.
        (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ s ∈ pgam (dg_local (sol_env pgs p (Inl (u, ctx)))) gsol
        ⟹ adm u ctx (call_info_of (CallEdge dst pars args) q) s
                     (call_enter pgs (CallEdge dst pars args) s) ≠ {}"
  shows "A_pgs,adm,root_ctx,prog_cfg p,cinit_stores pgs v ctx
          ⊆ pgam (reader pgs p (Inl (v, ctx))) gsol"
proof -
  interpret adapter: routed_analysis "analysis_spec pgs p" "λd e. pgam d e" pgs
      "prog_cfg p" buffer_key global_of "route pgs" Bot "Lifted init_st"
      "place_rg pgs (Lifted init_st)" "sol_env pgs p" "sol_vars pgs p" "root_query p" seed
      "λd. d = Bot" adm "λd e. map_lift (rd pgs) (place_cmb p d e)" γ_V empty_V classify
    by (rule routed_analysis_sound_of
          [where adm = adm, OF solves fwd_ok comb_fwd_ok cover_R total_R])
  have init_env_le: "init_env pgs p ≤ genv global_of (sol_env pgs p)"
    using init_env_le_sol[OF solves entry_cov adapter.pp_entry_init_global_bound[OF entry_cov]]
    by (simp add: sol_global_def)
  show ?thesis
    using adapter.activation_collect_dg_sound[OF entry_cov _ init_env_le, of "cinit_stores pgs" v
      ctx]
      cinit_le_init
    by (simp add: reader_def sol_global_def)
qed
```

The two context policies this assembly supports `entry_context_rel` is the relation an entry-state policy induces: a concrete call is admitted at every context some covering alternative of the entry answer routes to. With one alternative per call that is a single context per abstract caller state, but several concrete callers sharing an abstract state may still reach one context, which is exactly why a relation and not a function.

```
lemma admitted_contexts_alt:
  "admitted_contexts pgs p =
     routed_entry_context_rel (λci d. [entry_alt pgs p ci d gsol]) (λd e. pgam d e)
       (sol_env pgs p) global_of (route pgs)"
  by (simp add: admitted_contexts_def)

abbreviation entry_context_rel :: "'c context_policy" where
  "entry_context_rel ≡ admitted_contexts pgs p"
```

How a caller shows that this call is admitted at the context it computed: the concrete caller store is described by the recombined caller, and the entered store by the callee frame entered from it, each recombined against the solved global. The single alternative means a witness never has to name the alternatives list.

```
lemma admitted_contextsI:
  assumes caller: "s ∈ pgam (fst (entry_alt pgs p ci (dg_local (sol_env pgs p (Inl (u, ctx)))) gsol))
                    gsol"
    and entered_in: "s' ∈ pgam (entered ci (dg_local (sol_env pgs p (Inl (u, ctx))))) gsol"
  shows "route pgs u ctx (entered ci (dg_local (sol_env pgs p (Inl (u, ctx)))))
             (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci))
           ∈ entry_context_rel u ctx ci s s'"
  unfolding admitted_contexts_alt
  by (rule routed_entry_context_relI
        [where cont = "fst (entry_alt pgs p ci (dg_local (sol_env pgs p (Inl (u, ctx)))) gsol)"])
     (use caller entered_in in ‹simp_all add: entry_of_def sol_global_def›)
```

The same fact with the call action spelled as the caller has it. A witness names a literal `CallEdge`, while the rule above reconstructs one from the `call_info`'s three projections, and `rule` does not reduce the projections. Every concrete instance wants this shape.

```
lemma admitted_contextsI_call:
  assumes caller: "s ∈ pgam (fst (entry_alt pgs p (call_info_of (CallEdge dst pars args) q)
                     (dg_local (sol_env pgs p (Inl (u, ctx)))) gsol)) gsol"
    and entered_in: "s' ∈ pgam (entered (call_info_of (CallEdge dst pars args) q)
                          (dg_local (sol_env pgs p (Inl (u, ctx))))) gsol"
  shows "route pgs u ctx
           (entered (call_info_of (CallEdge dst pars args) q)
              (dg_local (sol_env pgs p (Inl (u, ctx)))))
           (CallEdge dst pars args)
         ∈ entry_context_rel u ctx (call_info_of (CallEdge dst pars args) q) s s'"
  using admitted_contextsI[OF caller entered_in] by simp

lemma entry_state_routed_analysis_sound:
  assumes solves: "terminates pgs p"
    and fwd_ok: "⋀u a v ctx. (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, a, v) ∈ intra (prog_cfg p) ⟹ (v, ctx) ∈ sol_vars pgs p"
    and call_fwd_ok: "⋀u ctx dst pars args q cont. (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ (FunctionEntry q,
              route pgs u ctx
                (entered (call_info_of (CallEdge dst pars args) q)
                   (dg_local (sol_env pgs p (Inl (u, ctx)))))
                (CallEdge dst pars args))
             ∈ sol_vars pgs p"
    and comb_fwd_ok: "⋀cl c1 dst pars args q cont. (cl, c1) ∈ sol_vars pgs p
        ⟹ (cl, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ (cont, c1) ∈ sol_vars pgs p"
  shows "routed_analysis (analysis_spec pgs p) (λd e. pgam d e) pgs (prog_cfg p)
     buffer_key global_of
     (route pgs) Bot (Lifted init_st) (place_rg pgs (Lifted init_st)) (sol_env pgs p)
     (sol_vars pgs p) (root_query p)
     seed (λd. d = Bot) entry_context_rel
     (λd e. map_lift (rd pgs) (place_cmb p d e)) γ_V empty_V classify"
proof (rule routed_analysis_sound_of
    [where adm = entry_context_rel, OF solves fwd_ok comb_fwd_ok])
  fix u ctx dst pars args q cont and s :: store and ctx'
  assume covV: "(u, ctx) ∈ sol_vars pgs p"
    and ce: "(u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)"
    and Rc: "ctx' ∈ entry_context_rel u ctx (call_info_of (CallEdge dst pars args) q) s
               (call_enter pgs (CallEdge dst pars args) s)"
  let ?ci = "call_info_of (CallEdge dst pars args) q"
  let ?caller = "dg_local (sol_env pgs p (Inl (u, ctx)))"
  from Rc[unfolded admitted_contexts_alt] obtain cont' entry
    where mem: "(cont', entry) ∈ set [entry_alt pgs p ?ci ?caller gsol]"
      and req0: "ctx' = route pgs u ctx entry
                   (CallEdge (ci_dst ?ci) (ci_formals ?ci) (ci_args ?ci))"
    by (rule routed_entry_context_relE)
  have "entry = entered ?ci ?caller" using mem by (simp add: entry_of_def) (metis snd_conv)
  with req0 have req: "route pgs u ctx (entered ?ci ?caller) (CallEdge dst pars args) = ctx'"
    by simp
  show "route pgs u ctx (entered ?ci ?caller) (CallEdge dst pars args) = ctx'
          ∧ (FunctionEntry q, ctx') ∈ sol_vars pgs p"
    using req call_fwd_ok[OF covV ce] by simp
next
  fix u ctx dst pars args q cont and s :: store
  assume covV: "(u, ctx) ∈ sol_vars pgs p"
    and ce: "(u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)"
    and sin: "s ∈ pgam (dg_local (sol_env pgs p (Inl (u, ctx)))) gsol"
  show "entry_context_rel u ctx (call_info_of (CallEdge dst pars args) q) s
                (call_enter pgs (CallEdge dst pars args) s) ≠ {}"
    unfolding admitted_contexts_alt
    by (rule routed_entry_context_rel_total)
       (use entry_cover[OF solves comb_fwd_ok[OF covV ce] ce sin]
        in ‹simp add: sol_global_def del: declared_global_iff›)
qed
```

The two endpoints an entry-state caller consumes: the per-context collecting bound, and the existence of a context for every valid activation trace, which a source-level caller needs to name a context witness at all. Both are the routed spine's own theorems at the interpretation just established.

```
context
  assumes solves: "terminates pgs p"
    and fwd_ok: "⋀u a v ctx. (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, a, v) ∈ intra (prog_cfg p) ⟹ (v, ctx) ∈ sol_vars pgs p"
    and call_fwd_ok: "⋀u ctx dst pars args q cont. (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ (FunctionEntry q,
```

```
              route pgs u ctx
                (entered (call_info_of (CallEdge dst pars args) q)
                   (dg_local (sol_env pgs p (Inl (u, ctx)))))
                (CallEdge dst pars args))
             ∈ sol_vars pgs p"
    and comb_fwd_ok: "⋀cl c1 dst pars args q cont. (cl, c1) ∈ sol_vars pgs p
        ⟹ (cl, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ (cl, c1) ∈ sol_vars pgs p"
begin

interpretation entry: routed_analysis "analysis_spec pgs p" "λd e. pgam d e" pgs
    "prog_cfg p" buffer_key global_of "route pgs" Bot "Lifted init_st"
    "place_rg pgs (Lifted init_st)" "sol_env pgs p" "sol_vars pgs p" "root_query p" seed
    "λd. d = Bot" entry_context_rel "λd e. map_lift (rd pgs) (place_cmb p d e)"
    γ_V empty_V classify
  by (rule entry_state_routed_analysis_sound [OF solves fwd_ok call_fwd_ok comb_fwd_ok])

lemma entry_init_env_le:
  assumes entry_cov: "(cfg_entry (prog_cfg p), root_ctx) ∈ sol_vars pgs p"
  shows "init_env pgs p ≤ genv global_of (sol_env pgs p)"
  using init_env_le_sol[OF solves entry_cov entry.pp_entry_init_global_bound[OF entry_cov]]
  by (simp add: sol_global_def)

corollary entry_state_activation_collect_sound:
  assumes entry_cov: "(cfg_entry (prog_cfg p), root_ctx) ∈ sol_vars pgs p"
  shows "A_pgs,entry_context_rel,root_ctx,prog_cfg p,cinit_stores pgs v ctx
         ⊆ pgam (reader pgs p (Inl (v, ctx))) gsol"
  using entry.activation_collect_dg_sound
      [OF entry_cov _ entry_init_env_le[OF entry_cov], of "cinit_stores pgs" v ctx]
    cinit_le_init
  by (simp add: reader_def sol_global_def)

theorem entry_state_has_context:
  assumes entry_cov: "(cfg_entry (prog_cfg p), root_ctx) ∈ sol_vars pgs p"
    and trace: "t ∈ T_pgs,prog_cfg p,cinit_stores pgs"
  shows "∃c. activation_context_rel pgs entry_context_rel root_ctx (prog_cfg p) t c"
  using entry.routed_valid_activation_trace_has_context
      [OF entry_cov _ entry_init_env_le[OF entry_cov] trace] cinit_le_init
  by simp
```

The two together: covering the entry is enough for the activation buckets to exhaust the context-insensitive collection, so a caller holding only a `node_collect` membership – which is what a source run delivers – can pass to the bucket its own call history produced. Without this the per-context bounds above say nothing about a run whose context is not known in advance.

```
theorem entry_state_node_collect_eq_Union:
  assumes entry_cov: "(cfg_entry (prog_cfg p), root_ctx) ∈ sol_vars pgs p"
  shows "C_pgs,prog_cfg p,cinit_stores pgs v
         = (⋃ctx. A_pgs,entry_context_rel,root_ctx,prog_cfg p,cinit_stores pgs v ctx)"
  by (rule node_collect_eq_Union_activation_of_has_context)
     (rule entry_state_has_context [OF entry_cov])

end
```

A route that never reads the state it is handed is a function of the call site and the caller's context alone, so the contexts it admits are the graph of a context function on concrete stores – the shape the call-string and context-insensitive policies both publish. `route_const` is the whole of what such a policy owes; `ctx_fun` is its trace-semantic counterpart.

```
corollary fun_route_activation_collect_sound:
  fixes ctx_fun :: "cfg_node ⇒ 'c ⇒ store ⇒ 'c"
  assumes route_const: "⋀u ctx d ca s. route pgs u ctx d ca = ctx_fun u ctx s"
    and solves: "terminates pgs p"
    and fwd_ok: "⋀u a v ctx. (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, a, v) ∈ intra (prog_cfg p) ⟹ (v, ctx) ∈ sol_vars pgs p"
    and call_fwd_ok: "⋀u ctx dst pars args q cont d. (u, ctx) ∈ sol_vars pgs p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ (FunctionEntry q, route pgs u ctx d (CallEdge dst pars args))
             ∈ sol_vars pgs p"
    and comb_fwd_ok: "⋀cl c1 dst pars args q cont. (cl, c1) ∈ sol_vars pgs p
        ⟹ (cl, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ (cont, c1) ∈ sol_vars pgs p"
    and entry_cov: "(cfg_entry (prog_cfg p), root_ctx) ∈ sol_vars pgs p"
  shows "A_pgs,context_policy_of_fun ctx_fun,root_ctx,prog_cfg p,cinit_stores pgs v ctx
         ⊆ pgam (reader pgs p (Inl (v, ctx))) gsol"
proof (rule activation_collect_sound_of[OF solves entry_cov fwd_ok comb_fwd_ok])
```

```
  fix u ctx dst pars args q cont and s :: store and ctx'
  assume covV: "(u, ctx) ∈ sol_vars pgs p"
    and ce: "(u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)"
    and Rc: "ctx' ∈ context_policy_of_fun ctx_fun u ctx
               (call_info_of (CallEdge dst pars args) q) s
               (call_enter pgs (CallEdge dst pars args) s)"
  have req: "route pgs u ctx d (CallEdge dst pars args) = ctx'" for d
    using Rc
    by (simp add: route_const[of _ _ _ _ "call_enter pgs (CallEdge dst pars args) s"])
  show "route pgs u ctx
          (entered (call_info_of (CallEdge dst pars args) q)
             (dg_local (sol_env pgs p (Inl (u, ctx)))))
          (CallEdge dst pars args) = ctx'
        ∧ (FunctionEntry q, ctx') ∈ sol_vars pgs p"
    using req call_fwd_ok[OF covV ce] by simp
next
  fix u ctx dst pars args q cont and s :: store
  show "context_policy_of_fun ctx_fun u ctx
                (call_info_of (CallEdge dst pars args) q) s
                (call_enter pgs (CallEdge dst pars args) s) ≠ {}"
    by simp
qed
```

The functional route's counterpart of the entry-state union. It needs nothing at all: `activation_context_of` is total, so every valid activation trace carries a context without any coverage having been established.

```
theorem fun_route_node_collect_eq_Union:
  "𝒞pgs,prog_cfg p,cinit_stores pgs v
     = (⋃ctx. 𝒜pgs,context_policy_of_fun ctx_fun,root_ctx,prog_cfg p,cinit_stores pgs v ctx)"
  by (rule node_collect_eq_Union_activation_of_fun)

end

end
```

Every variable in the local half

The placement every analysis uses unless it shares program globals: the specification is the component's own `dg_spec_of`, a point's state is its local unknown alone, and nothing is published at the analysis-wide global. Its placement obligations follow from the component's soundness and its single-alternative entry, so an instance owes exactly what it owed before the placement was a parameter.

```
theorem dg_analysis_whole_stateI:
  fixes comp :: "(vname ⇒ bool) ⇒ imp_prog ⇒ 's::semilattice_sup lifted local_spec"
    and seed :: "pp ⇒ 'c ⇒ 'k"
  assumes solver: "certified_solver solve solve_dom solve_c"
    and comp_sound:
      "⋀p. sound_local_spec (declared_global p)
               (λd. gamma_lift γV (map_lift (rd (declared_global p)) d))
               (comp (declared_global p) p)"
    and enter_single:
      "⋀p ci d. ls_enter (comp (declared_global p) p) (ls_channel (comp (declared_global p) p) d)
                    ci (d, d)
                  = [(d, dg_pipeline.comp_entry comp (declared_global p) p ci d)]"
    and empty_rd: "⋀p s. emp p s ⟷ emptyV (rd (declared_global p) s)"
    and emptyV_sound: "sound_emptiness emptyV γV"
    and seed_ne: "⋀v ctx. seed v ctx ≠ buffer_key"
    and classify_proved:
      "⋀c d s. classify c d = Check_Proved ⟹ s ∈ γV d ⟹ truthy (⟦c⟧e s)"
    and classify_refuted:
      "⋀c d s. classify c d = Check_Refuted ⟹ s ∈ γV d
         ⟹ ¬ truthy (⟦c⟧e s)"
    and bot_state_empty: "γV bot_state = {}"
    and init_sound:
      "⋀p. cinit_stores (declared_global p) ⊆ γV (rd (declared_global p) init_st)"
    and seed_ne_global: "⋀v ctx n. seed v ctx ≠ global_of n"
  shows "dg_analysis comp emp rd init_st buffer_key global_of seed
           solve solve_dom bot_state classify γV emptyV solve_c
           (λ𝒢 c. dg_spec_of c) (λ𝒢 d e. d) (λ𝒢 d. d) (λ𝒢 d. Bot) (λ𝒢 p. [])
           (λp ci d e. d)"
proof (rule dg_analysis.intro[OF solver dg_analysis_axioms.intro], goal_cases CompSound
    EnterSingle Contract Runs Deps EntrySound CmbBot Init EmptyRd EmptyV SeedNe SeedNeGlobal
    ClProved ClRefuted BotState InitSound)
  case (CompSound p) show ?case by (rule comp_sound)
next
  case (EnterSingle p ci d) show ?case by (rule enter_single)
```

```
next
  case (Contract p) show ?case
    unfolding dg_pipeline.analysis_spec_def by (rule dg_spec_of_contract[OF comp_sound])
next
  case (Runs p ci d σ) show ?case
    by (rule exI, simp add: dg_pipeline.analysis_spec_def dg_spec_of_def enter_single
        dg_pipeline.entry_alt_def, rule enter_runs_local_enter_transfer_mk_dg_man)
next
  case (Deps p ci d σ) show ?case
    by (rule exI, simp add: dg_pipeline.analysis_spec_def dg_spec_of_def enter_single
        dg_pipeline.entry_alt_def, rule enter_deps_local_enter_transfer_mk_dg_man)
next
  case (EntrySound p ci d σ pub s)
  let ?c = "comp (declared_global p) p"
  obtain q where "q ∈ set (ls_enter ?c (ls_channel ?c d) ci (d, d))"
      "s ∈ gamma_lift γV (map_lift (rd (declared_global p)) (fst q))"
      "call_enter (declared_global p) (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s
         ∈ gamma_lift γV (map_lift (rd (declared_global p)) (snd q))"
    using comp_sound[of p] EntrySound(3) ls_channel_sound[OF comp_sound[of p]]
    unfolding sound_local_spec_def by (metis fst_conv)
  then show ?case
    by (simp add: enter_single dg_pipeline.entry_alt_def)
next
  case (CmbBot p g) show ?case by simp
next
  case (Init p) show ?case by simp
next
  case (EmptyRd p s) show ?case by (rule empty_rd)
next
  case EmptyV show ?case by (rule emptyV_sound)
next
  case (SeedNe v ctx) show ?case by (rule seed_ne)
next
  case (SeedNeGlobal v ctx n) show ?case by (rule seed_ne_global)
next
  case (ClProved c d s) then show ?case by (rule classify_proved)
next
  case (ClRefuted c d s) then show ?case by (rule classify_refuted)
next
  case BotState show ?case by (rule bot_state_empty)
next
  case (InitSound p) show ?case by (rule init_sound)
qed
```

Program globals keyed by name

The placement that keeps each program global at its own unknown `global_of x`. The specification is the component wrapped by the keyed lifter, and a point's state is its local half recombined with the solved environment over the program's declared globals. The root publishes each declared global's initial value at its key, and the node-owned buffer stays empty.

Beyond the component's own soundness, the carrier owes monotone operations, that a name outside a read set is cut below `free` of that set, and two facts about concretization: the frame law `mix` of `keyed_split_contract`, and that a state is described by its local half recombined with its own cuts.

```
theorem dg_analysis_keyedI:
  fixes comp :: "(vname ⇒ bool) ⇒ imp_prog ⇒ 's::semilattice_sup lifted local_spec"
    and seed :: "pp ⇒ 'c ⇒ 'k"
    and global_of :: "vname ⇒ 'k"
    and cmb :: "'s lifted ⇒ 's lifted ⇒ 's lifted"
    and rg :: "vname ⇒ 's lifted ⇒ 's lifted"
  assumes solver: "certified_solver solve solve_dom solve_c"
    and comp_sound:
      "⋀p. sound_local_spec (declared_global p)
               (λd. gamma_lift γV (map_lift (rd (declared_global p)) d))
               (comp (declared_global p) p)"
    and enter_single:
      "⋀p ci d. ls_enter (comp (declared_global p) p) (ls_channel (comp (declared_global p) p) d)
                   ci (d, d)
                 = [(d, dg_pipeline.comp_entry comp (declared_global p) p ci d)]"
    and cmb_mono: "⋀d d' g g'. d ≤ d' ⟹ g ≤ g' ⟹ cmb d g ≤ cmb d' g'"
    and cmb_rl: "⋀x g. cmb (rl x) g = cmb x g"
    and rl_cmb: "⋀d g. rl (cmb d g) = rl d"
    and cmb_bot: "⋀g. cmb Bot g = Bot"
    and rg_mono: "⋀x v v'. v ≤ v' ⟹ rg x v ≤ rg x v'"
    and rg_free: "⋀x R v. x ∉ set R ⟹ rg x v ≤ free R"
```

```
    and mix: "⋀p l0 e r s s' W.
        s ∈ gamma_lift γV (map_lift (rd (declared_global p))
              (cmb l0 (full_view rg (declared_global_vars p) e)))
        ⟹ s' ∈ gamma_lift γV (map_lift (rd (declared_global p)) r)
        ⟹ (∀x. declared_global p x ⟶ x ∉ set W ⟶ s' x = s x)
        ⟹ s' ∈ gamma_lift γV (map_lift (rd (declared_global p))
              (cmb (rl r) (full_view rg (declared_global_vars p)
                 (λx. if x ∈ set W then rg x r else e x))))"
    and init_view: "⋀p. gamma_lift γV (map_lift (rd (declared_global p)) (Lifted init_st))
        ⊆ gamma_lift γV (map_lift (rd (declared_global p))
             (cmb (Lifted init_st) (full_view rg (declared_global_vars p)
               (λx. rg x (Lifted init_st)))))"
    and empty_rd: "⋀p s. emp p s ⟷ emptyV (rd (declared_global p) s)"
    and emptyV_sound: "sound_emptiness emptyV γV"
    and seed_ne: "⋀v ctx. seed v ctx ≠ buffer_key"
    and seed_ne_global: "⋀v ctx n. seed v ctx ≠ global_of n"
    and classify_proved:
      "⋀c d s. classify c d = Check_Proved ⟹ s ∈ γV d ⟹ truthy (⟦c⟧e s)"
    and classify_refuted:
      "⋀c d s. classify c d = Check_Refuted ⟹ s ∈ γV d
         ⟹ ¬ truthy (⟦c⟧e s)"
    and bot_state_empty: "γV bot_state = {}"
    and init_sound:
      "⋀p. cinit_stores (declared_global p) ⊆ γV (rd (declared_global p) init_st)"
  shows "dg_analysis comp emp rd init_st buffer_key global_of seed
          solve solve_dom bot_state classify γV emptyV solve_c
          (λp c. keyed_split_spec (declared_global p) cmb rl rg free c)
          (λp d e. cmb d (full_view rg (declared_global_vars p) e)) (λG. rl) (λG d. Bot)
          (λG p. map (λx. (x, rg x (Lifted init_st))) (declared_global_vars p))
          (λp ci d e. cmb d (view_of rg (call_global_reads (declared_global p) (ci_args ci)) e
             (free (call_global_reads (declared_global p) (ci_args ci)))))"
proof -
  let ?gm = "λp d. gamma_lift γV (map_lift (rd (declared_global p)) d)"
  have gm_mono: "⋀p x y. x ≤ y ⟹ ?gm p x ⊆ ?gm p y"
    using comp_sound unfolding sound_local_spec_def by metis
  show ?thesis
  proof (rule dg_analysis.intro[OF solver dg_analysis_axioms.intro], goal_cases CompSound
      EnterSingle Contract Runs Deps EntrySound CmbBot Init EmptyRd EmptyV SeedNe SeedNeGlobal
      ClProved ClRefuted BotState InitSound)
    case (CompSound p) show ?case by (rule comp_sound)
  next
    case (EnterSingle p ci d) show ?case by (rule enter_single)
  next
    case (Contract p) show ?case
      unfolding dg_pipeline.analysis_spec_def
      using comp_sound[where p = p] cmb_mono rg_mono rg_free mix[where p = p]
      by (rule keyed_split_contract)
  next
    case (Runs p ci d σ) show ?case
      using enter_runs_keyed_enter_transfer[of cmb rl rg free
          "call_global_reads (declared_global p) (ci_args ci)"
          "global_names_in (declared_global p) (ci_formals ci)"
          "λd. ls_enter (comp (declared_global p) p) (ls_channel (comp (declared_global p) p) d)
             ci (d, d)" d global_of σ]
      by (auto simp: dg_pipeline.analysis_spec_def dg_pipeline.entry_alt_def enter_single Let_def)
  next
    case (Deps p ci d σ) show ?case
      using enter_deps_keyed_enter_transfer[of cmb rl rg free
          "call_global_reads (declared_global p) (ci_args ci)"
          "global_names_in (declared_global p) (ci_formals ci)"
          "λd. ls_enter (comp (declared_global p) p) (ls_channel (comp (declared_global p) p) d)
             ci (d, d)" d global_of σ]
      by (auto simp: dg_pipeline.analysis_spec_def dg_pipeline.entry_alt_def enter_single Let_def)
  next
    case (EntrySound p ci d σ pub s)
    let ?G = "declared_global p" and ?xs = "declared_global_vars p"
    let ?c = "comp ?G p"
    let ?e = "genv global_of σ"
    let ?R = "call_global_reads ?G (ci_args ci)"
    let ?v = "view_of rg ?R ?e (free ?R)"
    let ?w = "cmb d ?v"
    let ?E = "dg_pipeline.comp_entry comp ?G p ci ?w"
    let ?W = "global_names_in ?G (ci_formals ci)"
    let ?ce = "call_enter ?G (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci))"
    have alt: "dg_pipeline.entry_alt comp (λG. rl)
        (λp ci d e. cmb d (view_of rg (call_global_reads (declared_global p) (ci_args ci)) e
```

```
        (free (call_global_reads (declared_global p) (ci_args ci)))))
      ?𝒢 p ci d ?e = (rl ?w, rl ?E)"
    by (simp add: dg_pipeline.entry_alt_def Let_def)
  have R0: "enter_runs (enter♯ (dg_pipeline.analysis_spec comp
      (λp. keyed_split_spec (declared_global p) cmb rl rg free) ?𝒢 p) ci)
    (mk_dg_man d global_of) σ [(rl ?w, rl ?E)] (pub_sides global_of rg ?W (?E ⊔ bot))"
    using enter_runs_keyed_enter_transfer[of cmb rl rg free ?R ?W
      "λd. ls_enter ?c (ls_channel ?c d) ci (d, d)" d global_of σ]
    by (simp add: dg_pipeline.analysis_spec_def enter_single)
  have pub: "pub = pub_sides global_of rg ?W (?E ⊔ bot)"
    using enter_runsD_sides[OF EntrySound(1)[unfolded alt], of "λ_. Answer bot"]
      enter_runsD_sides[OF R0, of "λ_. Answer bot"]
    by (simp add: bot_fun_def[symmetric])
  have E_Bot: "?E ⊔ Bot = ?E" by (cases ?E) simp_all
  have le: "rg x ?E ≤ ?e x" if "x ∈ set ?W" for x
  proof -
    have "DG bot (rg x ?E) ≤ pub (Inr (global_of x))"
      using pub_sides_ge[OF that, of rg "?E ⊔ bot" global_of] by (simp add: pub E_Bot)
    then have "rg x ?E ≤ genv global_of pub x" by (simp add: less_eq_dg_state_def)
    also have "... ≤ ?e x" using EntrySound(2) by (simp add: le_fun_def)
    finally show ?thesis .
  qed
  have view_ge: "full_view rg ?xs ?e ≤ ?v"
    unfolding full_view_def
  proof (rule view_of_le)
    fix x assume "x ∈ set ?xs"
    show "rg x (?e x) ≤ ?v"
    proof (cases "x ∈ set ?R")
      case True then show ?thesis by (rule view_of_ge_mem)
    next
      case False
      then show ?thesis using rg_free view_of_ge_acc by (blast intro: order_trans)
    qed
  qed simp
  have s0: "s ∈ ?gm p (cmb d (full_view rg ?xs ?e))" using EntrySound(3) by simp
  have sw: "s ∈ ?gm p ?w"
    using s0 gm_mono[OF cmb_mono[OF order_refl view_ge]] by blast
  have sE: "?ce s ∈ ?gm p ?E"
  proof -
    obtain q where "q ∈ set (ls_enter ?c (ls_channel ?c ?w) ci (?w, ?w))" "?ce s ∈ ?gm p (snd q)"
      using comp_sound[where p = p] sw ls_channel_sound[OF comp_sound[where p = p] sw]
      unfolding sound_local_spec_def by (metis fst_conv)
    then show ?thesis by (simp add: enter_single)
  qed
  have fr: "∀x. ?𝒢 x ⟶ x ∉ set ?W ⟶ ?ce s x = s x"
    by (auto intro: call_enter_frame)
  have "?ce s ∈ ?gm p (cmb (rl ?E) (full_view rg ?xs (λx. if x ∈ set ?W then rg x ?E else ?e x)))"
    by (rule mix[OF s0 sE fr])
  moreover have "cmb (rl ?E) (full_view rg ?xs (λx. if x ∈ set ?W then rg x ?E else ?e x))
      ≤ cmb (rl ?E) (full_view rg ?xs ?e)"
  proof (intro cmb_mono order_refl full_view_mono[OF rg_mono])
    fix x
    show "(if x ∈ set ?W then rg x ?E else ?e x) ≤ ?e x"
      using le[of x] by (cases "x ∈ set ?W") simp_all
  qed
  ultimately have "?ce s ∈ ?gm p (cmb (rl ?E) (full_view rg ?xs ?e))"
    using gm_mono by blast
  moreover have "s ∈ ?gm p (cmb (rl ?w) (full_view rg ?xs ?e))"
    using s0 by (simp add: rl_cmb cmb_rl)
  ultimately show ?case by (simp add: alt)
next
  case (CmbBot p g) show ?case by (simp add: cmb_bot)
next
  case (Init p)
  let ?xs = "declared_global_vars p"
  let ?ie = "dg_pipeline.init_env init_st buffer_key global_of (λ𝒢 d. Bot)
    (λ𝒢 p. map (λx. (x, rg x (Lifted init_st))) (declared_global_vars p)) (declared_global p) p"
  have ie: "rg x (Lifted init_st) ≤ ?ie x" if x: "x ∈ set ?xs" for x
  proof -
    have "rg x (Lifted init_st)
        ≤ (⨆(m, d)←filter (λ(m, d). m = x) (map (λy. (y, rg y (Lifted init_st))) ys). d)"
      if "x ∈ set ys" for ys
      using that by (induction ys) (auto intro: le_supI1 le_supI2)
    then show ?thesis using x by (simp add: dg_pipeline.init_env_def le_supI2)
  qed
  have "full_view rg ?xs (λx. rg x (Lifted init_st)) ≤ full_view rg ?xs ?ie"
```

```
    unfolding full_view_def
  proof (rule view_of_le)
    fix x assume x: "x ∈ set ?xs"
    show "rg x (rg x (Lifted init_st)) ≤ view_of rg ?xs ?ie bot"
      using rg_mono[OF ie[OF x], of x] view_of_ge_mem[OF x, of rg ?ie bot] by (rule order_trans)
  qed simp
  then show ?case
    using init_view[where p = p] gm_mono[OF cmb_mono[OF order_refl], where p = p] by blast
next
  case (EmptyRd p s) show ?case by (rule empty_rd)
next
  case EmptyV show ?case by (rule empty_V_sound)
next
  case (SeedNe v ctx) show ?case by (rule seed_ne)
next
  case (SeedNeGlobal v ctx n) show ?case by (rule seed_ne_global)
next
  case (ClProved c d s) then show ?case by (rule classify_proved)
next
  case (ClRefuted c d s) then show ?case by (rule classify_refuted)
next
  case BotState show ?case by (rule bot_state_empty)
next
  case (InitSound p) show ?case by (rule init_sound)
qed
qed
```

An executable analysis as the pipeline's component

What an executable non-relational analysis owes, stated over its own transfers: the abstract transfer it implements is sound, its executable mirror represents that transfer, and its routing, solver, classifier and entry state satisfy the pipeline's contracts on the represented abstract store. Its component is `exec_local_spec`, so the generic pipeline runs exactly its local specification, and every theorem above holds of it.

`route_abs` is the same routing decision taken on the abstract carrier, kept for the analyses that state it; the pipeline's soundness does not use it.

```
locale dg_analysis_exec = certified_solver solve solve_dom solve_c
  for tf_st :: "(vname ⇒ bool) ⇒ edge_action
                ⇒ 'a::numeric_domain default_st ⇒ 'a default_st"
    and enter_st :: "(vname ⇒ bool) ⇒ call_info ⇒ 'a default_st ⇒ 'a default_st"
    and init_st :: "'a default_st"
    and buffer_key :: 'k
    and global_of :: "unit ⇒ 'k"
    and seed :: "pp ⇒ 'c ⇒ 'k"
    and route :: "(vname ⇒ bool) ⇒ pp ⇒ 'c ⇒ 'a default_st lifted ⇒ call_action ⇒ 'c"
    and root_ctx :: 'c
    and solve :: "(pp × 'c, 'k, ('a default_st lifted, 'a default_st lifted) dg_state) eqsT
                  ⇒ pp × 'c
                  ⇒ (pp × 'c) set
                       × (pp × 'c + 'k ⇒ ('a default_st lifted, 'a default_st lifted) dg_state)"
    and solve_dom :: "(pp × 'c, 'k, ('a default_st lifted, 'a default_st lifted) dg_state) eqsT
                      ⇒ pp × 'c ⇒ bool"
    and bot_state :: "'a abs_state"
    and classify :: "exp ⇒ 'a abs_state ⇒ check_result"
    and sk :: "'a abs_state ⇒ 'a abs_state"
    and asn :: "vname ⇒ exp ⇒ 'a abs_state ⇒ 'a abs_state"
    and spc :: "special_call ⇒ vname ⇒ 'a abs_state ⇒ 'a abs_state"
    and br :: "exp ⇒ bool ⇒ 'a abs_state ⇒ 'a abs_state"
    and bd :: "pname ⇒ 'a abs_state ⇒ 'a abs_state"
    and rt :: "exp option ⇒ pname ⇒ 'a abs_state ⇒ 'a abs_state"
    and en :: "(vname ⇒ bool) ⇒ call_info ⇒ 'a abs_state ⇒ 'a abs_state"
    and ev :: "analysis_event ⇒ 'a abs_state ⇒ 'a abs_state"
    and route_abs :: "(vname ⇒ bool) ⇒ pp ⇒ 'c ⇒ 'a abs_state lifted ⇒ call_action ⇒ 'c"
    and solve_c :: "(pp × 'c, 'k,
                     ('a default_st lifted, 'a default_st lifted) dg_state) eqsT
                    ⇒ pp × 'c
                    ⇒ ((pp × 'c) set
                         × (pp × 'c + 'k
                              ⇒ ('a default_st lifted, 'a default_st lifted) dg_state)) option" +
  assumes tf_sound: "⋀𝒢. sound_nonrelational_transfer 𝒢 sk asn spc br bd rt (en 𝒢) ev"
    and tf_commute:
      "⋀𝒢 a s. live_default_st 𝒢 s
         ⟹ ϱ𝒢 (tf_st 𝒢 a s)
              = local_spec_step sk asn spc br bd rt ev a (ϱ𝒢 s)"
    and enter_commute:
```

```
    "⋀G ci s. ϱG (enter_st G ci s)
                = en G ci (ϱG s)"
  and route_agree:
    "⋀G u ctx d ca. route G u ctx d ca
       = route_abs G u ctx (ϱG d) ca"
  and exec_seed_ne_buffer_key: "⋀v ctx. seed v ctx ≠ buffer_key"
  and exec_classify_proved:
    "⋀c d s. classify c d = Check_Proved ⟹ s ∈ γ d ⟹ truthy (⟦c⟧e s)"
  and exec_classify_refuted:
    "⋀c d s. classify c d = Check_Refuted ⟹ s ∈ γ d
       ⟹ ¬ truthy (⟦c⟧e s)"
  and bot_state_eq: "bot_state = bot"
  and exec_init_sound: "⋀G. cinit_stores G ⊆ default_st_gamma G init_st"
  and exec_own_key: "⋀n. global_of n = buffer_key"

sublocale dg_analysis_exec ⊆ dg_analysis
  "λG p. exec_local_spec G (default_st_is_bot_for (declared_global_vars p))
          (tf_st G) (enter_st G)"
  "λp. default_st_is_bot_for (declared_global_vars p)"
  default_st_to_fun init_st buffer_key global_of seed route root_ctx
    solve solve_dom bot_state classify
  gamma_state is_empty_state solve_c "λG c. dg_spec_of c" "λG d e. d" "λG d. d"
    "λG d. Bot" "λG p. []" "λp ci d e. d"
proof (rule dg_analysis_whole_stateI[OF certified_solver_axioms],
  goal_cases CompSound EnterSingle EmptyExact EmptyVExact SeedNe ClProved ClRefuted BotState
  Init SeedNeGlobal)
  case (CompSound p)
  interpret dom: dg_domain_exec "declared_global p"
    "default_st_is_bot_for (declared_global_vars p)" "tf_st (declared_global p)"
    "enter_st (declared_global p)" sk asn spc br bd rt "en (declared_global p)" ev
    by unfold_locales
      (rule tf_commute, assumption,
       rule enter_commute,
       rule default_st_is_bot_for_iff[OF declared_global_iff])
  show ?case
    by (rule dom.exec_local_spec_sound[OF tf_sound, unfolded gamma_lift_default_st_gamma_to_fun])
next
  case (EnterSingle p ci d)
  then show ?case by (simp add: dg_pipeline.comp_entry_def)
next
  case (EmptyExact p s)
  then show ?case by (rule default_st_is_bot_for_iff[OF declared_global_iff])
next
  case EmptyVExact show ?case
    by (rule sound_emptinessI) (erule is_empty_state_gamma_state_empty)
next
  case (SeedNe v ctx) then show ?case by (rule exec_seed_ne_buffer_key)
next
  case (ClProved c d s) then show ?case by (rule exec_classify_proved)
next
  case (ClRefuted c d s) then show ?case by (rule exec_classify_refuted)
next
  case BotState then show ?case
    by (simp add: bot_state_eq is_empty_state_iff_gamma_state_empty[symmetric])
next
  case (Init p) then show ?case
    using exec_init_sound[of "declared_global p"] by (simp add: default_st_gamma_def)
next
  case (SeedNeGlobal v ctx n) show ?case
    unfolding exec_own_key by (rule exec_seed_ne_buffer_key)
qed
```

The component's soundness at the carrier's own concretization, which is what a caller composing components on the carrier needs.

```
lemma (in dg_analysis_exec) exec_comp_sound:
  "sound_local_spec (declared_global p) (gamma_lift (default_st_gamma (declared_global p)))
     (exec_local_spec (declared_global p) (default_st_is_bot_for (declared_global_vars p))
        (tf_st (declared_global p)) (enter_st (declared_global p)))"
  unfolding gamma_lift_default_st_gamma_to_fun by (rule comp_sound)
```

The state a call enters its callee with, at the executable component: the entry transfer, lifted over reachability.

```
lemma (in dg_analysis_exec) comp_entry_exec:
  "dg_pipeline.comp_entry
     (λG p. exec_local_spec G (default_st_is_bot_for (declared_global_vars p))
        (tf_st G) (enter_st G))
```

```
    𝒢 p ci d
  = transfer_lift (default_st_is_bot_for (declared_global_vars p)) (enter_st 𝒢 ci) d"
  by (simp add: dg_pipeline.comp_entry_def)
```

The same against the solved global, the shape `dg_pipeline.ctx_succ` reads, so a caller can fold an executable entry back into it.

```
lemma (in dg_analysis_exec) entry_of_exec:
  "entry_of 𝒢 p ci d (sol_global 𝒢 p)
   = transfer_lift (default_st_is_bot_for (declared_global_vars p)) (enter_st 𝒢 ci) d"
  by (simp add: dg_pipeline.entry_of_def dg_pipeline.entry_alt_def comp_entry_exec)

end
```

9.5 DG_Live_Unknowns

```
theory DG_Live_Unknowns
  imports DG_Analysis "Voblint_Compile.Live_Nodes"
begin
```

Which solved keys a terminating solve is closed on

The key set a terminating solve returns is not one the soundness argument can use as it stands: it may hold keys the solver queried under an earlier value and no longer reads, and keys of procedures whose result it never solved. `live_unknowns` keeps a solved key when its node belongs to a procedure (`Voblint_Compile.Live_Nodes`) whose result is solved at the same context. From what each generated equation reads under the final valuation, `live_unknowns_cover` shows that set closed under every intra edge, every call continuation and every call that enters a live state, with no premise beyond termination and a well-formed program. The endpoints below restate the routed soundness over these keys and carry it to the published table.

What a routed equation reads

```
lemma dep_aux_routed_node_rhs:
  assumes wf: "⋀w. ∀q ∈ set (routed_contribution_programs pred_sel site_sel route
                    it cmb extra g cx w). sp_wf q"
  shows "dep_aux τ (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d₀ g₀
          (v, cx))
     = (⋃q∈set (routed_contribution_programs pred_sel site_sel route it cmb extra g cx v).
          dep_program τ q)"
  by (cases "v = cfg_entry g")
     (simp_all add: routed_node_rhs_def Let_def dep_program_side_rhs_fold_dg_char[OF wf])

lemma dep_L_routed_node_rhs_intra:
  assumes fin: "finite (intra g)" and e: "(u, a, v) ∈ intra g"
    and wf: "⋀w. ∀q ∈ set (routed_contribution_programs
                    intra_predecessor_addr_list call_site_list route
                    (λcx src a. dg_spec_edge_program S a src slot) cmb extra g cx w). sp_wf q"
  shows "(u, cx) ∈ dep_L (routed_node_rhs intra_predecessor_addr_list call_site_list buffer_key_at route
           (λcx src a. dg_spec_edge_program S a src slot) cmb extra g bot0 d₀ g₀) τ (v, cx)"
proof -
  have "dg_spec_edge_program S a (Inl (u, cx)) slot
         ∈ set (routed_contribution_programs intra_predecessor_addr_list call_site_list route
                  (λcx src a. dg_spec_edge_program S a src slot) cmb extra g cx v)"
    using e fin
    by (force simp: routed_contribution_programs_def intra_predecessor_addr_list_def
        intra_predecessors_def)
  then show ?thesis
    unfolding dep_L_def dep_def
    using dep_dg_spec_edge_program_source[of "Inl (u, cx)" τ S a slot]
    by (auto simp: dep_aux_routed_node_rhs[OF wf])
qed

lemma dep_program_routed_call_program:
  assumes wfS: "dg_spec_wf S"
  shows "dep_program τ (routed_call_program S global_of seed resolve is_bot route ctx ca cc v)
     = insert (Inl (cc, ctx))
         (⋃q∈set (resolve v cc ca (dg_local (τ (Inl (cc, ctx))))).
            dep_program τ (routed_callee_call_program S global_of seed route is_bot ctx ca cc
                     (dg_local (τ (Inl (cc, ctx)))) q))"
  by (simp add: routed_call_program_def sp_compile_bind sp_wf_observes
      dep_program_side_rhs_fold_dg_char[OF sp_wf_routed_callee_call_programs[OF wfS]])

lemma routed_call_program_mem:
  assumes fin: "finite (calls g)" and e: "(u, ca, FunctionEntry q, k) ∈ calls g"
  shows "routed_call_program S global_of seed resolve is_bot route cx ca u k
          ∈ set (routed_contribution_programs pred_sel call_site_list route it
                   (routed_call_program S global_of seed resolve is_bot) extra g cx k)"
proof -
  have "(u, ca) ∈ set (call_site_list g k)" using e fin by auto
  then show ?thesis by (rule routed_contribution_programs_combineI)
qed

lemma dep_L_routed_node_rhs_call_site:
  assumes fin: "finite (calls g)" and e: "(u, ca, FunctionEntry q, k) ∈ calls g"
    and wfS: "dg_spec_wf S"
    and wf: "⋀w. ∀q ∈ set (routed_contribution_programs pred_sel call_site_list route it
                    (routed_call_program S global_of seed resolve is_bot) extra g cx w). sp_wf q"
  shows "(u, cx) ∈ dep_L (routed_node_rhs pred_sel call_site_list buffer_key_at route it
```

```
          (routed_call_program S global_of seed resolve is_bot) extra g bot0 d₀ g₀) τ (k, cx)"
proof -
  have "Inl (u, cx)
        ∈ dep_program τ (routed_call_program S global_of seed resolve is_bot route cx ca u k)"
    by (simp add: dep_program_routed_call_program[OF wfS])
  then show ?thesis
    unfolding dep_L_def dep_def
    using routed_call_program_mem
          [OF fin e, of S global_of seed resolve is_bot route cx pred_sel it extra]
    by (auto simp: dep_aux_routed_node_rhs[OF wf])
qed

lemma dep_L_routed_node_rhs_callee_result:
  assumes fin: "finite (calls g)" and e: "(u, ca, FunctionEntry q, k) ∈ calls g"
    and deps: "enter_deps (enter♯ S (call_info_of ca q))
                (mk_dg_man (dg_local (τ (Inl (u, cx)))) global_of) τ pairs D"
    and mem: "(c, ent) ∈ set pairs"
    and res: "q ∈ set (resolve k u ca (dg_local (τ (Inl (u, cx)))))"
    and nb: "¬ is_bot ent"
    and wfS: "dg_spec_wf S"
    and wf: "⋀w. ∀q ∈ set (routed_contribution_programs pred_sel call_site_list route it
                    (routed_call_program S global_of seed resolve is_bot) extra g cx w). sp_wf q"
  shows "(FunctionResult q, route u cx ent ca)
         ∈ dep_L (routed_node_rhs pred_sel call_site_list buffer_key_at route it
              (routed_call_program S global_of seed resolve is_bot) extra g bot0 d₀ g₀) τ (k, cx)"
proof -
  let ?d = "dg_local (τ (Inl (u, cx)))"
  have callee: "dep_program τ (routed_callee_call_program S global_of seed route is_bot cx ca u ?d q)
      = D ∪ dep_program τ (side_rhs_fold_dg bot
          (map (routed_call_alternative_program S global_of seed route is_bot cx ca u q) pairs))"
    unfolding routed_callee_call_program_def sp_compile_bind
    by (rule enter_depsD[OF deps, simplified])
  have "Inl (FunctionResult q, route u cx ent ca)
        ∈ dep_program τ (routed_callee_call_program S global_of seed route is_bot cx ca u ?d q)"
  proof -
    have "Inl (FunctionResult q, route u cx ent ca)
          ∈ dep_program τ (routed_call_alternative_program S global_of seed route is_bot cx ca u q
                       (c, ent))"
      using nb by (simp add: routed_call_alternative_program_def Let_def)
    then show ?thesis
      unfolding callee
        dep_program_side_rhs_fold_dg_char[OF sp_wf_routed_call_alternative_programs[OF wfS]]
      using mem by auto
  qed
  then have "Inl (FunctionResult q, route u cx ent ca)
             ∈ dep_program τ (routed_call_program S global_of seed resolve is_bot route cx ca u k)"
    using res by (auto simp: dep_program_routed_call_program[OF wfS])
  then show ?thesis
    unfolding dep_L_def dep_def
    using routed_call_program_mem
          [OF fin e, of S global_of seed resolve is_bot route cx pred_sel it extra]
    by (auto simp: dep_aux_routed_node_rhs[OF wf])
qed
```

Keys of a solve whose activation returned

The solved key set is closed under the equations' reads (`sol_vars_dep_closed`). Walking those reads backwards, `live_unknowns_cover` shows `live_unknowns` closed under intra edges, call continuations and the entries of callees a call actually enters.

```
context dg_analysis
begin

lemma analysis_contribs_wf [simp]:
  "∀q ∈ set (routed_contribution_programs intra_predecessor_addr_list call_site_list
       (route (declared_global p))
       (λctx' src a. dg_spec_edge_program (analysis_spec (declared_global p) p) a src
          global_of)
       (routed_call_program (analysis_spec (declared_global p) p) global_of seed rsv
          (λd. d = Bot))
       (routed_entry_seed_programs seed) cfg cx w). sp_wf q"
  by (rule routed_contribution_programs_wf) auto

lemma sol_vars_dep_closed:
  assumes solves: "terminates (declared_global p) p"
    and x: "x ∈ sol_vars (declared_global p) p"
```

```
    and y: "y ∈ dep_L
       (routed_node_rhs intra_predecessor_addr_list call_site_list (λ_. buffer_key)
          (route (declared_global p))
          (λctx' src a. dg_spec_edge_program (analysis_spec (declared_global p) p) a src global_of)
          (routed_call_program (analysis_spec (declared_global p) p) global_of seed
             (static_resolve (prog_cfg p)) (λd. d = Bot))
          (routed_entry_seed_programs seed)
          (prog_cfg p) Bot (Lifted init_st) (place_rg (declared_global p) (Lifted init_st)))
       (sol_env (declared_global p) p) x"
  shows "y ∈ sol_vars (declared_global p) p"
  using pp_routed[OF solves] x y by blast

lemma prog_cfg_finite: "finite (intra (prog_cfg p))" "finite (calls (prog_cfg p))"
  unfolding prog_cfg_def by (simp_all add: compile_prog_finite)

lemma sol_vars_back_intra:
  assumes solves: "terminates (declared_global p) p"
    and v: "(v, cx) ∈ sol_vars (declared_global p) p" and e: "(u, a, v) ∈ intra (prog_cfg p)"
  shows "(u, cx) ∈ sol_vars (declared_global p) p"
  by (rule sol_vars_dep_closed
        [OF solves v dep_L_routed_node_rhs_intra[OF prog_cfg_finite(1) e analysis_contribs_wf]])

lemma sol_vars_back_call_site:
  assumes solves: "terminates (declared_global p) p"
    and k: "(k, cx) ∈ sol_vars (declared_global p) p"
    and e: "(u, ca, FunctionEntry q, k) ∈ calls (prog_cfg p)"
  shows "(u, cx) ∈ sol_vars (declared_global p) p"
  by (rule sol_vars_dep_closed
        [OF solves k dep_L_routed_node_rhs_call_site
           [OF prog_cfg_finite(2) e dg_spec_wf_analysis_spec analysis_contribs_wf]])

lemma sol_vars_callee_result:
  assumes solves: "terminates (declared_global p) p"
    and k: "(k, cx) ∈ sol_vars (declared_global p) p"
    and e: "(u, ca, FunctionEntry q, k) ∈ calls (prog_cfg p)"
    and nb: "entry_of (declared_global p) p (call_info_of ca q)
               (dg_local (sol_env (declared_global p) p (Inl (u, cx))))
               (sol_global (declared_global p) p) ≠ Bot"
  shows "(FunctionResult q, ctx_succ (declared_global p) p u cx ca q)
         ∈ sol_vars (declared_global p) p"
proof -
  let ?d = "dg_local (sol_env (declared_global p) p (Inl (u, cx)))"
  let ?alt =
    "entry_alt (declared_global p) p (call_info_of ca q) ?d (sol_global (declared_global p) p)"
  have res: "q ∈ set (static_resolve (prog_cfg p) k u ca ?d)"
    using e prog_cfg_finite(2) by simp
  obtain D where deps: "enter_deps (enter♯ (analysis_spec (declared_global p) p) (call_info_of ca q))
     (mk_dg_man ?d global_of) (sol_env (declared_global p) p) [?alt] D"
    using place_enter_deps unfolding sol_global_def by blast
  have mem: "(fst ?alt, snd ?alt) ∈ set [?alt]" by simp
  show ?thesis
    unfolding ctx_succ_def entry_of_def
    by (rule sol_vars_dep_closed[OF solves k],
        rule dep_L_routed_node_rhs_callee_result[where resolve = "static_resolve (prog_cfg p)"
            and τ = "sol_env (declared_global p) p" and cx = cx,
          OF prog_cfg_finite(2) e deps mem res])
       (use nb in ‹simp add: entry_of_def›, rule dg_spec_wf_analysis_spec, rule
        analysis_contribs_wf)
qed

lemma sol_vars_back_reach:
  assumes solves: "terminates (declared_global p) p"
    and w: "(w, cx) ∈ sol_vars (declared_global p) p"
    and r: "local_reaches (prog_cfg p) y w"
  shows "(y, cx) ∈ sol_vars (declared_global p) p"
  using r[unfolded local_reaches_def]
proof (induction rule: converse_rtrancl_induct)
  case base show ?case by (rule w)
next
  case (step y z)
  from step.hyps(1) show ?case
    unfolding local_succ_rel_def
    using sol_vars_back_intra[OF solves step.IH] sol_vars_back_call_site[OF solves step.IH]
    by blast
qed
definition live_unknowns :: "imp_prog ⇒ (pp × 'c) set" where
```

```
  "live_unknowns p =
     {(u, cx) ∈ sol_vars (declared_global p) p.
        ∃q. prog_node (prog_table p) (prog_procs p) q u
          ∧ (FunctionResult q, cx) ∈ sol_vars (declared_global p) p}"

lemma live_unknowns_sub: "live_unknowns p ⊆ sol_vars (declared_global p) p"
  unfolding live_unknowns_def by blast

lemma live_unknowns_of_result:
  assumes wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
    and live: "prog_node (prog_table p) (prog_procs p) q y"
    and r: "(FunctionResult q, cx) ∈ sol_vars (declared_global p) p"
  shows "(y, cx) ∈ live_unknowns p"
proof -
  have "local_reaches (prog_cfg p) y (FunctionResult q)"
    unfolding prog_cfg_def by (rule prog_node_reaches[OF live])
  then have "(y, cx) ∈ sol_vars (declared_global p) p" by (rule sol_vars_back_reach[OF solves r])
  then show ?thesis unfolding live_unknowns_def using live r by blast
qed

theorem live_unknowns_cover:
  assumes wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
  shows "ctx_vars_cover_live (prog_cfg p) (λ_ _. True) (live_succ (declared_global p) p)
           root_ctx (live_unknowns p)"
proof (rule ctx_vars_cover_liveI)
  have root: "(FunctionResult prog_main_name, root_ctx) ∈ sol_vars (declared_global p) p"
    using pp_routed[OF solves] by (simp add: root_query_def prog_cfg_def)
  show "(cfg_entry (prog_cfg p), root_ctx) ∈ live_unknowns p"
    using live_unknowns_of_result[OF wf solves prog_node_main_entry[OF wf] root]
    by (simp add: prog_cfg_def)
next
  fix u a v ctx
  assume u: "(u, ctx) ∈ live_unknowns p" and e: "(u, a, v) ∈ intra (prog_cfg p)"
  from u obtain q where lq: "prog_node (prog_table p) (prog_procs p) q u"
    and r: "(FunctionResult q, ctx) ∈ sol_vars (declared_global p) p"
    unfolding live_unknowns_def by blast
  have "prog_node (prog_table p) (prog_procs p) q v"
    using prog_node_intra[OF wf lq] e by (simp add: prog_cfg_def)
  then show "(v, ctx) ∈ live_unknowns p" by (rule live_unknowns_of_result[OF wf solves _ r])
next
  fix u ctx dst fs as q k
  assume u: "(u, ctx) ∈ live_unknowns p"
    and e: "(u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls (prog_cfg p)"
  from u obtain r where lr: "prog_node (prog_table p) (prog_procs p) r u"
    and res: "(FunctionResult r, ctx) ∈ sol_vars (declared_global p) p"
    unfolding live_unknowns_def by blast
  have "prog_node (prog_table p) (prog_procs p) r k"
    using prog_node_calls[OF wf lr] e by (simp add: prog_cfg_def)
  then show "(k, ctx) ∈ live_unknowns p" by (rule live_unknowns_of_result[OF wf solves _ res])
next
  fix u ctx dst fs as q k ctx'
  assume u: "(u, ctx) ∈ live_unknowns p"
    and e: "(u, CallEdge dst fs as, FunctionEntry q, k) ∈ calls (prog_cfg p)"
    and s: "live_succ (declared_global p) p u ctx (CallEdge dst fs as) q = Some ctx'"
  from u obtain r where lr: "prog_node (prog_table p) (prog_procs p) r u"
    and res: "(FunctionResult r, ctx) ∈ sol_vars (declared_global p) p"
    unfolding live_unknowns_def by blast
  have "prog_node (prog_table p) (prog_procs p) r k"
    using prog_node_calls[OF wf lr] e by (simp add: prog_cfg_def)
  then have k: "(k, ctx) ∈ sol_vars (declared_global p) p"
    using live_unknowns_of_result[OF wf solves _ res] live_unknowns_sub by blast
  from s have nb: "entry_of (declared_global p) p (call_info_of (CallEdge dst fs as) q)
              (dg_local (sol_env (declared_global p) p (Inl (u, ctx))))
              (sol_global (declared_global p) p) ≠ Bot"
    and c': "ctx' = ctx_succ (declared_global p) p u ctx (CallEdge dst fs as) q"
    by (auto simp: live_succ_def split: if_splits)
  have rq: "(FunctionResult q, ctx') ∈ sol_vars (declared_global p) p"
    unfolding c' by (rule sol_vars_callee_result[OF solves k e nb])
  have "prog_node (prog_table p) (prog_procs p) q (FunctionEntry q)"
    using prog_node_callee_entry[OF wf] e by (simp add: prog_cfg_def)
  then show "(FunctionEntry q, ctx') ∈ live_unknowns p"
    by (rule live_unknowns_of_result[OF wf solves _ rq])
qed

lemma root_live_key:
  assumes wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
```

```
  shows "root_query p ∈ live_unknowns p"
proof -
  have r: "(FunctionResult prog_main_name, root_ctx) ∈ sol_vars (declared_global p) p"
    using pp_routed[OF solves] by (simp add: root_query_def prog_cfg_def)
  show ?thesis
    using live_unknowns_of_result[OF wf solves prog_node_result[OF prog_node_main_entry[OF wf]] r]
    by (simp add: root_query_def prog_cfg_def)
qed
```

The entry-state endpoint over the live keys

The routed soundness statement restated over `live_unknowns`, whose coverage needs no premise beyond termination, and carried to the published table and the source-level endpoints under the functional context policy.

```
context
  fixes p :: imp_prog
begin

interpretation dg_base: analysis_contract "analysis_spec (declared_global p) p"
    "λd e. pgam p d e" "declared_global p"
  by (rule place_contract)
```

Establishes `routed_analysis` over the live keys, for any relation admitting call contexts. Beyond termination and well-formedness a policy owes only its own two facts: an admitted context is the one this pipeline routes the entered state to, and every concrete call at a live key is admitted somewhere. `live_unknowns_cover` supplies callee-entry membership.

```
lemma routed_analysis_from_live_unknowns:
  fixes adm :: "'c context_policy"
  assumes wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
    and cover_R: "⋀u ctx dst pars args q cont s ctx'. (u, ctx) ∈ live_unknowns p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ ctx' ∈ adm u ctx (call_info_of (CallEdge dst pars args) q) s
              (call_enter (declared_global p) (CallEdge dst pars args) s)
        ⟹ route (declared_global p) u ctx
              (entered p (call_info_of (CallEdge dst pars args) q)
                 (dg_local (sol_env (declared_global p) p (Inl (u, ctx)))))
              (CallEdge dst pars args) = ctx'"
    and total_R: "⋀u ctx dst pars args q cont s. (u, ctx) ∈ live_unknowns p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ s ∈ pgam p (dg_local (sol_env (declared_global p) p (Inl (u, ctx)))) (gsol p)
        ⟹ adm u ctx (call_info_of (CallEdge dst pars args) q) s
                     (call_enter (declared_global p) (CallEdge dst pars args) s) ≠ {}"
  shows "routed_analysis (analysis_spec (declared_global p) p) (λd e. pgam p d e)
     (declared_global p) (prog_cfg p) buffer_key global_of
     (route (declared_global p)) Bot (Lifted init_st)
     (place_rg (declared_global p) (Lifted init_st))
     (sol_env (declared_global p) p) (live_unknowns p) (root_query p) seed (λd. d = Bot) adm
     (λd e. map_lift (rd (declared_global p)) (place_cmb p d e))
     γ_V empty_V classify"
proof (unfold_locales, goal_cases CmbWf ExtraWf FinE PP SgCov SgUncov Fwd FinC CallsUnique
    SeedUnknown SeedNeGlobal IsBotBot IsBotSound ResolveSound EnterCover EnterTotal CombFwd GammaRd
    EmptyExact ClProved ClRefuted VarsFin)
  case CmbWf show ?case by (rule sp_wf_routed_call_program[OF dg_spec_wf_analysis_spec])
next
  case ExtraWf then show ?case by (rule sp_wf_routed_entry_seed_programs)
next
  case FinE show ?case unfolding prog_cfg_def by (simp add: compile_prog_finite)
next
  case PP show ?case
    by (rule post_bounded_subset[OF post_bounded_of_part_post_solution[OF pp_routed[OF solves]]
          root_live_key[OF wf solves] live_unknowns_sub])
next
  case (SgCov v c) then show ?case by simp
next
  case (SgUncov v c) then show ?case by simp
next
  case (Fwd u a v c)
  show ?case
    by (rule ctx_vars_cover_live_edgeD[OF live_unknowns_cover[OF wf solves] Fwd(1) _ Fwd(3)]) simp
next
  case FinC show ?case unfolding prog_cfg_def by (simp add: compile_prog_finite)
next
  case CallsUnique show ?case
    unfolding calls_source_unique_def prog_cfg_def
    using compile_prog_calls_source_unique by blast
```

```
next
  case (SeedUnknown q ctx) show ?case by (rule seed_ne_buffer_key)
next
  case (SeedNeGlobal q ctx v) show ?case by (rule seed_ne_global_of)
next  case IsBotBot show ?case by simp
next
  case (IsBotSound d g') then show ?case by simp
next
  case (ResolveSound u ctx dst pars args q cont s)
  then show ?case unfolding prog_cfg_def by (simp add: compile_prog_finite)
next
  case (EnterCover u ctx dst pars args q cont s ctx')
  let ?ci = "call_info_of (CallEdge dst pars args) q"
  let ?caller = "dg_local (sol_env (declared_global p) p (Inl (u, ctx)))"
  let ?g = "genv global_of (sol_env (declared_global p) p)"
  let ?ent = "entered p ?ci ?caller"
  have sin: "s ∈ pgam p ?caller (gsol p)"
    using EnterCover(3) by (simp add: sol_global_def)
  have cov: "entry_pairs_cover (λd. pgam p d (gsol p)) s
      (call_enter (declared_global p) (CallEdge dst pars args) s)
      [entry_alt (declared_global p) p ?ci ?caller (gsol p)]"
  proof (rule entry_cover[OF solves _ EnterCover(2) sin])
    show "(cont, ctx) ∈ sol_vars (declared_global p) p"
      using ctx_vars_cover_live_combineD[OF live_unknowns_cover[OF wf solves] EnterCover(1,2)]
        live_unknowns_sub by blast
  qed
  obtain c e where ce: "entry_alt (declared_global p) p ?ci ?caller (gsol p) = (c, e)"
    by (cases "entry_alt (declared_global p) p ?ci ?caller (gsol p)")
  have nbE: "?ent ≠ Bot"
  proof
    assume "?ent = Bot"
    with cov ce show False by (simp add: entry_pairs_cover_def entry_of_def place_cmb_bot)
  qed
  have req: "route (declared_global p) u ctx ?ent (CallEdge dst pars args) = ctx'"
    by (rule cover_R[OF EnterCover(1,2,4)])
  have covE: "(FunctionEntry q, ctx') ∈ live_unknowns p"
    by (rule ctx_vars_cover_live_enterD[OF live_unknowns_cover[OF wf solves] EnterCover(1,2)])
       (use nbE req in ‹simp add: live_succ_def ctx_succ_def›)
  obtain pub where runs: "enter_runs (enter♯ (analysis_spec (declared_global p) p) ?ci)
      (mk_dg_man ?caller global_of) (sol_env (declared_global p) p)
      [entry_alt (declared_global p) p ?ci ?caller ?g] pub"
    using place_enter_runs by blast
  obtain deps where deps: "enter_deps (enter♯ (analysis_spec (declared_global p) p) ?ci)
      (mk_dg_man ?caller global_of) (sol_env (declared_global p) p)
      [entry_alt (declared_global p) p ?ci ?caller ?g] deps"
    using place_enter_deps by blast
  have ce': "entry_alt (declared_global p) p ?ci ?caller ?g = (c, e)"
    using ce by (simp add: sol_global_def)
  have sc: "s ∈ pgam p c ?g"
    and ec: "call_enter (declared_global p) (CallEdge dst pars args) s ∈ pgam p e ?g"
    using cov ce by (simp_all add: entry_pairs_cover_def sol_global_def)
  have re: "route (declared_global p) u ctx e (CallEdge dst pars args) = ctx'"
    using req ce by (simp add: entry_of_def)
  show ?case
    using runs deps sc ec re covE unfolding ce' by fastforce
next
  case (EnterTotal u ctx dst pars args q cont s)
  then show ?case by (intro total_R) (simp_all add: sol_global_def)
next
  case (CombFwd cl c1 dst pars args q cont)
  show ?case
    by (rule ctx_vars_cover_live_combineD[OF live_unknowns_cover[OF wf solves] CombFwd(1,2)])
next
  case (GammaRd d g') show ?case by simp
next
  case (EmptyExact v) then show ?case by (rule empty_V_soundD)
next
  case (ClProved c d s) then show ?case by (rule classify_proved)
next
  case (ClRefuted c d s) then show ?case by (rule classify_refuted)
next
  case VarsFin show ?case
    by (rule finite_subset[OF live_unknowns_sub vars_finite_of_terminates[OF solves]])
qed
```

The reader over the live keys agrees with the published one on them and describes nothing elsewhere, so a

bound proved for it is a bound on the published table.

```
lemma gamma_live_reader_le:
  "pgam p (solved_local_reader (live_unknowns p) (sol_env (declared_global p) p) (Inl (v, ctx)))
       (gsol p)
     ⊆ pgam p (reader (declared_global p) p (Inl (v, ctx))) (gsol p)"
proof (cases "(v, ctx) ∈ live_unknowns p")
  case True
  then have "(v, ctx) ∈ sol_vars (declared_global p) p" by (rule subsetD[OF live_unknowns_sub])
  with True show ?thesis by (simp add: reader_def)
next
  case False then show ?thesis by (simp add: place_cmb_bot)
qed

theorem activation_collect_sound_live_unknowns:
  fixes adm :: "'c context_policy"
  assumes wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
    and cover_R: "⋀u ctx dst pars args q cont s ctx'. (u, ctx) ∈ live_unknowns p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ ctx' ∈ adm u ctx (call_info_of (CallEdge dst pars args) q) s
              (call_enter (declared_global p) (CallEdge dst pars args) s)
        ⟹ route (declared_global p) u ctx
              (entered p (call_info_of (CallEdge dst pars args) q)
                 (dg_local (sol_env (declared_global p) p (Inl (u, ctx)))))
              (CallEdge dst pars args) = ctx'"
    and total_R: "⋀u ctx dst pars args q cont s. (u, ctx) ∈ live_unknowns p
        ⟹ (u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)
        ⟹ s ∈ pgam p (dg_local (sol_env (declared_global p) p (Inl (u, ctx)))) (gsol p)
        ⟹ adm u ctx (call_info_of (CallEdge dst pars args) q) s
                     (call_enter (declared_global p) (CallEdge dst pars args) s) ≠ {}"
  shows "𝒜declared_global p,adm,root_ctx,prog_cfg p,cinit_stores (declared_global p) v ctx
          ⊆ pgam p (reader (declared_global p) p (Inl (v, ctx))) (gsol p)"
proof -
  interpret live: routed_analysis "analysis_spec (declared_global p) p" "λd e. pgam p d e"
      "declared_global p" "prog_cfg p" buffer_key global_of
      "route (declared_global p)" Bot
        "Lifted init_st" "place_rg (declared_global p) (Lifted init_st)"
      "sol_env (declared_global p) p" "live_unknowns p" "root_query p" seed "λd. d = Bot" adm
      "λd e. map_lift (rd (declared_global p)) (place_cmb p d e)"
      γV emptyV classify
    by (rule routed_analysis_from_live_unknowns[where adm = adm, OF wf solves cover_R total_R])
  have entry_cov: "(cfg_entry (prog_cfg p), root_ctx) ∈ live_unknowns p"
    by (rule ctx_vars_cover_live_entryD[OF live_unknowns_cover[OF wf solves]])
  have init_env_le:
    "init_env (declared_global p) p ≤ genv global_of (sol_env (declared_global p) p)"
    using init_env_le_sol[OF solves subsetD[OF live_unknowns_sub entry_cov]
      live.pp_entry_init_global_bound[OF entry_cov]]
    by (simp add: sol_global_def)
  have "𝒜declared_global p,adm,root_ctx,prog_cfg p,cinit_stores (declared_global p) v ctx
        ⊆ pgam p (solved_local_reader (live_unknowns p) (sol_env (declared_global p) p) (Inl (v, ctx)))
            (gsol p)"
    using live.activation_collect_dg_sound
      [OF entry_cov _ init_env_le, of "cinit_stores (declared_global p)" v ctx] cinit_le_init
    by (simp add: sol_global_def)
  then show ?thesis using gamma_live_reader_le by blast
qed
```

A policy whose route ignores the entered state — a call string, or the unit context — admits exactly the context it computes, so both of its facts are immediate.

```
theorem fun_route_activation_collect_sound_of_terminates:
  fixes ctx_fun :: "cfg_node ⇒ 'c ⇒ store ⇒ 'c"
  assumes route_const: "⋀u ctx d ca s. route (declared_global p) u ctx d ca = ctx_fun u ctx s"
    and wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
  shows "𝒜declared_global p,context_policy_of_fun ctx_fun,root_ctx,          prog_cfg p,cinit_stores (declared_global p)
v ctx
          ⊆ pgam p (reader (declared_global p) p (Inl (v, ctx))) (gsol p)"
proof (rule activation_collect_sound_live_unknowns[OF wf solves])
  fix u ctx dst pars args q cont and s :: store and ctx'
  assume "(u, ctx) ∈ live_unknowns p"
    and "(u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)"
    and Rc: "ctx' ∈ context_policy_of_fun ctx_fun u ctx (call_info_of (CallEdge dst pars args) q) s
              (call_enter (declared_global p) (CallEdge dst pars args) s)"
  show "route (declared_global p) u ctx
          (entered p (call_info_of (CallEdge dst pars args) q)
             (dg_local (sol_env (declared_global p) p (Inl (u, ctx)))))
          (CallEdge dst pars args) = ctx'"
```

```
    using Rc
    by (simp add:
      route_const[of _ _ _ _ "call_enter (declared_global p) (CallEdge dst pars args) s"])
next
  fix u ctx dst pars args q cont and s :: store
  show "context_policy_of_fun ctx_fun u ctx (call_info_of (CallEdge dst pars args) q) s
              (call_enter (declared_global p) (CallEdge dst pars args) s) ≠ {}"
    by simp
qed
```

The same fact read off the published surface, and what it yields at a program point, at a check and at a source run. A store reaching a point lies in one of the point's contexts, so the point is bounded by the union of the states published there. At the unit policy the union has the one member `()`.

```
theorem fun_route_state_at_sound:
  fixes ctx_fun :: "cfg_node ⇒ 'c ⇒ store ⇒ 'c"
  assumes route_const: "⋀u ctx d ca s. route (declared_global p) u ctx d ca = ctx_fun u ctx s"
    and wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
  shows "𝒜declared_global p,context_policy_of_fun ctx_fun,root_ctx,          prog_cfg p,cinit_stores (declared_global p)
v ctx
          ⊆ γV (state_at (declared_global p) p ctx v)"
proof -
  have "𝒜declared_global p,context_policy_of_fun ctx_fun,root_ctx,          prog_cfg p,cinit_stores (declared_global p)
v ctx
       ⊆ gamma_lift γV (lookup_table (result (declared_global p) p) v ctx)"
    using fun_route_activation_collect_sound_of_terminates[OF route_const wf solves]
    unfolding gamma_reader_eq_lookup .
  then show ?thesis
    by (auto simp: state_at_unfold bot_state_empty split: lifted.splits)
qed

theorem fun_route_result_node_sound:
  fixes ctx_fun :: "cfg_node ⇒ 'c ⇒ store ⇒ 'c"
  assumes route_const: "⋀u ctx d ca s. route (declared_global p) u ctx d ca = ctx_fun u ctx s"
    and wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
  shows "𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v
         ⊆ (⋃ctx. γV (state_at (declared_global p) p ctx v))"
  using fun_route_node_collect_eq_Union[where ctx_fun = ctx_fun and p = p and v = v]
    fun_route_state_at_sound[OF route_const wf solves]
  by (simp add: SUP_mono')

theorem fun_route_report_proved_sound:
  fixes ctx_fun :: "cfg_node ⇒ 'c ⇒ store ⇒ 'c"
  assumes route_const: "⋀u ctx d ca s. route (declared_global p) u ctx d ca = ctx_fun u ctx s"
    and wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
    and mem: "(v, c, Check_Proved) ∈ set (report (declared_global p) p ctx)"
  shows "∀s ∈ 𝒜declared_global p,context_policy_of_fun ctx_fun,root_ctx,               prog_cfg p,cinit_stores (declared_gl
v ctx. truthy (⟦c⟧e s)"
proof -
  have fin: "finite (intra (prog_cfg p))"
    unfolding prog_cfg_def by (simp add: compile_prog_finite)
  show ?thesis
    by (rule classify_checks_proved_sound
          [where g = "prog_cfg p" and env = "state_at (declared_global p) p ctx"
             and classify = classify and γS = γV
             and reach = "λv. 𝒜declared_global p,context_policy_of_fun ctx_fun,root_ctx,                                prog_cf
v ctx",
           OF fin _ classify_proved fun_route_state_at_sound[OF route_const wf solves]])
       (use mem in ‹simp add: report_def state_at_def analysis_surface.report_def›)
qed

theorem fun_route_report_refuted_sound:
  fixes ctx_fun :: "cfg_node ⇒ 'c ⇒ store ⇒ 'c"
  assumes route_const: "⋀u ctx d ca s. route (declared_global p) u ctx d ca = ctx_fun u ctx s"
    and wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
    and mem: "(v, c, Check_Refuted) ∈ set (report (declared_global p) p ctx)"
  shows "∀s ∈ 𝒜declared_global p,context_policy_of_fun ctx_fun,root_ctx,               prog_cfg p,cinit_stores (declared_gl
v ctx. ¬ truthy (⟦c⟧e s)"
proof -
  have fin: "finite (intra (prog_cfg p))"
    unfolding prog_cfg_def by (simp add: compile_prog_finite)
  show ?thesis
    by (rule classify_checks_refuted_sound
          [where g = "prog_cfg p" and env = "state_at (declared_global p) p ctx"
             and classify = classify and γS = γV
             and reach = "λv. 𝒜declared_global p,context_policy_of_fun ctx_fun,root_ctx,                                prog_cf
```

```
v ctx",
          OF fin _ classify_refuted fun_route_state_at_sound[OF route_const wf solves]])
      (use mem in ‹simp add: report_def state_at_def analysis_surface.report_def›)
qed

theorem fun_route_source_sound:
  fixes ctx_fun :: "cfg_node ⇒ 'c ⇒ store ⇒ 'c" and s0 s :: store
  assumes route_const: "⋀u ctx d ca s. route (declared_global p) u ctx d ca = ctx_fun u ctx s"
    and wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
    and s0: "s0 ∈ cinit_stores (declared_global p)"
    and run: "declared_global p, prog_table p ⊢ (main_body (prog_table p), s0, [])
                →p* (residual, s, frs)"
  shows "∃v stk ctx. prog_table p, prog_cfg p, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
         ∧ s ∈ γV (state_at (declared_global p) p ctx v)"
proof -
  have cfg_eq: "prog_cfg p = compile_prog (prog_table p) (prog_procs p)"
    by (rule prog_cfg_def)
  have "∃v stk. prog_table p, prog_cfg p, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
        ∧ s ∈ (⋃ctx. γV (state_at (declared_global p) p ctx v))"
    unfolding cfg_eq
    by (rule source_sound_from_node_collect_cap[OF wf s0 run])
       (use fun_route_result_node_sound[OF route_const wf solves] in ‹simp add: cfg_eq›)
  then show ?thesis by blast
qed
```

Entry-state routing

Instantiates the live-key endpoints at `admitted_contexts`, the policy that routes a call by its entered state. Its two policy facts hold by construction, so the results need only termination and a well-formed program.

```
lemma entry_state_routed_analysis_from_live_unknowns:
  assumes wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
  shows "routed_analysis (analysis_spec (declared_global p) p) (λd e. pgam p d e)
     (declared_global p) (prog_cfg p) buffer_key global_of
     (route (declared_global p)) Bot (Lifted init_st)
     (place_rg (declared_global p) (Lifted init_st))
     (sol_env (declared_global p) p) (live_unknowns p) (root_query p) seed (λd. d = Bot)
     (admitted_contexts (declared_global p) p)
     (λd e. map_lift (rd (declared_global p)) (place_cmb p d e))
    γV emptyV classify"
proof (rule routed_analysis_from_live_unknowns[OF wf solves])
  fix u ctx dst pars args q cont and s :: store and ctx'
  assume "(u, ctx) ∈ live_unknowns p"
    and "(u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)"
    and Rc: "ctx' ∈ admitted_contexts (declared_global p) p u ctx
               (call_info_of (CallEdge dst pars args) q) s
               (call_enter (declared_global p) (CallEdge dst pars args) s)"
  let ?ci = "call_info_of (CallEdge dst pars args) q"
  let ?caller = "dg_local (sol_env (declared_global p) p (Inl (u, ctx)))"
  let ?ent = "entered p ?ci ?caller"
  from Rc[unfolded admitted_contexts_alt] obtain cont' entry
    where mem: "(cont', entry) ∈ set [entry_alt (declared_global p) p ?ci ?caller (gsol p)]"
      and req0: "ctx' = route (declared_global p) u ctx entry
                   (CallEdge (ci_dst ?ci) (ci_formals ?ci) (ci_args ?ci))"
    by (rule routed_entry_context_relE)
  have "entry = ?ent" using mem by (simp add: entry_of_def) (metis snd_conv)
  then show "route (declared_global p) u ctx ?ent (CallEdge dst pars args) = ctx'"
    using req0 by simp
next
  fix u ctx dst pars args q cont and s :: store
  assume uL: "(u, ctx) ∈ live_unknowns p"
    and ce: "(u, CallEdge dst pars args, FunctionEntry q, cont) ∈ calls (prog_cfg p)"
    and sin: "s ∈ pgam p (dg_local (sol_env (declared_global p) p (Inl (u, ctx)))) (gsol p)"
  have cont: "(cont, ctx) ∈ sol_vars (declared_global p) p"
    using ctx_vars_cover_live_combineD[OF live_unknowns_cover[OF wf solves] uL ce]
      live_unknowns_sub by blast
  show "admitted_contexts (declared_global p) p u ctx
                (call_info_of (CallEdge dst pars args) q) s
                (call_enter (declared_global p) (CallEdge dst pars args) s) ≠ {}"
    unfolding admitted_contexts_alt
    by (rule routed_entry_context_rel_total)
       (use entry_cover[OF solves cont ce sin]
         in ‹simp add: sol_global_def del: declared_global_iff›)
qed

theorem entry_state_activation_collect_sound_of_terminates:
```

```
  assumes wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
  shows "𝒜declared_global p,admitted_contexts (declared_global p) p,           root_ctx,prog_cfg p,cinit_stores (declared_glob
v ctx
          ⊆ pgam p (reader (declared_global p) p (Inl (v, ctx))) (gsol p)"
proof -
  interpret live: routed_analysis "analysis_spec (declared_global p) p" "λd e. pgam p d e"
      "declared_global p" "prog_cfg p" buffer_key global_of
      "route (declared_global p)" Bot
        "Lifted init_st" "place_rg (declared_global p) (Lifted init_st)"
      "sol_env (declared_global p) p" "live_unknowns p" "root_query p" seed "λd. d = Bot"
      "admitted_contexts (declared_global p) p"
      "λd e. map_lift (rd (declared_global p)) (place_cmb p d e)"
      γV emptyV classify
    by (rule entry_state_routed_analysis_from_live_unknowns[OF wf solves])
  have entry_cov: "(cfg_entry (prog_cfg p), root_ctx) ∈ live_unknowns p"
    by (rule ctx_vars_cover_live_entryD[OF live_unknowns_cover[OF wf solves]])
  have init_env_le:
    "init_env (declared_global p) p ≤ genv global_of (sol_env (declared_global p) p)"
    using init_env_le_sol[OF solves subsetD[OF live_unknowns_sub entry_cov]
      live.pp_entry_init_global_bound[OF entry_cov]]
    by (simp add: sol_global_def)
  have "𝒜declared_global p,admitted_contexts (declared_global p) p,          root_ctx,prog_cfg p,cinit_stores (declared_global
v ctx
       ⊆ pgam p (solved_local_reader (live_unknowns p) (sol_env (declared_global p) p) (Inl (v, ctx)))
           (gsol p)"
    using live.activation_collect_dg_sound
        [OF entry_cov _ init_env_le, of "cinit_stores (declared_global p)" v ctx] cinit_le_init
    by (simp add: sol_global_def)
  then show ?thesis using gamma_live_reader_le by blast
qed

corollary entry_state_lookup_sound_of_terminates:
  assumes wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
  shows "𝒜declared_global p,admitted_contexts (declared_global p) p,           root_ctx,prog_cfg p,cinit_stores (declared_glob
v ctx
          ⊆ gamma_lift γV (lookup_table (result (declared_global p) p) v ctx)"
  using entry_state_activation_collect_sound_of_terminates[OF wf solves]
  unfolding gamma_reader_eq_lookup .

lemma entry_state_has_context_of_terminates:
  assumes wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
    and t: "t ∈ 𝒯declared_global p,prog_cfg p,cinit_stores (declared_global p)"
  shows "∃c. activation_context_rel (declared_global p) (admitted_contexts (declared_global p) p)
              root_ctx (prog_cfg p) t c"
proof -
  interpret live: routed_analysis "analysis_spec (declared_global p) p" "λd e. pgam p d e"
      "declared_global p" "prog_cfg p" buffer_key global_of
      "route (declared_global p)" Bot
        "Lifted init_st" "place_rg (declared_global p) (Lifted init_st)"
      "sol_env (declared_global p) p" "live_unknowns p" "root_query p" seed "λd. d = Bot"
      "admitted_contexts (declared_global p) p"
      "λd e. map_lift (rd (declared_global p)) (place_cmb p d e)"
      γV emptyV classify
    by (rule entry_state_routed_analysis_from_live_unknowns[OF wf solves])
  have entry_cov: "(cfg_entry (prog_cfg p), root_ctx) ∈ live_unknowns p"
    by (rule ctx_vars_cover_live_entryD[OF live_unknowns_cover[OF wf solves]])
  have init_env_le:
    "init_env (declared_global p) p ≤ genv global_of (sol_env (declared_global p) p)"
    using init_env_le_sol[OF solves subsetD[OF live_unknowns_sub entry_cov]
      live.pp_entry_init_global_bound[OF entry_cov]]
    by (simp add: sol_global_def)
  show ?thesis
    using live.routed_valid_activation_trace_has_context[OF entry_cov _ init_env_le t] cinit_le_init
    by simp
qed

theorem entry_state_node_collect_eq_Union_of_terminates:
  assumes wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
  shows "𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v
         = (⋃ctx. 𝒜declared_global p,admitted_contexts (declared_global p) p,                    root_ctx,prog_cfg p,cini
v ctx)"
  by (rule node_collect_eq_Union_activation_of_has_context)
     (rule entry_state_has_context_of_terminates[OF wf solves])
```

The entry-state policy in one statement: the solve is fixed first, the policy is read off it, and that policy is total on valid traces and covered by the solved states. This order of quantifiers is why reading contexts off the result

is not circular.

```
corollary entry_state_policy_of_terminates:
  assumes wf: "wf_program_compile_input p" and solves: "terminates (declared_global p) p"
  shows "∀t ∈ 𝒯declared_global p,prog_cfg p,cinit_stores (declared_global p)·
           ∃c. activation_context_rel (declared_global p) (admitted_contexts (declared_global p) p)
                 root_ctx (prog_cfg p) t c"
    and "𝒜declared_global p,admitted_contexts (declared_global p) p,           root_ctx,prog_cfg p,cinit_stores (declared_glob
v ctx
           ⊆ pgam p (reader (declared_global p) p (Inl (v, ctx))) (gsol p)"
  using entry_state_has_context_of_terminates[OF wf solves]
    entry_state_activation_collect_sound_of_terminates[OF wf solves]
  by blast+

end

end

end
```

Chapter 10

Voblint_Nonrelational

10.1 Abstract_Arithmetic

```
theory Abstract_Arithmetic
  imports
    "Voblint_Domain.Three_Valued" "Voblint_Domain.Forward_Domain"
    "Voblint_Domain.Numeric_Queries"
    "Voblint_Domain.Nonrelational_State"
begin
```

Generic expression soundness

Sign, Interval, Parity, and Congruence each prove an `aval_<dom>_sound` lemma with the identical shape and proof script: structural induction on `exp`, discharged by the same per-operator soundness facts. Arithmetic (`plus_sound`/`minus_sound`/`times_sound`) and comparison/truthiness (`lt`/`eqb`/`tobool`) are genuinely domain-specific – each domain's own case-split proof over its own representation – so this locale does not try to share those. What it shares is the one induction that combines them, mirroring Goblint's `base.ml` expression evaluator: `Base` only ever calls `ID.add`/`ID.lt`/`ID.to_bool` and never recomputes a comparison's precision itself, so `ev` here only ever calls `lt`/`eqb`/`tobool` and never recomputes what a domain already knows about its own relation.

`ev`, `lit`, `lt`, `eqb`, and `tobool` are locale parameters, not derived: interpreting this locale at a domain's own `aval_<dom>`, literal-embedding function, and comparison/truthiness queries keeps every existing call site of `aval_<dom>` untouched – the shared `aval_dom_sound` lemma below is stated over the same externally-fixed functions, not a locale-internal reconstruction of them.

A comparison or logical result is always the concrete C truthiness value `0`/`1`. Goblint's integer domains move between the two readings with `of_bool` and `to_bool`; here the three-valued answer of a query is read back as a value by `of_bool_option`: `Some True` is `lit 1`, `Some False` is `lit 0`, and an unknown answer joins both — exactly Goblint's `id_binary_pred`/`id_unary_log`/`id_binary_log` pattern. A negated relation negates the query answer, and `And`/`Or` combine their operands' truthiness with `and_opt`/`or_opt`, which settle on one definite annihilating operand, mirroring `id_binary_log`'s own short-circuit.

Each operand is queried through `lt`/`eqb`/`tobool` only when it is not itself `is_empty`: an `is_empty` operand denotes an unreachable value (its concretization is empty), so the whole comparison collapses to `bot` rather than querying a value with no witness to reason about. This guard is not optional bookkeeping – `bot` is every domain's own least element, so a comparison against it can widen to conflicting query answers as the *other* operand widens; routing it through `bot` directly, before `lt`/ `eqb`/`tobool` ever see it, is what keeps `aval_dom_mono` provable without demanding an impossible monotonicity obligation from those queries at their own domain's bottom.

```
fun of_bool_option :: "(int ⇒ 'a::sup) ⇒ bool option ⇒ 'a" where
  "of_bool_option lit (Some b) = lit (if b then 1 else 0)"
| "of_bool_option lit None = lit 0 ⊔ lit 1"
```

`sound_arith_ops` pins an evaluator `ev` down by its equations: arithmetic through the supplied operations, comparisons through `lt`/`eqb` and `of_bool_option` behind the emptiness guard. Given sound operations and queries, the locale proves the evaluator sound.

```
locale sound_arith_ops =
  sound_truth_test tobool + sound_numeric_queries lt eqb
  for ev :: "exp ⇒ (vname ⇒ 'a::numeric_domain) ⇒ 'a" ("⟦_⟧♯")
    and lit :: "int ⇒ 'a"
    and pls :: "'a ⇒ 'a ⇒ 'a"
    and mns :: "'a ⇒ 'a ⇒ 'a"
    and tms :: "'a ⇒ 'a ⇒ 'a"
    and dvs :: "'a ⇒ 'a ⇒ 'a"
    and rem :: "'a ⇒ 'a ⇒ 'a"
    and lt :: "'a ⇒ 'a ⇒ bool option"
    and eqb :: "'a ⇒ 'a ⇒ bool option"
    and tobool :: "'a ⇒ bool option" +
  assumes ev_N[simp]: "⟦N n⟧♯ sigma = lit n"
    and ev_V[simp]: "⟦V x⟧♯ sigma = sigma x"
    and ev_Plus[simp]: "⟦Plus e1 e2⟧♯ sigma = pls (⟦e1⟧♯ sigma) (⟦e2⟧♯ sigma)"
    and ev_Minus[simp]: "⟦Minus e1 e2⟧♯ sigma = mns (⟦e1⟧♯ sigma) (⟦e2⟧♯ sigma)"
    and ev_Times[simp]: "⟦Times e1 e2⟧♯ sigma = tms (⟦e1⟧♯ sigma) (⟦e2⟧♯ sigma)"
    and ev_Div[simp]: "⟦Div e1 e2⟧♯ sigma = dvs (⟦e1⟧♯ sigma) (⟦e2⟧♯ sigma)"
    and ev_Mod[simp]: "⟦Mod e1 e2⟧♯ sigma = rem (⟦e1⟧♯ sigma) (⟦e2⟧♯ sigma)"
    and ev_Less[simp]: "⟦Less e1 e2⟧♯ sigma =
         (if is_empty (⟦e1⟧♯ sigma) ∨ is_empty (⟦e2⟧♯ sigma) then bot
          else of_bool_option lit (lt (⟦e1⟧♯ sigma) (⟦e2⟧♯ sigma)))"
    and ev_LessEq[simp]: "⟦LessEq e1 e2⟧♯ sigma =
         (if is_empty (⟦e1⟧♯ sigma) ∨ is_empty (⟦e2⟧♯ sigma) then bot
          else of_bool_option lit (map_option HOL.Not (lt (⟦e2⟧♯ sigma) (⟦e1⟧♯ sigma))))"
    and ev_Greater[simp]: "⟦Greater e1 e2⟧♯ sigma =
```

```
      (if is_empty (⟦e1⟧♯ sigma) ∨ is_empty (⟦e2⟧♯ sigma) then bot
       else of_bool_option lit (lt (⟦e2⟧♯ sigma) (⟦e1⟧♯ sigma)))"
  and ev_GreaterEq[simp]: "⟦GreaterEq e1 e2⟧♯ sigma =
      (if is_empty (⟦e1⟧♯ sigma) ∨ is_empty (⟦e2⟧♯ sigma) then bot
       else of_bool_option lit (map_option HOL.Not (lt (⟦e1⟧♯ sigma) (⟦e2⟧♯ sigma))))"
  and ev_NotEq[simp]: "⟦NotEq e1 e2⟧♯ sigma =
      (if is_empty (⟦e1⟧♯ sigma) ∨ is_empty (⟦e2⟧♯ sigma) then bot
       else of_bool_option lit (map_option HOL.Not (eqb (⟦e1⟧♯ sigma) (⟦e2⟧♯ sigma))))"
  and ev_Eq[simp]: "⟦exp.Eq e1 e2⟧♯ sigma =
      (if is_empty (⟦e1⟧♯ sigma) ∨ is_empty (⟦e2⟧♯ sigma) then bot
       else of_bool_option lit (eqb (⟦e1⟧♯ sigma) (⟦e2⟧♯ sigma)))"
  and ev_Not[simp]: "⟦exp.Not e⟧♯ sigma =
      (if is_empty (⟦e⟧♯ sigma) then bot
       else of_bool_option lit (map_option HOL.Not (tobool (⟦e⟧♯ sigma))))"
  and ev_And[simp]: "⟦And e1 e2⟧♯ sigma =
      (if is_empty (⟦e1⟧♯ sigma) ∨ is_empty (⟦e2⟧♯ sigma) then bot
       else of_bool_option lit (and_opt (tobool (⟦e1⟧♯ sigma)) (tobool (⟦e2⟧♯ sigma))))"
  and ev_Or[simp]: "⟦Or e1 e2⟧♯ sigma =
      (if is_empty (⟦e1⟧♯ sigma) ∨ is_empty (⟦e2⟧♯ sigma) then bot
       else of_bool_option lit (or_opt (tobool (⟦e1⟧♯ sigma)) (tobool (⟦e2⟧♯ sigma))))"
  and lit_sound[simp]: "n ∈ γ (lit n)"
  and plus_sound[intro]:
    "i ∈ γ (p::'a) ⟹ j ∈ γ q ⟹ i + j ∈ γ (pls p q)"
  and minus_sound[intro]:
    "i ∈ γ (p::'a) ⟹ j ∈ γ q ⟹ i - j ∈ γ (mns p q)"
  and times_sound[intro]:
    "i ∈ γ (p::'a) ⟹ j ∈ γ q ⟹ i * j ∈ γ (tms p q)"
  and div_sound[intro]:
    "i ∈ γ (p::'a) ⟹ j ∈ γ q ⟹ c_div i j ∈ γ (dvs p q)"
  and mod_sound[intro]:
    "i ∈ γ (p::'a) ⟹ j ∈ γ q ⟹ c_mod i j ∈ γ (rem p q)"
begin
```

Every comparison and connective reads its query answer back through `of_bool_option` behind the same emptiness guard. `of_bool_option_sound` and `of_bool_option_mono` discharge that shape once, so each induction case supplies only its own query facts.

```
lemma of_bool_option_sound:
  assumes "¬ E" and "⋀b. r = Some b ⟹ v = (if b then 1 else 0)" and "v ∈ {0, 1}"
  shows "v ∈ γ (if E then bot else of_bool_option lit r)"
  using assms
  by (cases r) (auto intro: gamma_sup_ub1[THEN subsetD] gamma_sup_ub2[THEN subsetD])

lemma aval_dom_sound:
  "s ∈ γ sigma ⟹ ⟦a⟧ₑ s ∈ γ (⟦a⟧♯ sigma)"
proof (induction a arbitrary: s sigma)
  case (Less e1 e2)
  have h1: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ sigma)" and h2: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ sigma)"
    using Less by simp_all
  show ?case unfolding ev_Less aval.simps
    by (rule of_bool_option_sound) (use h1 h2 is_empty_correct less_sound[OF _ h1 h2] in auto)
next
  case (LessEq e1 e2)
  have h1: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ sigma)" and h2: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ sigma)"
    using LessEq by simp_all
  show ?case unfolding ev_LessEq aval.simps
    by (rule of_bool_option_sound) (use h1 h2 is_empty_correct less_sound[OF _ h2 h1] in auto)
next
  case (Greater e1 e2)
  have h1: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ sigma)" and h2: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ sigma)"
    using Greater by simp_all
  show ?case unfolding ev_Greater aval.simps
    by (rule of_bool_option_sound) (use h1 h2 is_empty_correct less_sound[OF _ h2 h1] in auto)
next
  case (GreaterEq e1 e2)
  have h1: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ sigma)" and h2: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ sigma)"
    using GreaterEq by simp_all
  show ?case unfolding ev_GreaterEq aval.simps
    by (rule of_bool_option_sound) (use h1 h2 is_empty_correct less_sound[OF _ h1 h2] in auto)
next
  case (NotEq e1 e2)
  have h1: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ sigma)" and h2: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ sigma)"
    using NotEq by simp_all
  show ?case unfolding ev_NotEq aval.simps
    by (rule of_bool_option_sound) (use h1 h2 is_empty_correct eq_sound[OF _ h1 h2] in auto)
next
  case (Eq e1 e2)
```

```
  have h1: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ sigma)" and h2: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ sigma)"
    using Eq by simp_all
  show ?case unfolding ev_Eq aval.simps
    by (rule of_bool_option_sound) (use h1 h2 is_empty_correct eq_sound[OF _ h1 h2] in auto)
next
  case (Not e)
  have h: "⟦e⟧ₑ s ∈ γ (⟦e⟧♯ sigma)" using Not by simp
  show ?case unfolding ev_Not aval.simps
    by (rule of_bool_option_sound) (use h is_empty_correct tobool_sound[OF _ h] in auto)
next
  case (And e1 e2)
  have h1: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ sigma)" and h2: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ sigma)"
    using And by simp_all
  show ?case unfolding ev_And aval.simps
    by (rule of_bool_option_sound)
       (use h1 h2 is_empty_correct
          and_opt_sound[OF _ tobool_sound[OF _ h1] tobool_sound[OF _ h2]] in auto)
next
  case (Or e1 e2)
  have h1: "⟦e1⟧ₑ s ∈ γ (⟦e1⟧♯ sigma)" and h2: "⟦e2⟧ₑ s ∈ γ (⟦e2⟧♯ sigma)"
    using Or by simp_all
  show ?case unfolding ev_Or aval.simps
    by (rule of_bool_option_sound)
       (use h1 h2 is_empty_correct
          or_opt_sound[OF _ tobool_sound[OF _ h1] tobool_sound[OF _ h2]] in auto)
qed (auto dest: gamma_stateD)

end
```

The expression domain extends the truth test and the numeric queries, whose laws it assumes, and derives its evaluator's soundness by induction. An interpretation of `sound_arith_ops` therefore registers the whole forward interface once, and a later `sound_refinement`, `sound_minmax_ops` or check-layer interpretation over the same operations does not ask for these laws again.

```
sublocale sound_arith_ops ⊆ sound_evaluator gamma_state ev
  by unfold_locales (rule aval_dom_sound)
```

Monotonicity needs strictly more than soundness does, and one domain can supply the first without the second: the composite integer domain's arithmetic is monotone only away from `Refine_Fixpoint`, while its soundness holds at every refinement mode. Splitting the two apart is what lets that domain interpret the shared induction twice — unconditionally for `aval_dom_sound`, under its mode side condition for `aval_dom_mono` — rather than repeat both proofs by hand.

```
locale mono_arith_ops = sound_arith_ops + mono_truth_test tobool +
  assumes plus_mono[intro]:
      "p1 ≤ p2 ⟹ q1 ≤ q2 ⟹ pls p1 q1 ≤ pls p2 q2"
    and minus_mono[intro]:
      "p1 ≤ p2 ⟹ q1 ≤ q2 ⟹ mns p1 q1 ≤ mns p2 q2"
    and times_mono[intro]:
      "p1 ≤ p2 ⟹ q1 ≤ q2 ⟹ tms p1 q1 ≤ tms p2 q2"
    and div_mono[intro]:
      "p1 ≤ p2 ⟹ q1 ≤ q2 ⟹ dvs p1 q1 ≤ dvs p2 q2"
    and mod_mono[intro]:
      "p1 ≤ p2 ⟹ q1 ≤ q2 ⟹ rem p1 q1 ≤ rem p2 q2"
    and lt_mono:
      "¬ is_empty p1 ⟹ ¬ is_empty q1 ⟹ p1 ≤ p2 ⟹ q1 ≤ q2 ⟹
       lt p2 q2 = Some b ⟹ lt p1 q1 = Some b"
    and eqb_mono:
      "¬ is_empty p1 ⟹ ¬ is_empty q1 ⟹ p1 ≤ p2 ⟹ q1 ≤ q2 ⟹
       eqb p2 q2 = Some b ⟹ eqb p1 q1 = Some b"
begin

lemma of_bool_option_mono:
  assumes "E2 ⟹ E1" and "⋀b. ¬ E1 ⟹ r2 = Some b ⟹ r1 = Some b"
  shows "(if E1 then bot else of_bool_option lit r1) ≤ (if E2 then bot else of_bool_option lit r2)"
  using assms by (cases r1; cases r2) auto

lemma aval_dom_mono:
  "sigma1 ≤ sigma2 ⟹ ⟦a⟧♯ sigma1 ≤ ⟦a⟧♯ sigma2"
proof (induction a arbitrary: sigma1 sigma2)
  case (Less e1 e2)
  have p: "⟦e1⟧♯ sigma1 ≤ ⟦e1⟧♯ sigma2" and q: "⟦e2⟧♯ sigma1 ≤ ⟦e2⟧♯ sigma2"
    using Less by simp_all
  show ?case unfolding ev_Less
    by (rule of_bool_option_mono)
       (use is_empty_antimono[OF p] is_empty_antimono[OF q] lt_mono[OF _ _ p q] in auto)
```

```
next
  case (LessEq e1 e2)
  have p: "⟦e1⟧♯ sigma1 ≤ ⟦e1⟧♯ sigma2" and q: "⟦e2⟧♯ sigma1 ≤ ⟦e2⟧♯ sigma2"
    using LessEq by simp_all
  show ?case unfolding ev_LessEq
    by (rule of_bool_option_mono)
       (use is_empty_antimono[OF p] is_empty_antimono[OF q] lt_mono[OF _ _ q p] in auto)
next
  case (Greater e1 e2)
  have p: "⟦e1⟧♯ sigma1 ≤ ⟦e1⟧♯ sigma2" and q: "⟦e2⟧♯ sigma1 ≤ ⟦e2⟧♯ sigma2"
    using Greater by simp_all
  show ?case unfolding ev_Greater
    by (rule of_bool_option_mono)
       (use is_empty_antimono[OF p] is_empty_antimono[OF q] lt_mono[OF _ _ q p] in auto)
next
  case (GreaterEq e1 e2)
  have p: "⟦e1⟧♯ sigma1 ≤ ⟦e1⟧♯ sigma2" and q: "⟦e2⟧♯ sigma1 ≤ ⟦e2⟧♯ sigma2"
    using GreaterEq by simp_all
  show ?case unfolding ev_GreaterEq
    by (rule of_bool_option_mono)
       (use is_empty_antimono[OF p] is_empty_antimono[OF q] lt_mono[OF _ _ p q] in auto)
next
  case (NotEq e1 e2)
  have p: "⟦e1⟧♯ sigma1 ≤ ⟦e1⟧♯ sigma2" and q: "⟦e2⟧♯ sigma1 ≤ ⟦e2⟧♯ sigma2"
    using NotEq by simp_all
  show ?case unfolding ev_NotEq
    by (rule of_bool_option_mono)
       (use is_empty_antimono[OF p] is_empty_antimono[OF q] eqb_mono[OF _ _ p q] in auto)
next
  case (Eq e1 e2)
  have p: "⟦e1⟧♯ sigma1 ≤ ⟦e1⟧♯ sigma2" and q: "⟦e2⟧♯ sigma1 ≤ ⟦e2⟧♯ sigma2"
    using Eq by simp_all
  show ?case unfolding ev_Eq
    by (rule of_bool_option_mono)
       (use is_empty_antimono[OF p] is_empty_antimono[OF q] eqb_mono[OF _ _ p q] in auto)
next
  case (Not e)
  have p: "⟦e⟧♯ sigma1 ≤ ⟦e⟧♯ sigma2" using Not by simp
  show ?case unfolding ev_Not
    by (rule of_bool_option_mono) (use is_empty_antimono[OF p] tobool_mono[OF _ p] in auto)
next
  case (And e1 e2)
  have p: "⟦e1⟧♯ sigma1 ≤ ⟦e1⟧♯ sigma2" and q: "⟦e2⟧♯ sigma1 ≤ ⟦e2⟧♯ sigma2"
    using And by simp_all
  show ?case unfolding ev_And
    by (rule of_bool_option_mono)
       (use is_empty_antimono[OF p] is_empty_antimono[OF q]
          and_opt_mono[OF tobool_mono[OF _ p] tobool_mono[OF _ q]] in auto)
next
  case (Or e1 e2)
  have p: "⟦e1⟧♯ sigma1 ≤ ⟦e1⟧♯ sigma2" and q: "⟦e2⟧♯ sigma1 ≤ ⟦e2⟧♯ sigma2"
    using Or by simp_all
  show ?case unfolding ev_Or
    by (rule of_bool_option_mono)
       (use is_empty_antimono[OF p] is_empty_antimono[OF q]
          or_opt_mono[OF tobool_mono[OF _ p] tobool_mono[OF _ q]] in auto)
qed (auto simp add: le_funD)

end

sublocale mono_arith_ops ⊆ mono_evaluator gamma_state ev
  by unfold_locales (rule aval_dom_mono)

end
```

10.2 Special_Ops

```
theory Special_Ops
  imports "Voblint_Framework.Transfer_Algebra" "Voblint_Domain.Forward_Domain"
begin
```

Generic special-call dispatch

Every domain's `Nondet_Int`/`Min`/`Max` dispatch (`special_sign`, `special_ivl`, `special_parity`, ...) has the same shape: havoc to `top` for `Nondet_Int`, or apply a two-argument primitive to the evaluated operands for `Min`/`Max`. This theory proves that shape's soundness and monotonicity once, against an abstract pair of primitives (`special_min`/`special_max`) and an abstract expression evaluator, so each domain only has to supply its own minimum and maximum primitives and their soundness/monotonicity facts – the case-split dispatch and its proof are not repeated per domain.

`special_min`/`special_max` are bundled as a record rather than as two bare locale parameters so a concrete instance (e.g. `sign_special_ops`) is a first-class value, not just a locale interpretation.

```
record 'a special_ops =
  special_min :: "'a => 'a => 'a"
  special_max :: "'a => 'a => 'a"
```

`sound_minmax_ops` asks for a sound evaluator and primitives whose results contain `min`/`max` of any two described integers.

```
locale sound_minmax_ops = sound_evaluator gamma_state ev
  for ops :: "'a::numeric_domain special_ops"
    and ev  :: "exp => 'a abs_state => 'a" ("⟦_⟧♯") +
  assumes special_min_sound[intro]:
    "i ∈ γ p ⟹ j ∈ γ q ⟹ min i j ∈ γ (special_min ops p q)"
  assumes special_max_sound[intro]:
    "i ∈ γ p ⟹ j ∈ γ q ⟹ max i j ∈ γ (special_max ops p q)"
```

Monotonicity is a separate layer, as for the evaluator and the backward operations: soundness of the special calls needs none of it.

```
locale mono_minmax_ops = sound_minmax_ops + mono_evaluator gamma_state ev +
  assumes special_min_mono[intro]:
    "p1 ≤ p2 ⟹ q1 ≤ q2 ⟹ special_min ops p1 q1 ≤ special_min ops p2 q2"
  assumes special_max_mono[intro]:
    "p1 ≤ p2 ⟹ q1 ≤ q2 ⟹ special_max ops p1 q1 ≤ special_max ops p2 q2"

context sound_minmax_ops
begin

definition special_transfer ::
    "special_call => vname => (vname => 'a) => (vname => 'a)"
where
  "special_transfer sc x d =
     d(x := (case sc of
                Nondet_Int => top
              | Min a b => special_min ops (⟦a⟧♯ d) (⟦b⟧♯ d)
              | Max a b => special_max ops (⟦a⟧♯ d) (⟦b⟧♯ d)))"

lemma special_transfer_Nondet_Int [simp]:
  "special_transfer Nondet_Int x d = d(x := top)"
  unfolding special_transfer_def by simp

lemma special_transfer_Min [simp]:
  "special_transfer (Min a b) x d = d(x := special_min ops (⟦a⟧♯ d) (⟦b⟧♯ d))"
  unfolding special_transfer_def by simp

lemma special_transfer_Max [simp]:
  "special_transfer (Max a b) x d = d(x := special_max ops (⟦a⟧♯ d) (⟦b⟧♯ d))"
  unfolding special_transfer_def by simp
```

Each dispatch case writes one variable, so the whole proof is `Voblint_Framework.Transfer_Algebra`'s `gamma_state_upd` — a sound state updated at `x` with a value sound for the new abstract element stays sound — once the case supplies that the written value is in the concretization. Only the three `v ∈ gamma _` obligations are special-call specific; the per-variable reasoning is entirely `gamma_state_upd`'s.

```
lemma special_transfer_sound:
  assumes 𝒢: "s ∈ γ d" and sr: "special_result sc s v"
  shows "s(x := v) ∈ γ (special_transfer sc x d)"
proof -
```

```
  show ?thesis
  proof (cases sc)
    case Nondet_Int
    with sr 𝒢 show ?thesis by auto
  next
    case (Min a b)
    with sr have "v = min (⟦a⟧ₑ s) (⟦b⟧ₑ s)" by simp
    moreover from 𝒢 have "⟦a⟧ₑ s ∈ γ (⟦a⟧♯ d)" and "⟦b⟧ₑ s ∈ γ (⟦b⟧♯ d)"
      by (rule aval_abs_sound)+
    ultimately show ?thesis using Min 𝒢 by auto
  next
    case (Max a b)
    with sr have "v = max (⟦a⟧ₑ s) (⟦b⟧ₑ s)" by simp
    moreover from 𝒢 have "⟦a⟧ₑ s ∈ γ (⟦a⟧♯ d)" and "⟦b⟧ₑ s ∈ γ (⟦b⟧♯ d)"
      by (rule aval_abs_sound)+
    ultimately show ?thesis using Max 𝒢 by auto
  qed
qed

end

context mono_minmax_ops
begin

lemma special_transfer_mono:
  assumes le: "sigma1 ≤ sigma2"
  shows "special_transfer sc x sigma1 ≤ special_transfer sc x sigma2"
proof (cases sc)
  case Nondet_Int
  with le show ?thesis by (simp add: le_funD le_funI)
next
  case (Min a b)
  have "special_min ops (⟦a⟧♯ sigma1) (⟦b⟧♯ sigma1)
        ≤ special_min ops (⟦a⟧♯ sigma2) (⟦b⟧♯ sigma2)"
    using le by (intro special_min_mono aval_abs_mono)
  with le Min show ?thesis unfolding le_fun_def by auto
next
  case (Max a b)
  have "special_max ops (⟦a⟧♯ sigma1) (⟦b⟧♯ sigma1)
        ≤ special_max ops (⟦a⟧♯ sigma2) (⟦b⟧♯ sigma2)"
    using le by (intro special_max_mono aval_abs_mono)
  with le Max show ?thesis unfolding le_fun_def by auto
qed

end

end
```

10.3 Nonrelational_Transfer

```
theory Nonrelational_Transfer
  imports
    Nonrelational_Ops
    "Voblint_Framework.Check_Answer"
    "Voblint_Result.DG_Analysis"
begin
```

What each kind of CFG edge does to one abstract value per variable

A non-relational domain gives every variable one abstract value. Say what an expression evaluates to, what the whole-value element is, what `Min`/`Max` do, and how a guard refines its operands, and every edge operation is already fixed: an assignment overwrites its target with the evaluated right-hand side, a return writes the same into the return variable, a guard runs the backward branch the refinement operations determine, procedure entry resets the callee frame to the whole-value element and binds the formals, and skip, procedure-body entry and check observation leave the store alone.

`sound_nonrelational_ops` is the certificate for one `nonrelational_ops` bundle: the special operations are sound, the refinement operations form a reductive backward domain over the bundle's evaluator, and the queries are sound checks over the same evaluator. From it this theory states every edge operation once and proves each sound — a concrete store described by the input is still described by the output — ending in the transfer contract the analysis framework asks a domain for. `mono_nonrelational_ops` adds monotonicity for a bundle whose evaluator, special operations and refinement are monotone. A domain proves its capability certificates, interprets one of the two locales once at its bundle, and names the results; it proves nothing here again.

`tf_abs` is the per-edge dispatcher these operations add up to, and `tf_abs_eq_generic` identifies it with `generic_tf_abs` at the derived branch, which is what lets `tf_st_for_commute` inherit the executable mirror's commutation instead of restating it.

```
locale sound_nonrelational_ops =
  sound_minmax_ops "n_special ops" "n_aval ops"
  + backward: sound_refinement "r_intersect (n_refine ops)" "n_aval ops"
      "r_tobool (n_refine ops)" "r_inv_less (n_refine ops)" "r_inv_eq (n_refine ops)"
      "r_inv_plus (n_refine ops)" "r_inv_minus (n_refine ops)" "r_inv_times (n_refine ops)"
  + check: sound_check_query "q_less (n_query ops)" "q_eq (n_query ops)" gamma_state
      "n_aval ops"
  for ops :: "'a::numeric_domain nonrelational_ops"
begin
```

Assignment, return, and the operations that do nothing

`assign` overwrites the target with the evaluated expression and `ret` does the same for `ret_var`. `skip`, `body` and `event` are the identity. Each operation gets its soundness lemma here.

```
definition assign :: "vname ⇒ exp ⇒ 'a abs_state ⇒ 'a abs_state" where
  "assign x a d = d(x := n_aval ops a d)"

definition skip :: "'a abs_state ⇒ 'a abs_state" where
  "skip d = d"

definition body :: "pname ⇒ 'a abs_state ⇒ 'a abs_state" where
  "body p d = d"
```

A check observes its condition but never refines the store — narrowing a state against a checked condition is `sound_check_query`'s job — so `event` is the identity like `skip` and `body`.

```
definition event :: "analysis_event ⇒ 'a abs_state ⇒ 'a abs_state" where
  "event evt d = d"

definition ret :: "exp option ⇒ pname ⇒ 'a abs_state ⇒ 'a abs_state" where
  "ret e p d = (case e of None ⇒ d | Some a ⇒ assign ret_var a d)"

lemma assign_sound:
  assumes 𝒢: "s ∈ γ d"
  shows "s(x := ⟦a⟧ₑ s) ∈ γ (assign x a d)"
  unfolding assign_def
  using gamma_stateD[OF 𝒢] 𝒢 by blast

lemma skip_sound: "s ∈ γ d ⟹ s ∈ γ (skip d)"
  by (simp add: skip_def)

lemma body_sound: "s ∈ γ d ⟹ s ∈ γ (body p d)"
  by (simp add: body_def)
```

```
lemma event_sound: "s ∈ γ d ⟹ s ∈ γ (event evt d)"
  by (simp add: event_def)

lemma ret_sound:
  assumes 𝒢: "s ∈ γ d"
  shows "s(ret_var := (case e of None ⇒ s ret_var | Some a ⇒ ⟦a⟧e s)) ∈ γ (ret e p d)"
  using assign_sound[OF 𝒢] 𝒢 by (cases e) (simp_all add: ret_def)
```

Classifier-parametric procedure entry

Entry is the only operation that consults a classifier, inside `enter_frame`: assignment, guard and return never do, so everything above is classifier-free and only these three constants take one.

```
definition enter_frame_for :: "(vname ⇒ bool) ⇒ 'a abs_state ⇒ 'a abs_state" where
  "enter_frame_for 𝒢 = enter_frame 𝒢 top"

definition enter_for ::
    "(vname ⇒ bool) ⇒ vname list ⇒ exp list ⇒ 'a abs_state ⇒ 'a abs_state" where
  "enter_for 𝒢 = enter_binding 𝒢 top (n_aval ops)"

definition enter_ci_for ::
    "(vname ⇒ bool) ⇒ call_info ⇒ 'a abs_state ⇒ 'a abs_state" where
  "enter_ci_for 𝒢 ci = enter_for 𝒢 (ci_formals ci) (ci_args ci)"

lemma enter_for_sound:
  assumes 𝒢: "s ∈ γ d"
  shows "bind_formals xs (map (λe. ⟦e⟧e s) es) (enter_state cls s)
         ∈ γ (enter_for cls xs es d)"
  unfolding enter_for_def enter_binding_concrete[symmetric]
proof (rule enter_binding_sound[OF 𝒢 gamma_top])
  fix e
  show "⟦e⟧e s ∈ γ (n_aval ops e d)"
    using gamma_stateD[OF 𝒢] by blast
qed

lemma enter_ci_for_sound:
  assumes 𝒢: "s ∈ γ d"
  shows "bind_formals (ci_formals ci) (map (λe. ⟦e⟧e s) (ci_args ci)) (enter_state cls s)
         ∈ γ (enter_ci_for cls ci d)"
  using enter_for_sound[OF 𝒢, of ‹ci_formals ci› ‹ci_args ci›]
  by (simp add: enter_ci_for_def)
```

The transfer contract, and the per-edge dispatcher

The eight operations discharge the framework's contract in one `unfold_locales`: the call boundary the contract fixes is the structural one, so only the per-edge operations and the callee entry are the domain's own.

```
lemma is_sound_nonrelational_transfer:
  "sound_nonrelational_transfer 𝒢 skip assign special_transfer backward.branch body ret (enter_ci_for 𝒢) event"
  by unfold_locales
     (simp_all add: assign_sound special_transfer_sound backward.branch_sound skip_sound body_sound
        ret_sound enter_ci_for_sound event_sound)
```

The per-edge dispatcher. The executable mirror a domain runs dispatches on the action, so the equations relating the two through the represented function are stated at this shape rather than one lemma per operation.

```
definition tf_abs :: "edge_action ⇒ 'a abs_state ⇒ 'a abs_state" where
  "tf_abs = local_spec_step skip assign special_transfer backward.branch body ret event"

lemma tf_abs_simps [simp]:
  "tf_abs EA_Nop = skip"
  "tf_abs (EA_Assign x e) = assign x e"
  "tf_abs (EA_Special sc y) = special_transfer sc y"
  "tf_abs (EA_Assume b) = backward.branch b True"
  "tf_abs (EA_AssumeNot b) = backward.branch b False"
  "tf_abs (EA_Body p) = body p"
  "tf_abs (EA_Ret eo p) = ret eo p"
  "tf_abs (EA_Check l c) = event (Check_Event l c)"
  by (simp_all add: tf_abs_def)
```

The bundle an executable mirror unfolds against. `tf_abs` itself is not in it: `tf_abs_simps` is already `[simp]`, so a proof reaches the operation the action selects without naming the dispatcher.

```
lemmas op_defs =
  assign_def skip_def body_def event_def ret_def
  enter_frame_for_def enter_for_def enter_ci_for_def
```

Agreement with the executable mirror

Each operation above is the one the bundle determines — `Nondet_Int`'s $\top$, and every other case is definitional — so this domain's dispatcher is `generic_tf_abs` at its own bundle and branch. That identification is the whole content: the executable mirror's commutation is already proved once against `generic_tf_abs`.

```
lemma tf_abs_eq_generic: "tf_abs = generic_tf_abs ops backward.branch"
proof (rule ext, rule ext)
  fix a :: edge_action and d :: "'a abs_state"
  show "tf_abs a d = generic_tf_abs ops backward.branch a d"
    by (cases a)
       (simp_all add: op_defs split: special_call.splits option.splits)
qed
```

The derived guard filter is the backward domain's executable branch, so it commutes with the abstract guard on every live store, and the whole executable step with it.

```
theorem tf_st_for_commute:
  assumes "live_default_st 𝒢 s"
  shows
    "ϱ𝒢 (generic_tf_st_for ops 𝒢 a s) =
     tf_abs a (ϱ𝒢 s)"
  unfolding tf_abs_eq_generic
  by (rule generic_tf_st_for_commute)
     (simp add: backward.branch_st_with_ops [simplified] backward.branch_st_commute[OF assms])

lemma enter_st_for_commute:
  "ϱ𝒢 (generic_enter_st_for ops 𝒢 ci s) =
   enter_ci_for 𝒢 ci (ϱ𝒢 s)"
  by (simp add: generic_enter_st_for_def op_defs enter_binding_def enter_frame_def)
```

Registering the bundle with the pipeline

The bundle already settles everything `dg_analysis_exec` asks of the domain: the derived transfer is sound, the executable step and entry represent it, and the derived classifier is sound in both directions. What remains belongs to the context policy (the routing agreement and the seed key), the solver (a `certified_solver`) and the initial state, and stays a premise.

```
theorem dg_analysis_execI:
  assumes route_agree:
      "⋀𝒢 u ctx d ca. route 𝒢 u ctx d ca
         = route_abs 𝒢 u ctx (ϱ𝒢 d) ca"
    and seed_ne_buffer_key: "⋀v ctx. seed v ctx ≠ buffer_key"
    and solver: "certified_solver solve solve_dom solve_c"
    and init_sound: "⋀𝒢. cinit_stores 𝒢 ⊆ default_st_gamma 𝒢 init_st"
    and own_key: "⋀n. global_of n = buffer_key"
  shows "dg_analysis_exec (generic_tf_st_for ops) (generic_enter_st_for ops) init_st buffer_key
           global_of seed
           route solve solve_dom bot check.classify_check
           skip assign special_transfer backward.branch body ret enter_ci_for event route_abs solve_c"
proof (rule dg_analysis_exec.intro[OF solver dg_analysis_exec_axioms.intro], goal_cases)
  case (1 𝒢) show ?case by (rule is_sound_nonrelational_transfer)
next
  case (2 𝒢 a s) then show ?case
    unfolding tf_abs_def[symmetric] by (rule tf_st_for_commute)
next
  case (3 𝒢 ci s) show ?case
    by (rule enter_st_for_commute)
qed (fact route_agree seed_ne_buffer_key init_sound own_key
      check.classify_check_proved check.classify_check_refuted refl)+
```

The specification and its soundness, before any context

What the bundle supplies to the framework on its own: a whole-state D/G specification over the executable step and entry, and its soundness against the concretization of the read-back local value, λ`d g.` γ `(`$\varrho_{\mathcal{G}}$ `d)`. Neither mentions a context, a routing function, a seed key or a solver, so every domain reads them off its bundle instead of restating them.

```
definition spec_exec ::
  "(vname ⇒ bool) ⇒ ('a default_st ⇒ bool)
   ⇒ ('x, 'k, unit, 'a default_st lifted, 'a default_st lifted) dg_spec" where
  "spec_exec 𝒢 empty_pred =
     exec_dg_spec 𝒢 empty_pred (generic_tf_st_for ops 𝒢) (generic_enter_st_for ops 𝒢)"

lemma dg_spec_wf_spec_exec [intro, simp]: "dg_spec_wf (spec_exec 𝒢 empty_pred)"
```

```
  by (simp add: spec_exec_def)

context
  includes default_st_syntax
  fixes 𝒢 :: "vname ⇒ bool" and empty_pred :: "'a default_st ⇒ bool"
  assumes exact: "⋀s. empty_pred s = is_empty_state (ϱ𝒢 s)"
begin

interpretation dom: dg_domain_exec
  𝒢 empty_pred "generic_tf_st_for ops 𝒢" "generic_enter_st_for ops 𝒢"
  skip assign special_transfer backward.branch body ret "enter_ci_for 𝒢" event
  by unfold_locales
     (rule tf_st_for_commute[unfolded tf_abs_def], assumption,
      rule enter_st_for_commute, rule exact)

lemma gamma_exec_readback: "dom.gamma_exec = (λd g. γ (ϱ𝒢 d))"
  by (intro ext) (simp add: dom.gamma_exec_def gamma_lift_default_st_gamma_to_fun)

theorem sound_exec:
  "analysis_contract (spec_exec 𝒢 empty_pred) (λd g. γ (ϱ𝒢 d)) 𝒢"
proof -
  have "analysis_contract dom.spec_st (λd e. dom.gamma_exec d (e ())) 𝒢"
    by (rule dom.analysis_contract_st[OF is_sound_nonrelational_transfer])
  then show ?thesis by (simp add: gamma_exec_readback spec_exec_def)
qed
```

Entry is stated apart from `analysis_contract`, so a routed instance cites it separately; the alternative list is the singleton this entry answers.

```
theorem entry_cover_exec:
  assumes "s ∈ γ (ϱ𝒢 d)"
  shows "entry_pairs_cover (λd'. γ (ϱ𝒢 d')) s
           (call_enter 𝒢 (CallEdge (ci_dst ci) (ci_formals ci) (ci_args ci)) s)
           [(d, transfer_lift empty_pred (generic_enter_st_for ops 𝒢 ci) d)]"
proof -
  have "s ∈ gamma_lift (default_st_gamma 𝒢) d"
    using assms by (simp add: gamma_lift_default_st_gamma_to_fun)
  from dom.entry_pairs_cover_st [OF is_sound_nonrelational_transfer this]
  show ?thesis by (simp add: gamma_lift_default_st_gamma_to_fun)
qed

end

end
```

Monotone bundles

Monotonicity is a separate certificate: the transfer contract asks only for soundness, and a bundle whose evaluator or refinement has no monotonicity proof (Int's fixpoint refinement) is still a sound one. A monotone bundle additionally has monotone special operations and a monotone backward domain, and then every derived operation is monotone.

```
locale mono_nonrelational_ops = sound_nonrelational_ops ops
  + mono_minmax_ops "n_special ops" "n_aval ops"
  + backward: mono_refinement "r_intersect (n_refine ops)" "n_aval ops"
      "r_tobool (n_refine ops)" "r_inv_less (n_refine ops)" "r_inv_eq (n_refine ops)"
      "r_inv_plus (n_refine ops)" "r_inv_minus (n_refine ops)" "r_inv_times (n_refine ops)"
  for ops :: "'a::numeric_domain nonrelational_ops"
begin

lemma assign_mono: "d1 ≤ d2 ⟹ assign x a d1 ≤ assign x a d2"
  unfolding assign_def by (simp add: aval_abs_mono le_funD le_funI)

lemma ret_mono: "d1 ≤ d2 ⟹ ret e p d1 ≤ ret e p d2"
  by (cases e) (simp_all add: ret_def assign_mono)

lemma enter_for_mono:
  assumes "d1 ≤ d2"
  shows "enter_for 𝒢 xs es d1 ≤ enter_for 𝒢 xs es d2"
  unfolding enter_for_def
  by (rule enter_binding_mono[OF assms]) (rule aval_abs_mono[OF assms])

end
```

Certifying a bundle from its capability certificates

`sound_nonrelational_opsI` and `mono_nonrelational_opsI` assemble the bundle certificates from the special-call, refinement and check-query certificates, so a domain never unfolds the combined locales itself.

```
lemma sound_nonrelational_opsI:
  assumes "sound_minmax_ops (n_special ops) (n_aval ops)"
    and "sound_refinement (r_intersect (n_refine ops)) (n_aval ops)
           (r_tobool (n_refine ops)) (r_inv_less (n_refine ops)) (r_inv_eq (n_refine ops))
           (r_inv_plus (n_refine ops)) (r_inv_minus (n_refine ops)) (r_inv_times (n_refine ops))"
    and "sound_check_query (q_less (n_query ops)) (q_eq (n_query ops)) gamma_state (n_aval ops)"
  shows "sound_nonrelational_ops ops"
  by (intro sound_nonrelational_ops.intro assms)

lemma mono_nonrelational_opsI:
  assumes special: "mono_minmax_ops (n_special ops) (n_aval ops)"
    and backward: "mono_refinement (r_intersect (n_refine ops)) (n_aval ops)
           (r_tobool (n_refine ops)) (r_inv_less (n_refine ops)) (r_inv_eq (n_refine ops))
           (r_inv_plus (n_refine ops)) (r_inv_minus (n_refine ops)) (r_inv_times (n_refine ops))"
    and "sound_check_query (q_less (n_query ops)) (q_eq (n_query ops)) gamma_state (n_aval ops)"
  shows "mono_nonrelational_ops ops"
proof -
  interpret mono_minmax_ops "n_special ops" "n_aval ops" by (rule special)
  interpret mono_refinement "r_intersect (n_refine ops)" "n_aval ops"
      "r_tobool (n_refine ops)" "r_inv_less (n_refine ops)" "r_inv_eq (n_refine ops)"
      "r_inv_plus (n_refine ops)" "r_inv_minus (n_refine ops)" "r_inv_times (n_refine ops)"
    by (rule backward)
  show ?thesis
    by (intro mono_nonrelational_ops.intro sound_nonrelational_opsI assms)
       unfold_locales
qed

end
```

10.4 Nonrelational_Ops

```
theory Nonrelational_Ops
  imports
    Special_Ops
    Exec_Backward
    "Voblint_Framework.DG_Local_State_Spec"
    "Voblint_Exec.Default_St_Restriction_Refinement"
begin

unbundle default_st_syntax
```

The primitives one abstract value per variable is built from

Say what an expression evaluates to, which comparisons the domain decides, how a guard refines its operands, how the two special calls combine two values, and every edge of a compiled graph is determined; the whole-value element is the class $\top$. `nonrelational_ops` is that bundle of primitive choices, written once per domain. Everything else is derived from it: the guard filter on the executable store here, the abstract branch and the transfer in `Nonrelational_Transfer`, which certifies one such bundle.

Two constructions follow from it here. `generic_enter_st_for` is procedure entry on the executable store: evaluate the actuals in the caller's state, reset the callee frame to the whole-value element, bind the formals. `generic_tf_st_for` is the per-edge executable step, and `generic_tf_st_for_commute` says it agrees with the abstract dispatcher `generic_tf_abs` on the function the executable store represents, provided the guard filter commutes with the abstract branch on the state at hand.

`generic_tf_abs` takes the abstract branch as a separate argument so this commutation lemma needs no soundness locale. For a certified bundle the branch is not an independent choice: `sound_nonrelational_ops` instantiates it with the backward branch of `n_aval` and `n_refine` and discharges the guard obligation once, generically.

The bundle is constrained to `'a::{bot, top}` and the executable step to `'a::executable_domain`, never `'a::numeric_domain`. `numeric_domain` fixes `gamma` as a class operation, and code generation for a definition at that sort must resolve `gamma`'s code equation for the concrete type even though nothing here calls it. `ivl`'s `gamma_ivl` has no well-sorted code equation (`int` is not of sort `enum`), so that unused obligation breaks unrelated `by eval` proofs downstream.

The comparisons a domain decides, each answering `None` when it cannot tell.

```
record 'a query_ops =
  q_less :: "'a ⇒ 'a ⇒ bool option"
  q_eq   :: "'a ⇒ 'a ⇒ bool option"

record ('a::"{bot, top}") nonrelational_ops =
  n_aval    :: "exp => (vname => 'a) => 'a"
  n_query   :: "'a query_ops"
  n_refine  :: "'a refine_ops"
  n_special :: "'a special_ops"
```

The guard filter on the executable store is derived from the evaluator and the refinement operations, never supplied: the bundle determines it.

```
abbreviation n_bfilter ::
  "'a::executable_domain nonrelational_ops
   ⇒ (vname ⇒ bool) ⇒ exp ⇒ bool ⇒ 'a default_st ⇒ 'a default_st" where
  "n_bfilter ops ≡ branch_st_with (n_aval ops) (n_refine ops)"
```

Procedure entry

`generic_enter_st_for` opens a callee frame with every local at `top`, then binds the formals to the arguments evaluated in the caller's store.

```
definition generic_enter_st_for ::
    "'a::{bot, top} nonrelational_ops => (vname => bool) => call_info =>
       'a default_st => 'a default_st" where
  "generic_enter_st_for ops 𝒢 ci s =
     bind_formals_default_st 𝒢 (ci_formals ci)
       (map (λe. n_aval ops e (ϱ𝒢 s)) (ci_args ci))
       (enter_frame_D_default_st top s)"
```

The per-edge step, on both stores

`generic_tf_st_for` is the executable step on the split local/global store, routing each write through `location_of`. `generic_tf_abs` states the same edge semantics on a plain abstract state via `local_spec_step`.

```
fun generic_tf_st_for ::
    "'a::executable_domain nonrelational_ops => (vname => bool) => edge_action =>
       'a default_st => 'a default_st" where
    "generic_tf_st_for ops 𝒢 EA_Nop s = s"
  | "generic_tf_st_for ops 𝒢 (EA_Assign x a) s =
       s⟨location_of 𝒢 x := n_aval ops a (ϱ𝒢 s)⟩"
  | "generic_tf_st_for ops 𝒢 (EA_Special sc x) s =
       s⟨location_of 𝒢 x :=
         (case sc of
            Nondet_Int => top
          | Min a b => special_min (n_special ops)
                         (n_aval ops a (ϱ𝒢 s))
                         (n_aval ops b (ϱ𝒢 s))
          | Max a b => special_max (n_special ops)
                         (n_aval ops a (ϱ𝒢 s))
                         (n_aval ops b (ϱ𝒢 s)))⟩"
  | "generic_tf_st_for ops 𝒢 (EA_Assume b) s = n_bfilter ops 𝒢 b True s"
  | "generic_tf_st_for ops 𝒢 (EA_AssumeNot b) s = n_bfilter ops 𝒢 b False s"
  | "generic_tf_st_for ops 𝒢 (EA_Body p) s = s"
  | "generic_tf_st_for ops 𝒢 (EA_Ret None p) s = s"
  | "generic_tf_st_for ops 𝒢 (EA_Ret (Some a) p) s =
       s⟨location_of 𝒢 ret_var := n_aval ops a (ϱ𝒢 s)⟩"
  | "generic_tf_st_for ops 𝒢 (EA_Check l cnd) s = s"

definition generic_tf_abs ::
    "'a::{bot, top} nonrelational_ops => (exp => bool => 'a abs_state => 'a abs_state) =>
       edge_action => 'a abs_state => 'a abs_state" where
  "generic_tf_abs ops br =
     local_spec_step
       (λsigma. sigma)
       (λx a sigma. sigma(x := n_aval ops a sigma))
       (λsc x sigma. sigma(x := (case sc of
            Nondet_Int => top
          | Min a b => special_min (n_special ops) (n_aval ops a sigma) (n_aval ops b sigma)
          | Max a b => special_max (n_special ops) (n_aval ops a sigma) (n_aval ops b sigma))))
       br
       (λp sigma. sigma)
       (λeo p sigma. case eo of None => sigma | Some a => sigma(ret_var := n_aval ops a sigma))
       (λevt sigma. sigma)"

lemma generic_tf_abs_simps [simp]:
  "generic_tf_abs ops br EA_Nop sigma = sigma"
  "generic_tf_abs ops br (EA_Assign x a) sigma = sigma(x := n_aval ops a sigma)"
  "generic_tf_abs ops br (EA_Special sc x) sigma =
     sigma(x := (case sc of
        Nondet_Int => top
      | Min a b => special_min (n_special ops) (n_aval ops a sigma) (n_aval ops b sigma)
      | Max a b => special_max (n_special ops) (n_aval ops a sigma) (n_aval ops b sigma)))"
  "generic_tf_abs ops br (EA_Assume b) = br b True"
  "generic_tf_abs ops br (EA_AssumeNot b) = br b False"
  "generic_tf_abs ops br (EA_Body p) sigma = sigma"
  "generic_tf_abs ops br (EA_Ret None p) sigma = sigma"
  "generic_tf_abs ops br (EA_Ret (Some a) p) sigma = sigma(ret_var := n_aval ops a sigma)"
  "generic_tf_abs ops br (EA_Check l cnd) sigma = sigma"
  by (simp_all add: generic_tf_abs_def)
```

Every case but the guard rewrites by `readback`'s own update equation, which is what leaves the guard as the only obligation an instance still has to discharge.

```
theorem generic_tf_st_for_commute:
  fixes ops :: "'a::executable_domain nonrelational_ops"
  assumes branch:
    "⋀b pol. ϱ𝒢 (n_bfilter ops 𝒢 b pol s) =
              br b pol (ϱ𝒢 s)"
  shows
    "ϱ𝒢 (generic_tf_st_for ops 𝒢 a s) =
     generic_tf_abs ops br a (ϱ𝒢 s)"
proof (cases a)
  case EA_Nop
  then show ?thesis by simp
next
  case (EA_Assign x e)
  then show ?thesis by simp
next
  case (EA_Special sc x)
  then show ?thesis by (cases sc) simp_all
next
```

```
  case (EA_Assume b)
  then show ?thesis by (simp add: branch)
next
  case (EA_AssumeNot b)
  then show ?thesis by (simp add: branch)
next
  case (EA_Body p)
  then show ?thesis by simp
next
  case (EA_Ret eo p)
  then show ?thesis by (cases eo) simp_all
next
  case (EA_Check l cnd)
  then show ?thesis by simp
qed

unbundle no default_st_syntax

end
```

10.5 Exec_Backward

```
theory Exec_Backward
    imports "Voblint_Exec.Default_St_Reachability" "Voblint_Domain.Backward_Domain_Mono"
begin

unbundle default_st_syntax
```

Generic executable mirror of backward filtering

Every interpretation of `sound_refinement` gets an executable `'a default_st` mirror of its `afilter` / `bfilter` for free, parameterized by an explicit location classifier $\mathcal{G}$: `afilter_st` / `bfilter_st` and their commutation with the abstract filters through `readback`, proved once here so no domain needs to repeat the induction by hand. Each concrete domain names its specialization through its existing `sound_refinement` interpretation; Sign, Interval, Congruence, and Int all reuse the same correspondence.

Standalone executable recursion, outside the semantic locale

`afilter_st_lift`/`bfilter_st_lift` below are proved *inside* `sound_refinement`, so once exported their `.simps` each carry that locale's own soundness assumptions as a hypothesis – a fact code generation cannot discharge, so a recursive `fun` defined that way has no usable code equation, no matter how it is later aliased. `refine_ops` packages the same raw operations directly, and `afilter_st_lift_with`/ `bfilter_st_lift_with` recurse over an explicit `'a::executable_domain` value instead of interpreting the soundness locale, so their equations carry no such premise. The `sound_refinement` context below builds an `ops` value from its own fixed operations and proves the two recursions agree; `branch_st` calls the standalone form directly.

```
record 'a refine_ops =
  r_tobool    :: "'a ⇒ bool option"
  r_inv_less  :: "bool ⇒ 'a ⇒ 'a ⇒ 'a × 'a"
  r_inv_eq    :: "bool ⇒ 'a ⇒ 'a ⇒ 'a × 'a"
  r_inv_plus  :: "'a ⇒ 'a ⇒ 'a ⇒ 'a × 'a"
  r_inv_minus :: "'a ⇒ 'a ⇒ 'a ⇒ 'a × 'a"
  r_inv_times :: "'a ⇒ 'a ⇒ 'a ⇒ 'a × 'a"
  r_intersect :: "'a ⇒ 'a ⇒ 'a"

lemma refine_ops_eta [simp]:
  "(|r_tobool = r_tobool R, r_inv_less = r_inv_less R, r_inv_eq = r_inv_eq R,
    r_inv_plus = r_inv_plus R, r_inv_minus = r_inv_minus R,
    r_inv_times = r_inv_times R, r_intersect = r_intersect R|) = R"
  by (cases R) simp

definition feasible_with ::
  "(exp ⇒ 'a abs_state ⇒ 'a) ⇒ 'a::executable_domain refine_ops ⇒ exp ⇒ bool ⇒ 'a abs_state ⇒ bool"
where
  "feasible_with ev ops e pol sigma =
     (¬ is_empty (ev e sigma) ∧ r_tobool ops (ev e sigma) ≠ Some (¬ pol))"

fun afilter_st_lift_with ::
  "(exp ⇒ 'a abs_state ⇒ 'a) ⇒ 'a::executable_domain refine_ops
   ⇒ (vname ⇒ bool) ⇒ exp ⇒ 'a ⇒ 'a default_st lifted ⇒ 'a default_st lifted"
where
    "afilter_st_lift_with ev ops 𝒢 (V x) a x_lift = do {
       s <- x_lift;
       default_st_set_lift (Lifted s) (location_of 𝒢 x)
         (r_intersect ops a (ϱ𝒢 s x))
     }"
  | "afilter_st_lift_with ev ops 𝒢 (Plus e1 e2) a x_lift = do {
       s <- x_lift;
       let (a1, a2) = r_inv_plus ops a
             (ev e1 (ϱ𝒢 s))
             (ev e2 (ϱ𝒢 s));
       afilter_st_lift_with ev ops 𝒢 e1 a1 (afilter_st_lift_with ev ops 𝒢 e2 a2 (Lifted s))
     }"
  | "afilter_st_lift_with ev ops 𝒢 (Minus e1 e2) a x_lift = do {
       s <- x_lift;
       let (a1, a2) = r_inv_minus ops a
             (ev e1 (ϱ𝒢 s))
             (ev e2 (ϱ𝒢 s));
       afilter_st_lift_with ev ops 𝒢 e1 a1 (afilter_st_lift_with ev ops 𝒢 e2 a2 (Lifted s))
     }"
  | "afilter_st_lift_with ev ops 𝒢 (Times e1 e2) a x_lift = do {
       s <- x_lift;
       let (a1, a2) = r_inv_times ops a
```

```
            (ev e1 (ϱ𝒢 s))
            (ev e2 (ϱ𝒢 s));
      afilter_st_lift_with ev ops 𝒢 e1 a1 (afilter_st_lift_with ev ops 𝒢 e2 a2 (Lifted s))
    }"
  | "afilter_st_lift_with ev ops 𝒢 _ a x_lift = x_lift"

lemma afilter_st_lift_with_Bot [simp]: "afilter_st_lift_with ev ops 𝒢 e a Bot = Bot"
  by (induction e) simp_all

fun bfilter_st_lift_with ::
  "(exp ⇒ 'a abs_state ⇒ 'a) ⇒ 'a::executable_domain refine_ops
   ⇒ (vname ⇒ bool) ⇒ exp ⇒ bool ⇒ 'a default_st lifted ⇒ 'a default_st lifted"
where
    "bfilter_st_lift_with ev ops 𝒢 (Less e1 e2) res x_lift = do {
      s <- x_lift;
      let (a1, a2) = r_inv_less ops res
            (ev e1 (ϱ𝒢 s))
            (ev e2 (ϱ𝒢 s));
      afilter_st_lift_with ev ops 𝒢 e1 a1 (afilter_st_lift_with ev ops 𝒢 e2 a2 (Lifted s))
    }"
  | "bfilter_st_lift_with ev ops 𝒢 (GreaterEq e1 e2) res x_lift = do {
      s <- x_lift;
      let (a1, a2) = r_inv_less ops (¬ res)
            (ev e1 (ϱ𝒢 s))
            (ev e2 (ϱ𝒢 s));
      afilter_st_lift_with ev ops 𝒢 e1 a1 (afilter_st_lift_with ev ops 𝒢 e2 a2 (Lifted s))
    }"
  | "bfilter_st_lift_with ev ops 𝒢 (Greater e1 e2) res x_lift = do {
      s <- x_lift;
      let (a1, a2) = r_inv_less ops res
            (ev e2 (ϱ𝒢 s))
            (ev e1 (ϱ𝒢 s));
      afilter_st_lift_with ev ops 𝒢 e2 a1 (afilter_st_lift_with ev ops 𝒢 e1 a2 (Lifted s))
    }"
  | "bfilter_st_lift_with ev ops 𝒢 (LessEq e1 e2) res x_lift = do {
      s <- x_lift;
      let (a1, a2) = r_inv_less ops (¬ res)
            (ev e2 (ϱ𝒢 s))
            (ev e1 (ϱ𝒢 s));
      afilter_st_lift_with ev ops 𝒢 e2 a1 (afilter_st_lift_with ev ops 𝒢 e1 a2 (Lifted s))
    }"
  | "bfilter_st_lift_with ev ops 𝒢 (Not b) res x_lift =
      bfilter_st_lift_with ev ops 𝒢 b (¬ res) x_lift"
  | "bfilter_st_lift_with ev ops 𝒢 (And b1 b2) True x_lift =
      bfilter_st_lift_with ev ops 𝒢 b1 True (bfilter_st_lift_with ev ops 𝒢 b2 True x_lift)"
  | "bfilter_st_lift_with ev ops 𝒢 (And b1 b2) False x_lift = do {
      s <- x_lift;
      (if feasible_with ev ops b1 False (ϱ𝒢 s)
       then bfilter_st_lift_with ev ops 𝒢 b1 False (Lifted s) else Bot)
      ⊔ (if feasible_with ev ops b2 False (ϱ𝒢 s)
         then bfilter_st_lift_with ev ops 𝒢 b2 False (Lifted s) else Bot)
    }"
  | "bfilter_st_lift_with ev ops 𝒢 (Or b1 b2) True x_lift = do {
      s <- x_lift;
      (if feasible_with ev ops b1 True (ϱ𝒢 s)
       then bfilter_st_lift_with ev ops 𝒢 b1 True (Lifted s) else Bot)
      ⊔ (if feasible_with ev ops b2 True (ϱ𝒢 s)
         then bfilter_st_lift_with ev ops 𝒢 b2 True (Lifted s) else Bot)
    }"
  | "bfilter_st_lift_with ev ops 𝒢 (Or b1 b2) False x_lift =
      bfilter_st_lift_with ev ops 𝒢 b1 False (bfilter_st_lift_with ev ops 𝒢 b2 False x_lift)"
  | "bfilter_st_lift_with ev ops 𝒢 (Eq e1 e2) res x_lift = do {
      s <- x_lift;
      let (a1, a2) = r_inv_eq ops res
            (ev e1 (ϱ𝒢 s))
            (ev e2 (ϱ𝒢 s));
      afilter_st_lift_with ev ops 𝒢 e1 a1 (afilter_st_lift_with ev ops 𝒢 e2 a2 (Lifted s))
    }"
  | "bfilter_st_lift_with ev ops 𝒢 (NotEq e1 e2) res x_lift = do {
      s <- x_lift;
      let (a1, a2) = r_inv_eq ops (¬ res)
            (ev e1 (ϱ𝒢 s))
            (ev e2 (ϱ𝒢 s));
      afilter_st_lift_with ev ops 𝒢 e1 a1 (afilter_st_lift_with ev ops 𝒢 e2 a2 (Lifted s))
    }"
  | "bfilter_st_lift_with ev ops 𝒢 e res x_lift = do {
```

```
      s <- x_lift;
      let (a1, a2) = r_inv_eq ops (¬ res)
            (ev e (ϱ_𝒢 s))
            (ev (N 0) (ϱ_𝒢 s));
      afilter_st_lift_with ev ops 𝒢 e a1 (Lifted s)
    }"

lemma bfilter_st_lift_with_Bot [simp]: "bfilter_st_lift_with ev ops 𝒢 b res Bot = Bot"
proof (induction b arbitrary: res)
  case (And b1 b2) then show ?case by (cases res) simp_all
next
  case (Or b1 b2) then show ?case by (cases res) simp_all
qed simp_all
```

`map_lift_sup`'s left-hand side is `map_lift ?f (?x ⊔ ?y)`, so using it with a schematic `?f` makes the simplifier search higher-order matches and, in the join cases below, exceed its unification bound. Specializing `f` to `default_st_to_fun 𝒢` once leaves a first-order rewrite that needs no search — and the specialization is proved by the case split directly rather than through `map_lift_sup`, since resolving against that rule is itself where the search happens.

```
lemma map_lift_default_st_to_fun_sup [simp]:
  "ϱ_𝒢 (x ⊔ y :: _ default_st lifted) =
     ϱ_𝒢 x ⊔ ϱ_𝒢 y"
  by (cases x; cases y) simp_all
```

The executable branch, determined by the evaluator and the refinement operations alone. A non-relational analysis derives its guard filter from its operations through this, so no domain supplies a filter of its own; `branch_st_with_ops` below identifies it with the locale's `branch_st`.

```
definition branch_st_with ::
  "(exp ⇒ 'a abs_state ⇒ 'a) ⇒ 'a::executable_domain refine_ops
   ⇒ (vname ⇒ bool) ⇒ exp ⇒ bool ⇒ 'a default_st ⇒ 'a default_st"
where
  "branch_st_with ev ops 𝒢 e pol s =
     (if feasible_with ev ops e pol (ϱ_𝒢 s)
      then collapse_lift (bfilter_st_lift_with ev ops 𝒢 e pol (Lifted s))
      else bot)"

context sound_refinement
begin

fun afilter_st ::
  "(vname => bool) => exp => 'a => 'a default_st => 'a default_st"
where
    "afilter_st 𝒢 (V x) a s =
       s⟨location_of 𝒢 x := intersect a (ϱ_𝒢 s x)⟩"
  | "afilter_st 𝒢 (Plus e1 e2) a s =
       (let (a1, a2) = inv_plus a
              (⟦e1⟧♯ (ϱ_𝒢 s))
              (⟦e2⟧♯ (ϱ_𝒢 s))
        in afilter_st 𝒢 e1 a1 (afilter_st 𝒢 e2 a2 s))"
  | "afilter_st 𝒢 (Minus e1 e2) a s =
       (let (a1, a2) = inv_minus a
              (⟦e1⟧♯ (ϱ_𝒢 s))
              (⟦e2⟧♯ (ϱ_𝒢 s))
        in afilter_st 𝒢 e1 a1 (afilter_st 𝒢 e2 a2 s))"
  | "afilter_st 𝒢 (Times e1 e2) a s =
       (let (a1, a2) = inv_times a
              (⟦e1⟧♯ (ϱ_𝒢 s))
              (⟦e2⟧♯ (ϱ_𝒢 s))
        in afilter_st 𝒢 e1 a1 (afilter_st 𝒢 e2 a2 s))"
  | "afilter_st 𝒢 _ a s = s"

fun bfilter_st ::
  "(vname => bool) => exp => bool => 'a default_st => 'a default_st"
where
    "bfilter_st 𝒢 (Less e1 e2) res s =
       (let (a1, a2) = inv_less res
              (⟦e1⟧♯ (ϱ_𝒢 s))
              (⟦e2⟧♯ (ϱ_𝒢 s))
        in afilter_st 𝒢 e1 a1 (afilter_st 𝒢 e2 a2 s))"
  | "bfilter_st 𝒢 (GreaterEq e1 e2) res s =
       (let (a1, a2) = inv_less (¬ res)
              (⟦e1⟧♯ (ϱ_𝒢 s))
              (⟦e2⟧♯ (ϱ_𝒢 s))
```

```
      in afilter_st 𝒢 e1 a1 (afilter_st 𝒢 e2 a2 s))"
  | "bfilter_st 𝒢 (Greater e1 e2) res s =
      (let (a1, a2) = inv_less res
             ([⟦e2⟧]♯ (ϱ𝒢 s))
             ([⟦e1⟧]♯ (ϱ𝒢 s))
       in afilter_st 𝒢 e2 a1 (afilter_st 𝒢 e1 a2 s))"
  | "bfilter_st 𝒢 (LessEq e1 e2) res s =
      (let (a1, a2) = inv_less (¬ res)
             ([⟦e2⟧]♯ (ϱ𝒢 s))
             ([⟦e1⟧]♯ (ϱ𝒢 s))
       in afilter_st 𝒢 e2 a1 (afilter_st 𝒢 e1 a2 s))"
  | "bfilter_st 𝒢 (Not b) res s = bfilter_st 𝒢 b (¬ res) s"
  | "bfilter_st 𝒢 (And b1 b2) True s =
      bfilter_st 𝒢 b1 True (bfilter_st 𝒢 b2 True s)"
  | "bfilter_st 𝒢 (And b1 b2) False s =
      (if feasible b1 False (ϱ𝒢 s)
       then bfilter_st 𝒢 b1 False s else bot)
      ⊔ (if feasible b2 False (ϱ𝒢 s)
           then bfilter_st 𝒢 b2 False s else bot)"
  | "bfilter_st 𝒢 (Or b1 b2) True s =
      (if feasible b1 True (ϱ𝒢 s)
       then bfilter_st 𝒢 b1 True s else bot)
      ⊔ (if feasible b2 True (ϱ𝒢 s)
           then bfilter_st 𝒢 b2 True s else bot)"
  | "bfilter_st 𝒢 (Or b1 b2) False s =
      bfilter_st 𝒢 b1 False (bfilter_st 𝒢 b2 False s)"
  | "bfilter_st 𝒢 (Eq e1 e2) res s =
      (let (a1, a2) = inv_eq res
             ([⟦e1⟧]♯ (ϱ𝒢 s))
             ([⟦e2⟧]♯ (ϱ𝒢 s))
       in afilter_st 𝒢 e1 a1 (afilter_st 𝒢 e2 a2 s))"
  | "bfilter_st 𝒢 (NotEq e1 e2) res s =
      (let (a1, a2) = inv_eq (¬ res)
             ([⟦e1⟧]♯ (ϱ𝒢 s))
             ([⟦e2⟧]♯ (ϱ𝒢 s))
       in afilter_st 𝒢 e1 a1 (afilter_st 𝒢 e2 a2 s))"
  | "bfilter_st 𝒢 e res s =
      (let (a1, a2) = inv_eq (¬ res)
             ([⟦e⟧]♯ (ϱ𝒢 s))
             ([⟦N 0⟧]♯ (ϱ𝒢 s))
       in afilter_st 𝒢 e a1 s)"
```

`afilter_st_lift`/`bfilter_st_lift` mirror `afilter_st`/`bfilter_st`'s own recursion exactly, but thread a `default_st lifted` value through it: a `Bot` input never reaches a further narrowing step, and a leaf whose freshly narrowed element is `is_empty` collapses to `Bot` via `default_st_set_lift`. The compound cases either sequence two single-variable narrows (never re-checking a location the other branch already settled) or join two gated branches via ⊔. Joining two live branches can never produce a witness-bottom result, since `is_empty` is downward closed (⟦`?a` ≤ `?b; is_empty ?b`⟧ ⟹ `is_empty ?a`) and each branch's own value is a lower bound of the join; a branch whose gate ruled its polarity out contributes `bot` instead, so the join is witness-bottom exactly when both arms are – either because a gate ruled the polarity out, or because the arm's own recursion narrowed to `Bot`. A permitting gate is therefore not on its own enough to keep the join live. No per-domain code is needed: this is generic in the `sound_refinement` operations, exactly like `afilter_st`/`bfilter_st` themselves.

`ops` packages this locale's own fixed operations into a `refine_ops` value, so the lifted filters below are the standalone recursion instantiated here rather than a second copy of it. It is an `abbreviation`, not a `definition`: it must inline to the record literal at every use site, including inside `branch_st`'s own executable definition below, rather than naming a separate locale-internal constant whose own code equation would carry `sound_refinement`'s assumptions as a premise – the same problem `afilter_st_lift_with`/`bfilter_st_lift_with` exist to avoid, one level up.

```
abbreviation ops :: "'a refine_ops" where
  "ops ≡ (|r_tobool = tobool, r_inv_less = inv_less,
           r_inv_eq = inv_eq, r_inv_plus = inv_plus, r_inv_minus = inv_minus,
           r_inv_times = inv_times, r_intersect = intersect|)"

lemma feasible_with_ops [simp]: "feasible_with aval_abs ops = feasible"
  by (intro ext) (simp add: feasible_with_def feasible_def)

definition afilter_st_lift ::
  "(vname => bool) => exp => 'a => 'a default_st lifted => 'a default_st lifted"
where
  "afilter_st_lift 𝒢 e a x_lift = afilter_st_lift_with aval_abs ops 𝒢 e a x_lift"
```

```
lemmas afilter_st_lift_simps [simp] =
  afilter_st_lift_with.simps [of aval_abs ops, folded afilter_st_lift_def]

lemma afilter_st_lift_Bot [simp]: "afilter_st_lift 𝒢 e a Bot = Bot"
  by (simp add: afilter_st_lift_def)

lemma afilter_st_commute:
  "ϱ𝒢 (afilter_st 𝒢 e a s) =
     afilter e a (ϱ𝒢 s)"
proof (induction e arbitrary: a s)
  case (N n)
  then show ?case by simp
next
  case (V x)
  then show ?case by simp
next
  case (Plus e1 e2)
  show ?case by (simp add: Plus.IH split: prod.splits)
next
  case (Minus e1 e2)
  show ?case by (simp add: Minus.IH split: prod.splits)
next
  case (Times e1 e2)
  show ?case by (simp add: Times.IH split: prod.splits)
next
  case (Less e1 e2) then show ?case by simp
next
  case (Div e1 e2) then show ?case by simp
next
  case (Mod e1 e2) then show ?case by simp
next
  case (LessEq e1 e2) then show ?case by simp
next
  case (Greater e1 e2) then show ?case by simp
next
  case (GreaterEq e1 e2) then show ?case by simp
next
  case (Eq e1 e2) then show ?case by simp
next
  case (NotEq e1 e2) then show ?case by simp
next
  case (Not e) then show ?case by simp
next
  case (And e1 e2) then show ?case by simp
next
  case (Or e1 e2) then show ?case by simp
qed
```

`bfilter_st_lift` mirrors `bfilter_lifted`'s recursion one constructor at a time, exactly as `bfilter_st` mirrors plain `bfilter`: the sequential cases (`Not`/`And True`/`Or False`/`Less`/`Eq`) chain two narrowing steps, and the compound-join cases (`And False`/`Or True`) gate each arm to structural `Bot` via `feasible` and join at the lifted level, ahead of the join rather than after it. This is what `bfilter`'s own join (kept plain and pointwise so `bfilter_sign`/`bfilter_ivl` stay code-generatable) cannot afford: a witness-bottom arm collapses to `Bot` here regardless of what the other arm's unrefined locations look like, so `Bot ⊔ x = x` discards it exactly, with no whole-expression probe needed afterward.

```
definition bfilter_st_lift ::
  "(vname => bool) => exp => bool => 'a default_st lifted => 'a default_st lifted"
where
  "bfilter_st_lift 𝒢 b res x_lift = bfilter_st_lift_with aval_abs ops 𝒢 b res x_lift"

lemmas bfilter_st_lift_simps [simp] =
  bfilter_st_lift_with.simps [of aval_abs ops, folded bfilter_st_lift_def afilter_st_lift_def]

lemma bfilter_st_lift_Bot [simp]: "bfilter_st_lift 𝒢 b res Bot = Bot"
  by (simp add: bfilter_st_lift_def)

end

context sound_refinement
begin

lemma afilter_lift_step:
  fixes s :: "'a default_st"
  assumes IH1: "!!s'. live_default_st 𝒢 s' ==>
```

```
              ϱ_G (afilter_st_lift G e1 a1 (Lifted s')) =
              normalize_lift is_empty_state (afilter e1 a1 (ϱ_G s'))"
    and IH2: "ϱ_G (afilter_st_lift G e2 a2 (Lifted s)) =
              normalize_lift is_empty_state (afilter e2 a2 (ϱ_G s))"
  shows "map_lift ϱ_G
          (afilter_st_lift G e1 a1 (afilter_st_lift G e2 a2 (Lifted s))) =
        normalize_lift is_empty_state
          (afilter e1 a1 (afilter e2 a2 (ϱ_G s)))"
proof (cases "is_empty_state (afilter e2 a2 (ϱ_G s))")
  case True
  then have bot2: "afilter_st_lift G e2 a2 (Lifted s) = Bot"
    using IH2 by (cases "afilter_st_lift G e2 a2 (Lifted s)") simp_all
  have "is_empty_state (afilter e1 a1 (afilter e2 a2 (ϱ_G s)))"
    using is_empty_state_antimono[OF afilter_reductive True] .
  with bot2 show ?thesis by simp
next
  case False
  then obtain t where t: "afilter_st_lift G e2 a2 (Lifted s) = Lifted t"
    using IH2 by (cases "afilter_st_lift G e2 a2 (Lifted s)") simp_all
  have ft: "ϱ_G t = afilter e2 a2 (ϱ_G s)"
    using IH2 t False by simp
  have live_t: "live_default_st G t"
    unfolding live_default_st_def ft using False by simp
  show ?thesis
    unfolding t using IH1[OF live_t] ft by simp
qed
```

`bfilter_lift_bind_step` composes two sequential lifted narrowing steps (`And True`, `Or False`): if the inner step already went `Bot`, the outer step is never reached and the composite is `Bot` too; otherwise it recurses on the inner step's live witness. This is (≫=)'s own associativity, specialized to the executable/lifted correspondence.

```
lemma bfilter_lift_bind_step:
  fixes s :: "'a default_st"
  assumes IH1: "!!s'. live_default_st G s' ==>
              ϱ_G (bfilter_st_lift G b1 res (Lifted s')) =
              bfilter_lifted b1 res (ϱ_G s')"
    and IH2: "ϱ_G (bfilter_st_lift G b2 res (Lifted s)) =
              bfilter_lifted b2 res (ϱ_G s)"
  shows "map_lift ϱ_G
          (bfilter_st_lift G b1 res (bfilter_st_lift G b2 res (Lifted s))) =
        bind (bfilter_lifted b2 res (ϱ_G s)) (bfilter_lifted b1 res)"
proof (cases "bfilter_st_lift G b2 res (Lifted s)")
  case Bot
  then show ?thesis using IH2 by simp
next
  case (Lifted t)
    have eq: "Lifted (ϱ_G t)
              = bfilter_lifted b2 res (ϱ_G s)"
      using IH2 Lifted by simp
  have live_t: "live_default_st G t"
  proof -
    have "normalized_lift is_empty_state (Lifted (ϱ_G t))"
      using bfilter_lifted_normalized[of b2 res "ϱ_G s"] eq by simp
    then show ?thesis by (simp add: live_default_st_def)
  qed
    show ?thesis
      unfolding Lifted using IH1[OF live_t] eq[symmetric] by simp
qed
```

`bfilter_lift_gate_step` is the executable/lifted correspondence for a single gated disjunct, feeding `feasible` the same represented state `bfilter_lifted`'s own gate reads: a feasible disjunct recurses, an infeasible one contributes `Bot` on both sides.

```
lemma bfilter_lift_gate_step:
  fixes s :: "'a default_st"
  assumes IH: "ϱ_G (bfilter_st_lift G b pol (Lifted s)) =
               bfilter_lifted b pol (ϱ_G s)"
  shows "map_lift ϱ_G
          (if feasible b pol (ϱ_G s)
           then bfilter_st_lift G b pol (Lifted s) else Bot)
        = (if feasible b pol (ϱ_G s)
           then bfilter_lifted b pol (ϱ_G s) else Bot)"
  by (cases "feasible b pol (ϱ_G s)") (simp_all add: IH)


lemma afilter_st_lift_correct:
```

```
  fixes s :: "'a default_st"
  assumes "live_default_st 𝒢 s"
  shows "ϱ𝒢 (afilter_st_lift 𝒢 e a (Lifted s)) =
         normalize_lift is_empty_state (afilter e a (ϱ𝒢 s))"
using assms proof (induction e arbitrary: a s)
  case (N n)
  then show ?case by (simp add: live_default_st_def)
next
    case (V x)
  show ?case
    unfolding afilter_st_lift_simps bind_lift_left_identity afilter.simps
      refine_ops.select_convs
    by (rule default_st_set_lift_correct[OF V.prems])
next
  case (Plus e1 e2)
  show ?case
    unfolding afilter_st_lift_simps afilter_Plus_unfold
              bind_lift_left_identity Let_def case_prod_beta
    using afilter_lift_step[OF Plus.IH(1) Plus.IH(2)[OF Plus.prems]]
    by simp
next
  case (Minus e1 e2)
  show ?case
    unfolding afilter_st_lift_simps afilter_Minus_unfold
              bind_lift_left_identity Let_def case_prod_beta
    using afilter_lift_step[OF Minus.IH(1) Minus.IH(2)[OF Minus.prems]]
    by simp
next
  case (Times e1 e2)
  show ?case
    unfolding afilter_st_lift_simps afilter_Times_unfold
              bind_lift_left_identity Let_def case_prod_beta
    using afilter_lift_step[OF Times.IH(1) Times.IH(2)[OF Times.prems]]
    by simp
next
  case (Less e1 e2) then show ?case by (simp add: live_default_st_def)
next
  case (Div e1 e2) then show ?case by (simp add: live_default_st_def)
next
  case (Mod e1 e2) then show ?case by (simp add: live_default_st_def)
next
  case (LessEq e1 e2) then show ?case by (simp add: live_default_st_def)
next
  case (Greater e1 e2) then show ?case by (simp add: live_default_st_def)
next
  case (GreaterEq e1 e2) then show ?case by (simp add: live_default_st_def)
next
  case (Eq e1 e2) then show ?case by (simp add: live_default_st_def)
next
  case (NotEq e1 e2) then show ?case by (simp add: live_default_st_def)
next
  case (Not e) then show ?case by (simp add: live_default_st_def)
next
  case (And e1 e2) then show ?case by (simp add: live_default_st_def)
next
  case (Or e1 e2) then show ?case by (simp add: live_default_st_def)
qed

lemma bfilter_st_lift_correct:
  fixes s :: "'a default_st"
  assumes "live_default_st 𝒢 s"
    shows "ϱ𝒢 (bfilter_st_lift 𝒢 b res (Lifted s)) =
         bfilter_lifted b res (ϱ𝒢 s)"
using assms proof (induction b arbitrary: res s)
  case (N n)
  show ?case
    using N.prems
    by (simp add: Let_def case_prod_beta bind_lift_left_identity
        afilter_st_lift_correct[OF N.prems] live_default_st_def)
next
  case (V x)
  show ?case
    by (simp add: Let_def case_prod_beta bind_lift_left_identity
        default_st_set_lift_correct[OF V.prems] fun_upd_def)
next
  case (Plus e1 e2)
```

```
  show ?case
    by (simp add: Let_def case_prod_beta bind_lift_left_identity
        afilter_lift_step[OF afilter_st_lift_correct afilter_st_lift_correct[OF Plus.prems]])
next
  case (Minus e1 e2)
  show ?case
    by (simp add: Let_def case_prod_beta bind_lift_left_identity
        afilter_lift_step[OF afilter_st_lift_correct afilter_st_lift_correct[OF Minus.prems]])
next
  case (Times e1 e2)
  show ?case
    by (simp add: Let_def case_prod_beta bind_lift_left_identity
        afilter_lift_step[OF afilter_st_lift_correct afilter_st_lift_correct[OF Times.prems]])
next
  case (Div e1 e2)
  then show ?case
    by (simp add: Let_def case_prod_beta bind_lift_left_identity live_default_st_def)
next
  case (Mod e1 e2)
  then show ?case
    by (simp add: Let_def case_prod_beta bind_lift_left_identity live_default_st_def)
next
  case (Not b)
  then show ?case by (simp add: Not.IH)
next
    case (And b1 b2)
  show ?case
  proof (cases res)
    case True
    then show ?thesis
      using bfilter_lift_bind_step[OF And.IH(1) And.IH(2)[OF And.prems]]
      by simp
  next
    case False
    have g1: "map_lift ϱ_𝒢
                (if feasible b1 False (ϱ_𝒢 s)
                 then bfilter_st_lift 𝒢 b1 False (Lifted s) else Bot)
              = (if feasible b1 False (ϱ_𝒢 s)
                 then bfilter_lifted b1 False (ϱ_𝒢 s) else Bot)"
      by (rule bfilter_lift_gate_step[OF And.IH(1)[OF And.prems]])
    have g2: "map_lift ϱ_𝒢
                (if feasible b2 False (ϱ_𝒢 s)
                 then bfilter_st_lift 𝒢 b2 False (Lifted s) else Bot)
              = (if feasible b2 False (ϱ_𝒢 s)
                 then bfilter_lifted b2 False (ϱ_𝒢 s) else Bot)"
      by (rule bfilter_lift_gate_step[OF And.IH(2)[OF And.prems]])
    show ?thesis
      using False
      by (simp add: g1 g2)
  qed
next
  case (Or b1 b2)
  show ?case
  proof (cases res)
    case True
    have g1: "map_lift ϱ_𝒢
                (if feasible b1 True (ϱ_𝒢 s)
                 then bfilter_st_lift 𝒢 b1 True (Lifted s) else Bot)
              = (if feasible b1 True (ϱ_𝒢 s)
                 then bfilter_lifted b1 True (ϱ_𝒢 s) else Bot)"
      by (rule bfilter_lift_gate_step[OF Or.IH(1)[OF Or.prems]])
    have g2: "map_lift ϱ_𝒢
                (if feasible b2 True (ϱ_𝒢 s)
                 then bfilter_st_lift 𝒢 b2 True (Lifted s) else Bot)
              = (if feasible b2 True (ϱ_𝒢 s)
                 then bfilter_lifted b2 True (ϱ_𝒢 s) else Bot)"
      by (rule bfilter_lift_gate_step[OF Or.IH(2)[OF Or.prems]])
    show ?thesis
      using True
      by (simp add: g1 g2)
  next
    case False
    then show ?thesis
      using bfilter_lift_bind_step[OF Or.IH(1) Or.IH(2)[OF Or.prems]]
      by simp
  qed
```

```
next
    case (Less e1 e2)
  show ?case
    by (simp add: Let_def case_prod_beta bind_lift_left_identity
        afilter_lift_step[OF afilter_st_lift_correct afilter_st_lift_correct[OF Less.prems]])
next
  case (LessEq e1 e2)
  show ?case
    by (simp add: Let_def case_prod_beta bind_lift_left_identity
        afilter_lift_step[OF afilter_st_lift_correct afilter_st_lift_correct[OF LessEq.prems]])
next
  case (Greater e1 e2)
  show ?case
    by (simp add: Let_def case_prod_beta bind_lift_left_identity
        afilter_lift_step[OF afilter_st_lift_correct afilter_st_lift_correct[OF Greater.prems]])
next
  case (GreaterEq e1 e2)
  show ?case
    by (simp add: Let_def case_prod_beta bind_lift_left_identity
        afilter_lift_step[OF afilter_st_lift_correct afilter_st_lift_correct[OF GreaterEq.prems]])
next
  case (Eq e1 e2)
    show ?case
      by (simp add: Let_def case_prod_beta bind_lift_left_identity
          afilter_lift_step[OF afilter_st_lift_correct afilter_st_lift_correct[OF Eq.prems]])
next
  case (NotEq e1 e2)
    show ?case
      by (simp add: Let_def case_prod_beta bind_lift_left_identity
          afilter_lift_step[OF afilter_st_lift_correct afilter_st_lift_correct[OF NotEq.prems]])
qed
```

`branch_st` is `branch`'s executable `default_st` mirror. It carries no runtime liveness test: `live_default_st`/`is_empty_state` have no code equation by design (they existentially quantify over `vname`), so an executable definition must never mention them. `branch_st_commute` therefore only holds on live inputs – a live-input contract, not a gap. Production never calls `tf_st` on a dead payload: `transfer_lift` already normalizes every intermediate result through `normalize_lift`, so a dead result becomes structural `Bot` before it can be handed to the next step's `tf_st`. On a dead input `bfilter_st_lift`'s leaf updates (`default_st_set_lift`) may fail to rediscover a pre-existing, untouched witness-bottom location the way `bfilter_lifted_witness_bottom` guarantees at the specification level, so `branch_st`'s raw result need not match `branch`'s there; the lifted commutation theorem the solver actually uses (`step_lift_commute`) closes this gap from the normalization invariant instead of a runtime check.

```
definition branch_st ::
  "(vname => bool) => exp => bool => 'a default_st => 'a default_st"
where
  "branch_st 𝒢 e pol s =
     (if feasible_with aval_abs ops e pol (ϱ𝒢 s)
      then collapse_lift (bfilter_st_lift_with aval_abs ops 𝒢 e pol (Lifted s))
      else bot)"

lemma branch_st_with_ops: "branch_st_with aval_abs ops = branch_st"
  by (simp add: fun_eq_iff branch_st_with_def branch_st_def)

lemma branch_st_commute:
  assumes "live_default_st 𝒢 s"
  shows
    "ϱ𝒢 (branch_st 𝒢 e pol s) =
       branch e pol (ϱ𝒢 s)"
  proof (cases "feasible e pol (ϱ𝒢 s)")
  case True
  have eq: "ϱ𝒢 (bfilter_st_lift 𝒢 e pol (Lifted s)) =
              bfilter_lifted e pol (ϱ𝒢 s)"
    by (rule bfilter_st_lift_correct[OF assms])
  have "ϱ𝒢 (collapse_lift (bfilter_st_lift 𝒢 e pol (Lifted s)))
          = collapse_lift (map_lift ϱ𝒢
                            (bfilter_st_lift 𝒢 e pol (Lifted s)))"
    by (rule collapse_lift_map_lift[where f = "default_st_to_fun 𝒢",
          OF default_st_to_fun_bot, symmetric])
  also have "... = collapse_lift (bfilter_lifted e pol (ϱ𝒢 s))"
    using eq by simp
    finally show ?thesis
      using True
      by (simp add: branch_st_def branch_def branch_lifted_def
          bfilter_st_lift_def [symmetric])
next
```

```
  case False
  then show ?thesis
    by (simp add: branch_st_def branch_def branch_lifted_def)
qed

end

unbundle no default_st_syntax

end
```

10.6 Nonrel_Registration

```
theory Nonrel_Registration
  imports
    Nonrelational_Transfer
    "Voblint_Result.DG_Live_Unknowns"
    "Voblint_Framework.Call_String_Context"
    "Voblint_Solver.Globals_Rule"
begin
```

Registering a non-relational bundle with the pipeline

`dg_analysis_execI` leaves a non-relational domain four obligations: the context policy's route agreement and seed key, the solver, and the initial state. The first three belong to the context policy and the solver, not to the domain, so they are discharged here once per policy. A domain supplies its bundle and its initial state with the coverage lemma, and one interpretation of the locale for a policy registers it there for every update rule `r` (and, for call strings, every bound `k`).

```
locale nonrel_registration = sound_nonrelational_ops ops
  for ops :: "'a::numeric_domain nonrelational_ops" +
  fixes init_st :: "'a default_st"
  assumes init_gamma: "⋀𝒢. cinit_stores 𝒢 ⊆ default_st_gamma 𝒢 init_st"
```

At the unit context

One context per callee: every activation is keyed by `()`, so the route reads nothing of the entered state and agrees with its abstraction trivially.

```
locale nonrel_unit_rule = nonrel_registration ops init_st for ops init_st +
  fixes r :: globals_rule
begin

sublocale dg_analysis_exec
    "generic_tf_st_for ops" "generic_enter_st_for ops" init_st
    "Analysis_Global ()" Analysis_Global Activation_Seed "λ_. route_unit" "()"
    "TD_side_rule_Interp_solve r"
    "TD_side_rule_Interp.solve_dom TYPE((unit, unit) global_unknown)
       TYPE(('a default_st lifted, 'a default_st lifted) dg_state) r"
    bot check.classify_check
    skip assign special_transfer backward.branch body ret
    enter_ci_for event "λ_. route_unit"
    "TD_side_rule_Interp_solve_c r"
  by (rule dg_analysis_execI) (simp_all add: td_certified_solver init_gamma)

end
```

At the entry-state context

An activation is keyed by the abstract values its formals hold on entry. The executable route reads them off the stored state, and `exec_formals_route_commute` says it agrees with the route on the abstraction.

```
locale nonrel_es_rule = nonrel_registration ops init_st for ops init_st +
  fixes r :: globals_rule
begin

sublocale dg_analysis_exec
    "generic_tf_st_for ops" "generic_enter_st_for ops" init_st
    "Analysis_Global ()" Analysis_Global Activation_Seed exec_formals_route "[]"
    "TD_side_rule_Interp_solve r"
    "TD_side_rule_Interp.solve_dom TYPE((unit, 'a list) global_unknown)
       TYPE(('a default_st lifted, 'a default_st lifted) dg_state) r"
    bot check.classify_check
    skip assign special_transfer backward.branch body ret
    enter_ci_for event "λ_. formals_route_lifted_gen"
    "TD_side_rule_Interp_solve_c r"
  by (rule dg_analysis_execI)
     (simp_all add: exec_formals_route_commute[symmetric] td_certified_solver init_gamma)

end
```

At the call-string context

An activation is keyed by its last `k` call sites. The route reads only the call site, never the entered state, which is `cs_route_indep_of_data`.

```
locale nonrel_cs_rule = nonrel_registration ops init_st for ops init_st +
  fixes k :: nat and r :: globals_rule
begin

sublocale dg_analysis_exec
    "generic_tf_st_for ops" "generic_enter_st_for ops" init_st
    Call_String_Context.Global "λ_. Call_String_Context.Global"
    Call_String_Context.Seed "λ_. cs_route k" "[]"
    "TD_side_rule_Interp_solve r"
    "TD_side_rule_Interp.solve_dom TYPE(call_string_gk)
       TYPE(('a default_st lifted, 'a default_st lifted) dg_state) r"
    bot check.classify_check
    skip assign special_transfer backward.branch body ret
    enter_ci_for event "λ_. cs_route k"
    "TD_side_rule_Interp_solve_c r"
  by (rule dg_analysis_execI)
     (simp_all add: cs_route_indep_of_data td_certified_solver init_gamma)

end
```

A domain discharges each policy's locale from the same two facts.

```
lemma nonrel_registrationI:
  assumes "sound_nonrelational_ops ops"
    and "⋀G. cinit_stores G ⊆ default_st_gamma G init_st"
  shows "nonrel_registration ops init_st"
  using assms by (intro nonrel_registration.intro nonrel_registration_axioms.intro)

lemma nonrel_unit_ruleI:
  "nonrel_registration ops init_st ⟹ nonrel_unit_rule ops init_st"
  by (rule nonrel_unit_rule.intro)

lemma nonrel_es_ruleI:
  "nonrel_registration ops init_st ⟹ nonrel_es_rule ops init_st"
  by (rule nonrel_es_rule.intro)

lemma nonrel_cs_ruleI:
  "nonrel_registration ops init_st ⟹ nonrel_cs_rule ops init_st"
  by (rule nonrel_cs_rule.intro)

end
```

Chapter 11

Voblint__Analysis__Sign

11.1 Sign_Warrowing

```
theory Sign_Warrowing
  imports "Voblint_Domain.Sign_Lattice"
begin
```

Sign warrowing and domain instance

Sign is finite, so widening is join and narrowing keeps the left argument.

```
definition narrow_sign_td :: "sign ⇒ sign ⇒ sign" where
  "narrow_sign_td a b = a"

instantiation sign :: warrowing begin
  definition "((a :: sign) ▽ b) = join_sign a b"
  definition "((a :: sign) △ b) = narrow_sign_td a b"
instance proof intro_classes
  fix a b :: sign
  show "a ≤ (a ▽ b)"
    unfolding less_eq_sign_def widen_sign_def by (rule join_sign_ub1)
  show "b ≤ (a ▽ b)"
    unfolding less_eq_sign_def widen_sign_def by (rule join_sign_ub2)
  show "b ≤ a ⟹ b ≤ (a △ b)"
    unfolding narrow_sign_def narrow_sign_td_def by simp
  show "b ≤ a ⟹ (a △ b) ≤ a"
    unfolding narrow_sign_def narrow_sign_td_def by simp
qed
end

instantiation sign :: numeric_domain begin
definition gamma_abs_sign [simp]: "numeric_domain_class.gamma (a :: sign) = gamma_sign a"
definition is_empty_sign [simp]: "is_empty (a :: sign) = is_bottom_sign a"
definition to_string_sign [simp]: "to_string (a :: sign) = string_of_sign a"
instance proof intro_classes
  show "numeric_domain_class.gamma (bot :: sign) = {}"
    unfolding bot_sign_def by simp
next
  show "numeric_domain_class.gamma (top :: sign) = UNIV"
    by (simp add: gamma_sign_top)
next
  fix a b :: sign
  assume H: "a ≤ b"
  show "numeric_domain_class.gamma a ⊆ numeric_domain_class.gamma b"
  proof -
    have "gamma_sign a ⊆ gamma_sign b"
      using H unfolding less_eq_sign_def by (rule gamma_sign_mono)
    then show ?thesis by simp
  qed
next
  fix a :: sign
  show "is_empty a ⟷ numeric_domain_class.gamma a = {}"
    by (simp add: is_bottom_sign_correct)
qed
end

end
```

11.2 Sign_Arithmetic

```
theory Sign_Arithmetic
  imports Sign_Warrowing "Voblint_Nonrelational.Abstract_Arithmetic"
begin
```

What arithmetic does to a known sign

Every operation here works on the seven-element lattice a variable's sign can take, from `SBot` (no value at all) through `SNeg`, `SNonPos`, `SZero`, `SNonNeg` and `SPos` to `STop` (any value). Addition and multiplication are given as complete tables over those seven cases; a combination the lattice cannot pin down — a negative plus a positive, say — lands on `STop`.

On top of the tables sit three queries that answer about the numbers a sign denotes rather than about the sign itself: `sign_lt`, `sign_eqb` and `sign_tobool` return `Some b` when every pair of concrete values agrees on `b` and `None` when they do not. Each operation and query is proved sound and monotone, which is exactly what the shared `sound_arith_ops` locale needs; interpreting it at the end yields `sign_arith.aval_dom_sound` and `sign_arith.aval_dom_mono` for whole expressions.

```
instantiation sign :: plus begin
fun plus_sign :: "sign => sign => sign" where
    "plus_sign SBot    _       = SBot"
  | "plus_sign _       SBot    = SBot"
  | "plus_sign SNeg    SNeg    = SNeg"
  | "plus_sign SNeg    SNonPos = SNeg"
  | "plus_sign SNonPos SNeg    = SNeg"
  | "plus_sign SNonPos SNonPos = SNonPos"
  | "plus_sign SPos    SPos    = SPos"
  | "plus_sign SPos    SNonNeg = SPos"
  | "plus_sign SNonNeg SPos    = SPos"
  | "plus_sign SNonNeg SNonNeg = SNonNeg"
  | "plus_sign SZero   b       = b"
  | "plus_sign a       SZero   = a"
  | "plus_sign _       _       = STop"
instance ..
end

instantiation sign :: minus begin
fun minus_sign :: "sign => sign => sign" where
    "minus_sign SBot    _       = SBot"
  | "minus_sign _       SBot    = SBot"
  | "minus_sign SNeg    SPos    = SNeg"
  | "minus_sign SNeg    SNonNeg = SNeg"
  | "minus_sign SPos    SNeg    = SPos"
  | "minus_sign SPos    SNonPos = SPos"
  | "minus_sign SNeg    SZero   = SNeg"
  | "minus_sign SPos    SZero   = SPos"
  | "minus_sign SZero   SZero   = SZero"
  | "minus_sign SZero   SNeg    = SPos"
  | "minus_sign SZero   SPos    = SNeg"
  | "minus_sign SZero   SNonNeg = SNonPos"
  | "minus_sign SZero   SNonPos = SNonNeg"
  | "minus_sign SNonNeg SZero   = SNonNeg"
  | "minus_sign SNonNeg SNeg    = SPos"
  | "minus_sign SNonNeg SNonPos = SNonNeg"
  | "minus_sign SNonPos SZero   = SNonPos"
  | "minus_sign SNonPos SPos    = SNeg"
  | "minus_sign SNonPos SNonNeg = SNonPos"
  | "minus_sign _       _       = STop"
instance ..
end

instantiation sign :: times begin
fun times_sign :: "sign => sign => sign" where
    "times_sign SBot    _       = SBot"
  | "times_sign _       SBot    = SBot"
  | "times_sign SZero   _       = SZero"
  | "times_sign _       SZero   = SZero"
  | "times_sign SNeg    SNeg    = SPos"
  | "times_sign SPos    SPos    = SPos"
  | "times_sign SNeg    SPos    = SNeg"
  | "times_sign SPos    SNeg    = SNeg"
  | "times_sign SNeg    SNonPos = SNonNeg"
  | "times_sign SNonPos SNeg    = SNonNeg"
  | "times_sign SNeg    SNonNeg = SNonPos"
```

```
  | "times_sign SNonNeg SNeg    = SNonPos"
  | "times_sign SPos    SNonNeg = SNonNeg"
  | "times_sign SNonNeg SPos    = SNonNeg"
  | "times_sign SPos    SNonPos = SNonPos"
  | "times_sign SNonPos SPos    = SNonPos"
  | "times_sign SNonNeg SNonNeg = SNonNeg"
  | "times_sign SNonNeg SNonPos = SNonPos"
  | "times_sign SNonPos SNonNeg = SNonPos"
  | "times_sign SNonPos SNonPos = SNonNeg"
  | "times_sign _       _       = STop"
instance ..
end


definition sign_div :: "sign ⇒ sign ⇒ sign" where
  "sign_div a b = (if a = SBot ∨ b = SBot then SBot else (a * b) ⊔ SZero)"

definition sign_mod :: "sign ⇒ sign ⇒ sign" where
  "sign_mod a b =
    (if a = SBot ∨ b = SBot then SBot else if b = SZero then a else a ⊔ SZero)"

lemma sign_div_sound:
  assumes "i ∈ gamma_sign a" "j ∈ gamma_sign b"
  shows "c_div i j ∈ gamma_sign (sign_div a b)"
  using assms
  by (cases a; cases b; auto simp: sign_div_def sup_sign_def
      intro: c_div_nonneg c_div_nonpos)

lemma sign_mod_sound:
  assumes "i ∈ gamma_sign a" "j ∈ gamma_sign b"
  shows "c_mod i j ∈ gamma_sign (sign_mod a b)"
  using assms
  by (cases a; cases b; auto simp: sign_mod_def sup_sign_def
      intro: c_mod_nonneg c_mod_nonpos)

lemma sign_div_mono:
  "a1 ≤ a2 ⟹ b1 ≤ b2 ⟹ sign_div a1 b1 ≤ sign_div a2 b2"
  by (cases a1; cases a2; cases b1; cases b2;
      simp add: sign_div_def sup_sign_def less_eq_sign_def)

lemma sign_mod_mono:
  "a1 ≤ a2 ⟹ b1 ≤ b2 ⟹ sign_mod a1 b1 ≤ sign_mod a2 b2"
  by (cases a1; cases a2; cases b1; cases b2;
      simp add: sign_mod_def sup_sign_def less_eq_sign_def)
```

Comparison and truthiness queries

`sign_lt`/`sign_eqb`/`sign_tobool` are Sign's three-valued `bool option` queries for `Voblint_Nonrelational.Abstract_Arithmetic`'s `sound_arith_ops` locale: `Some True`/`Some False` when the two operands' sign bounds alone decide it, `None` otherwise. Every guard here is a `sign_le` test against a fixed threshold, so downward-closedness under $\leq$ (needed for `aval_dom_mono`) follows from `sign_le`'s own transitivity, not from any extra reasoning about `SBot`.

```
fun sign_lt :: "sign ⇒ sign ⇒ bool option" where
  "sign_lt a b =
     (if sign_le a SNeg ∧ sign_le b SNonNeg then Some True
      else if sign_le a SNonPos ∧ sign_le b SPos then Some True
      else if sign_le b SNonPos ∧ sign_le a SNonNeg then Some False
      else if sign_le b SNeg ∧ sign_le a SPos then Some False
      else None)"

fun sign_eqb :: "sign ⇒ sign ⇒ bool option" where
  "sign_eqb a b =
     (if a = SZero ∧ b = SZero then Some True
      else if (sign_le a SNeg ∧ sign_le b SNonNeg) ∨ (sign_le b SNeg ∧ sign_le a SNonNeg)
           ∨ (sign_le a SPos ∧ sign_le b SNonPos) ∨ (sign_le b SPos ∧ sign_le a SNonPos)
      then Some False
      else None)"

fun sign_tobool :: "sign ⇒ bool option" where
  "sign_tobool a =
     (if sign_le a SNeg ∨ sign_le a SPos then Some True
      else if sign_le a SZero then Some False
      else None)"

lemma sign_lt_sound:
  assumes "sign_lt a b = Some c" and "i ∈ gamma_sign a" and "j ∈ gamma_sign b"
```

```
  shows "(i < j) = c"
  using assms unfolding less_eq_sign_def[symmetric]
  by (cases a; cases b; auto split: if_splits)

lemma sign_eqb_sound:
  assumes "sign_eqb a b = Some c" and "i ∈ gamma_sign a" and "j ∈ gamma_sign b"
  shows "(i = j) = c"
  using assms unfolding less_eq_sign_def[symmetric]
  by (cases a; cases b; auto split: if_splits)

lemma sign_tobool_sound:
  assumes "sign_tobool a = Some c" and "i ∈ gamma_sign a"
  shows "(i ≠ 0) = c"
  using assms unfolding less_eq_sign_def[symmetric]
  by (cases a; auto split: if_splits)

lemma sign_le_trans_below:
  fixes a1 a2 :: sign
  assumes "sign_le a2 t" and "a1 ≤ a2"
  shows "sign_le a1 t"
  using assms sign_le_trans unfolding less_eq_sign_def
  by blast

lemma sign_lt_mono:
  assumes hp: "¬ is_empty (a1::sign)" and hq: "¬ is_empty b1"
      and hab: "a1 ≤ a2" and hbb: "b1 ≤ b2"
      and hwide: "sign_lt a2 b2 = Some c"
  shows "sign_lt a1 b1 = Some c"
proof (cases c)
  case True
  then have "sign_le a2 SNeg ∧ sign_le b2 SNonNeg ∨ sign_le a2 SNonPos ∧ sign_le b2 SPos"
    using hwide by (auto split: if_splits)
  then have "sign_le a1 SNeg ∧ sign_le b1 SNonNeg ∨ sign_le a1 SNonPos ∧ sign_le b1 SPos"
    using sign_le_trans_below[OF _ hab] sign_le_trans_below[OF _ hbb] by blast
  then show ?thesis using True by auto
next
  case False
  then have "sign_le b2 SNonPos ∧ sign_le a2 SNonNeg ∨ sign_le b2 SNeg ∧ sign_le a2 SPos"
    using hwide by (auto split: if_splits)
  then have hyp: "sign_le b1 SNonPos ∧ sign_le a1 SNonNeg ∨ sign_le b1 SNeg ∧ sign_le a1 SPos"
    using sign_le_trans_below[OF _ hab] sign_le_trans_below[OF _ hbb] by blast
  show ?thesis using hyp False hp hq unfolding is_empty_sign is_bottom_sign_def
    by (cases a1; cases b1; auto)
qed

lemma sign_eqb_mono:
  assumes hp: "¬ is_empty (a1::sign)" and hq: "¬ is_empty b1"
      and hab: "a1 ≤ a2" and hbb: "b1 ≤ b2"
      and hwide: "sign_eqb a2 b2 = Some c"
  shows "sign_eqb a1 b1 = Some c"
proof (cases c)
  case True
  then have "a2 = SZero ∧ b2 = SZero" using hwide by (auto split: if_splits)
  then have "sign_le a2 SZero" and "sign_le b2 SZero"
    unfolding less_eq_sign_def[symmetric] by simp_all
  then have hle1: "sign_le a1 SZero" and hle2: "sign_le b1 SZero"
    using sign_le_trans_below[OF _ hab] sign_le_trans_below[OF _ hbb] by blast+
  have ha1: "a1 = SBot ∨ a1 = SZero" using hle1 by (cases a1) auto
  have hb1: "b1 = SBot ∨ b1 = SZero" using hle2 by (cases b1) auto
  show ?thesis using True hp hq ha1 hb1 unfolding is_empty_sign is_bottom_sign_def by auto
next
next
  case False
  then have "sign_le a2 SNeg ∧ sign_le b2 SNonNeg ∨ sign_le b2 SNeg ∧ sign_le a2 SNonNeg
             ∨ sign_le a2 SPos ∧ sign_le b2 SNonPos ∨ sign_le b2 SPos ∧ sign_le a2 SNonPos"
    using hwide by (auto split: if_splits)
  then have "sign_le a1 SNeg ∧ sign_le b1 SNonNeg ∨ sign_le b1 SNeg ∧ sign_le a1 SNonNeg
             ∨ sign_le a1 SPos ∧ sign_le b1 SNonPos ∨ sign_le b1 SPos ∧ sign_le a1 SNonPos"
    using sign_le_trans_below[OF _ hab] sign_le_trans_below[OF _ hbb] by blast
  then show ?thesis using False by auto
qed

lemma sign_tobool_mono:
  assumes "¬ is_empty (a1::sign)" and "a1 ≤ a2" and "sign_tobool a2 = Some c"
  shows "sign_tobool a1 = Some c"
  using assms unfolding is_empty_sign is_bottom_sign_def
```

```
  by (cases a1; cases a2; auto simp: less_eq_sign_def split: if_splits)
```

Abstract expression evaluation

`aval_sign` evaluates an expression bottom-up with the sign operations above. A comparison is `bot` when either operand is empty; otherwise it is the literal its three-valued query decides, or the join of `0` and `1` on `None`. These are the equations `sound_arith_ops` expects of an evaluator.

```
fun aval_sign :: "exp => (vname => sign) => sign" where
    "aval_sign (N n)        d = sign_of_int n"
  | "aval_sign (V x)        d = d x"
  | "aval_sign (Plus  a b)  d = aval_sign a d + aval_sign b d"
  | "aval_sign (Minus a b)  d = aval_sign a d - aval_sign b d"
  | "aval_sign (Times a b)  d = aval_sign a d * aval_sign b d"
  | "aval_sign (Div a b)  d = sign_div (aval_sign a d) (aval_sign b d)"
  | "aval_sign (Mod a b)  d = sign_mod (aval_sign a d) (aval_sign b d)"
  | "aval_sign (Less a b)   d =
       (if is_empty (aval_sign a d) ∨ is_empty (aval_sign b d) then bot
        else of_bool_option sign_of_int (sign_lt (aval_sign a d) (aval_sign b d)))"
  | "aval_sign (LessEq a b)   d =
       (if is_empty (aval_sign a d) ∨ is_empty (aval_sign b d) then bot
        else of_bool_option sign_of_int
               (map_option HOL.Not (sign_lt (aval_sign b d) (aval_sign a d))))"
  | "aval_sign (Greater a b)   d =
       (if is_empty (aval_sign a d) ∨ is_empty (aval_sign b d) then bot
        else of_bool_option sign_of_int (sign_lt (aval_sign b d) (aval_sign a d)))"
  | "aval_sign (GreaterEq a b)   d =
       (if is_empty (aval_sign a d) ∨ is_empty (aval_sign b d) then bot
        else of_bool_option sign_of_int
               (map_option HOL.Not (sign_lt (aval_sign a d) (aval_sign b d))))"
  | "aval_sign (NotEq a b) d =
       (if is_empty (aval_sign a d) ∨ is_empty (aval_sign b d) then bot
        else of_bool_option sign_of_int
               (map_option HOL.Not (sign_eqb (aval_sign a d) (aval_sign b d))))"
  | "aval_sign (exp.Eq a b) d =
       (if is_empty (aval_sign a d) ∨ is_empty (aval_sign b d) then bot
        else of_bool_option sign_of_int (sign_eqb (aval_sign a d) (aval_sign b d)))"
  | "aval_sign (exp.Not a)  d =
       (if is_empty (aval_sign a d) then bot
        else of_bool_option sign_of_int (map_option HOL.Not (sign_tobool (aval_sign a d))))"
  | "aval_sign (And a b)    d =
       (if is_empty (aval_sign a d) ∨ is_empty (aval_sign b d) then bot
        else of_bool_option sign_of_int
               (and_opt (sign_tobool (aval_sign a d)) (sign_tobool (aval_sign b d))))"
  | "aval_sign (Or a b)     d =
       (if is_empty (aval_sign a d) ∨ is_empty (aval_sign b d) then bot
        else of_bool_option sign_of_int
               (or_opt (sign_tobool (aval_sign a d)) (sign_tobool (aval_sign b d))))"

lemma sign_plus_sound:
  assumes "i ∈ gamma_sign a" "j ∈ gamma_sign b"
  shows "i + j ∈ gamma_sign (a + b)"
  using assms by (cases a; cases b; auto)

lemma sign_minus_sound:
  assumes "i ∈ gamma_sign a" "j ∈ gamma_sign b"
  shows "i - j ∈ gamma_sign (a - b)"
  using assms by (cases a; cases b; auto)

lemma sign_times_sound:
  assumes "i ∈ gamma_sign a" "j ∈ gamma_sign b"
  shows "i * j ∈ gamma_sign (a * b)"
  using assms by (cases a; cases b; auto simp: mult_neg_neg mult_neg_pos mult_pos_neg
                                               zero_le_mult_iff mult_le_0_iff)

lemma sign_plus_mono1:
  "a1 ≤ a2 ⟹ a1 + b ≤ a2 + (b::sign)"
  unfolding less_eq_sign_def
  by (cases a1; cases a2; cases b; simp)

lemma sign_plus_mono2:
  "b1 ≤ b2 ⟹ a + b1 ≤ a + (b2::sign)"
  unfolding less_eq_sign_def
  by (cases a; cases b1; cases b2; simp)

lemma sign_minus_mono1:
```

```
  "a1 ≤ a2 ⟹ a1 - b ≤ a2 - (b::sign)"
  unfolding less_eq_sign_def
  by (cases a1; cases a2; cases b; simp)

lemma sign_minus_mono2:
  "b1 ≤ b2 ⟹ a - b1 ≤ a - (b2::sign)"
  unfolding less_eq_sign_def
  by (cases a; cases b1; cases b2; simp)

lemma sign_times_mono1:
  "a1 ≤ a2 ⟹ a1 * b ≤ a2 * (b::sign)"
  unfolding less_eq_sign_def
  by (cases a1; cases a2; cases b; simp)

lemma sign_times_mono2:
  "b1 ≤ b2 ⟹ a * b1 ≤ a * (b2::sign)"
  unfolding less_eq_sign_def
  by (cases a; cases b1; cases b2; simp)

lemma sign_plus_combine_mono:
  "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ a1 + b1 ≤ a2 + (b2::sign)"
  by (meson order.trans sign_plus_mono1 sign_plus_mono2)

lemma sign_minus_combine_mono:
  "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ a1 - b1 ≤ a2 - (b2::sign)"
  by (meson order.trans sign_minus_mono1 sign_minus_mono2)

lemma sign_times_combine_mono:
  "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ a1 * b1 ≤ a2 * (b2::sign)"
  by (meson order.trans sign_times_mono1 sign_times_mono2)

interpretation sign_arith: mono_arith_ops
    aval_sign sign_of_int "(+)" "(-)" "(*)" sign_div sign_mod sign_lt sign_eqb sign_tobool
  apply unfold_locales
  apply (simp_all add: sign_of_int_gamma sign_plus_sound sign_minus_sound sign_times_sound
    sign_div_sound sign_mod_sound
                       sign_plus_combine_mono sign_minus_combine_mono sign_times_combine_mono
                         sign_div_mono sign_mod_mono
                       sign_lt_sound sign_eqb_sound sign_tobool_sound[unfolded truthy_def]
                       sup_sign_def
                  del: sign_lt.simps sign_eqb.simps sign_tobool.simps)
  apply (blast intro: sign_lt_mono[unfolded is_empty_sign] sign_eqb_mono[unfolded is_empty_sign]
                      sign_tobool_mono[unfolded is_empty_sign])+
  done


end
```

11.3 Sign_Backward

```
theory Sign_Backward
  imports
    Sign_Arithmetic
    "Voblint_Nonrelational.Exec_Backward"
    "Voblint_Domain.Backward_Numeric_Queries"
begin
```

Sign backward filtering

The sign meet lives with the lattice in `Voblint_Domain.Sign_Lattice`.

Backward-analysis: inverse operators

Per the backward-domain plan, `inv_less_sign` provides sign-specific refinement when a guard `e1 < e2` is known true or false. Plus/minus/times are too coarse for useful arithmetic inversion in sign, so they instantiate the shared `inv_conservative` (identity) instead of a per-domain no-op; the structural bfilter propagation (And/Or/Not/Eq) is where sign gains.

```
fun inv_less_sign :: "bool => sign => sign => sign * sign" where
    "inv_less_sign True  a1 a2 =
       (let a1' = if sign_le a2 SNonPos then a1 ⊓ SNeg else a1 ;
                a2' = if sign_le a1 SNonNeg then a2 ⊓ SPos else a2
        in (a1', a2'))"
  | "inv_less_sign False a1 a2 =
       (let a1' = if sign_le a2 SPos then a1 ⊓ SPos
                  else if sign_le a2 SNonNeg then a1 ⊓ SNonNeg
                  else a1 ;
                a2' = if sign_le a1 SNeg then a2 ⊓ SNeg
                  else if sign_le a1 SNonPos then a2 ⊓ SNonPos
                  else a2
        in (a1', a2'))"

lemma inv_less_sign_sound:
  "n1 ∈ gamma_sign a1 ⟹ n2 ∈ gamma_sign a2 ⟹ (n1 < n2) = res
   ⟹ n1 ∈ gamma_sign (fst (inv_less_sign res a1 a2))
     ∧ n2 ∈ gamma_sign (snd (inv_less_sign res a1 a2))"
  by (cases res; cases a1; cases a2;
      auto simp: less_eq_sign_def; linarith)
```

`inv_eq_sign` narrows on a guard `e1 = e2` known true or false. The true branch narrows both operands to their intersection, exactly matching what `bfilter` already computes for `Eq _ _ True` via `(⊓)` directly. The false branch only narrows a boundary singleton away from a wider operand that has `SZero` as one of its bounds: excluding `0` from `SNonNeg` leaves `SPos`, from `SNonPos` leaves `SNeg`; both directions, symmetric in which operand is at most `SZero`. Two jointly-at-most-`SZero` operands are jointly unreachable. Every other pair — including any pair where the excluded value would not sit at a representable lattice boundary — passes through unchanged: the seven-element lattice cannot express ñonzeroör an interior exclusion as a single value.

The guard tests `sign_le a SZero` (the set `{SBot, SZero}`), not literal equality `a = SZero`: `SBot` is strictly below `SZero`, so an equality guard would let a wider input at `SZero` trigger narrowing that a strictly more precise input at `SBot` would not — a genuine monotonicity failure, not merely a missed precision opportunity. The order-based guard closes that gap the same way `inv_less_sign`'s own `sign_le a2 SNonPos`-style guards do; it also makes the `SBot` case sound for free (vacuously, since `gamma_sign SBot = {}`), without `inv_less_sign`'s separate reliance on `SBot ⊓ _ = SBot`.

```
fun inv_eq_sign :: "bool => sign => sign => sign * sign" where
    "inv_eq_sign True  a1 a2 = (a1 ⊓ a2, a1 ⊓ a2)"
  | "inv_eq_sign False a1 a2 =
       (let a1' = if sign_le a1 SZero ∧ sign_le a2 SZero then SBot
                  else if sign_le a2 SZero ∧ sign_le a1 SNonNeg then a1 ⊓ SPos
                  else if sign_le a2 SZero ∧ sign_le a1 SNonPos then a1 ⊓ SNeg
                  else a1 ;
                a2' = if sign_le a1 SZero ∧ sign_le a2 SZero then SBot
                  else if sign_le a1 SZero ∧ sign_le a2 SNonNeg then a2 ⊓ SPos
                  else if sign_le a1 SZero ∧ sign_le a2 SNonPos then a2 ⊓ SNeg
                  else a2
        in (a1', a2'))"

lemma inv_eq_sign_sound:
  "n1 ∈ gamma_sign a1 ⟹ n2 ∈ gamma_sign a2 ⟹ (n1 = n2) = res
   ⟹ n1 ∈ gamma_sign (fst (inv_eq_sign res a1 a2))
```

```
    ∧ n2 ∈ gamma_sign (snd (inv_eq_sign res a1 a2))"
proof (cases res)
  case True
  assume h1: "n1 ∈ gamma_sign a1" and h2: "n2 ∈ gamma_sign a2"
    and heq: "(n1 = n2) = res"
  have "n1 = n2" using heq True by simp
  then have "n1 ∈ gamma_sign a2" using h2 by simp
  then have "n1 ∈ gamma_sign (a1 ⊓ a2)"
    using inf_sign_sound[OF h1] by simp
  then show ?thesis using True ‹n1 = n2› by simp
next
  case False
  assume h1: "n1 ∈ gamma_sign a1" and h2: "n2 ∈ gamma_sign a2"
    and heq: "(n1 = n2) = res"
  have hne: "n1 ≠ n2" using heq False by simp
  show ?thesis
    using h1 h2 hne False
    by (cases a1; cases a2;
        auto simp: less_eq_sign_def Let_def)
qed

lemma inv_eq_sign_mono:
  assumes "a1 ≤ (a1' :: sign)"
      and "a2 ≤ a2'"
  shows
    "le_pair (inv_eq_sign r a1 a2) (inv_eq_sign r a1' a2')"
proof (cases r)
  case True
  then show ?thesis using assms by (simp add: le_infI1 le_infI2)
next
  case False
  then show ?thesis
    using assms unfolding less_eq_sign_def
    by (cases a1; cases a1'; simp; cases a2; cases a2'; simp add: Let_def)
qed
```

Monotonicity of `inv_less_sign` needs a case split on both the guard's truth value and which side of the shared narrowing threshold each operand sits; `narrow1_mono`/`narrow2_mono` factor the `if`-cascade shape common to both branches so the case split is done once, generically in a `semilattice_inf` operand, rather than twice inline.

```
lemma sign_le_iff:
  "sign_le a b ⟷ a ≤ b"
  by (cases a; cases b; auto simp: less_eq_sign_def)

lemma narrow1_mono:
  fixes x x' y y' c d :: "'a::semilattice_inf"
  assumes xx': "x ≤ x'"
      and yy': "y ≤ y'"
  shows
    "(if y ≤ c then x ⊓ d else x)
      ≤ (if y' ≤ c then x' ⊓ d else x')"
  using xx' yy' inf_mono
  by auto

lemma narrow2_mono:
  fixes x x' y y' c1 c2 :: "'a::semilattice_inf"
  assumes xx': "x ≤ x'"
      and yy': "y ≤ y'"
      and cc': "c1 ≤ c2"
  shows
    "(if y ≤ c1 then x ⊓ c1
      else if y ≤ c2 then x ⊓ c2
      else x)
      ≤
     (if y' ≤ c1 then x' ⊓ c1
      else if y' ≤ c2 then x' ⊓ c2
      else x')"
  using xx' yy' cc' inf_mono
  by (auto simp add: inf.coboundedI1 inf.coboundedI2)

lemma inv_less_sign_mono:
  assumes A1: "a1 ≤ (a1' :: sign)"
      and A2: "a2 ≤ a2'"
  shows
    "le_pair (inv_less_sign r a1 a2) (inv_less_sign r a1' a2')"
proof (cases r)
  case True
```

```
  have fst_mono:
    "(if a2 ≤ SNonPos then a1 ⊓ SNeg else a1)
      ≤
     (if a2' ≤ SNonPos then a1' ⊓ SNeg else a1')"
    using narrow1_mono[OF A1 A2, of SNonPos SNeg] .

  have snd_mono:
    "(if a1 ≤ SNonNeg then a2 ⊓ SPos else a2)
      ≤
     (if a1' ≤ SNonNeg then a2' ⊓ SPos else a2')"
    using narrow1_mono[OF A2 A1, of SNonNeg SPos] .

  show ?thesis
    using True fst_mono snd_mono
    by (simp only: inv_less_sign.simps Let_def sign_le_iff prod.sel)
next
  case False

  have pos_order: "(SPos :: sign) ≤ SNonNeg"
    by (simp add: less_eq_sign_def)

  have neg_order: "(SNeg :: sign) ≤ SNonPos"
    by (simp add: less_eq_sign_def)

  have fst_mono:
    "(if a2 ≤ SPos then a1 ⊓ SPos
      else if a2 ≤ SNonNeg then a1 ⊓ SNonNeg
      else a1)
      ≤
     (if a2' ≤ SPos then a1' ⊓ SPos
      else if a2' ≤ SNonNeg then a1' ⊓ SNonNeg
      else a1')"
    using narrow2_mono[OF A1 A2, of SPos SNonNeg] pos_order by simp

  have snd_mono:
    "(if a1 ≤ SNeg then a2 ⊓ SNeg
      else if a1 ≤ SNonPos then a2 ⊓ SNonPos
      else a2)
      ≤
     (if a1' ≤ SNeg then a2' ⊓ SNeg
      else if a1' ≤ SNonPos then a2' ⊓ SNonPos
      else a2')"
    using narrow2_mono[OF A2 A1, of SNeg SNonPos] neg_order by simp

  show ?thesis
    using False fst_mono snd_mono
    by (simp only: inv_less_sign.simps Let_def sign_le_iff prod.sel)
qed
```

The refinement operations and their certificate

Sign's refinement choices: the lattice meet, the sign-specific comparison inverses, and the conservative identity for arithmetic. The guard filters and the branch are not stated here; `Sign_Transfer` derives them from this record and the evaluator.

```
definition sign_refine_ops :: "sign refine_ops" where
  "sign_refine_ops =
     ⦇r_tobool = sign_tobool, r_inv_less = inv_less_sign, r_inv_eq = inv_eq_sign,
      r_inv_plus = inv_conservative, r_inv_minus = inv_conservative,
      r_inv_times = inv_conservative, r_intersect = inf⦈"

lemma sign_refine_ops_simps [simp]:
  "r_tobool sign_refine_ops = sign_tobool"
  "r_inv_less sign_refine_ops = inv_less_sign"
  "r_inv_eq sign_refine_ops = inv_eq_sign"
  "r_inv_plus sign_refine_ops = inv_conservative"
  "r_inv_minus sign_refine_ops = inv_conservative"
  "r_inv_times sign_refine_ops = inv_conservative"
  "r_intersect sign_refine_ops = inf"
  by (simp_all add: sign_refine_ops_def)
```

The certificate discharges soundness, monotonicity, and reductiveness together against `mono_refinement`. Each `inv_*`'s mono/reductive obligation is one `le_pair` fact, transparent notation for the componentwise $\wedge$ the per-operator lemmas above already prove.

```
lemma sign_backward_domain:
  "mono_refinement inf aval_sign sign_tobool
     inv_less_sign inv_eq_sign inv_conservative inv_conservative inv_conservative"
proof unfold_locales
qed (use sign_tobool_mono in ‹simp_all add: inf_sign_sound inv_less_sign_sound
       inv_eq_sign_sound inv_conservative_def sign_tobool_sound inf_mono sign_arith.aval_dom_mono
       inv_less_sign_mono inv_eq_sign_mono le_infI1 le_infI2›)

end
```

11.4 Sign_Special

```
theory Sign_Special
  imports Sign_Arithmetic "Voblint_Nonrelational.Special_Ops"
begin
```

Sign: special-call semantics

`sign_min`/`sign_max` exist solely as the abstract implementation of the `Min`/`Max` special calls (`VIMP_Special.special_call`); they are not part of Sign's core arithmetic interface the way `plus_sign`/`times_sign` are (VIMP's expression language has no `Min`/`Max` arithmetic operator, only the special-call form). If a future consumer needs them independently of special-call dispatch, that is the point to reconsider their home, not before.

`sign_min`/`sign_max` abstract the two-argument `Min`/`Max` special calls: the same brute-force per-constructor table as `plus_sign`/`minus_sign`/`times_sign`, sound and, where the two argument signs pin down which side realizes the extremum, exact (e.g. `sign_min SNeg _ = SNeg`: the minimum cannot exceed its negative left argument, so it remains negative regardless of the right side).

```
fun sign_min :: "sign => sign => sign" where
    "sign_min SBot    _       = SBot"
  | "sign_min _       SBot    = SBot"
  | "sign_min SNeg    SNeg    = SNeg"
  | "sign_min SNonPos SNonPos = SNonPos"
  | "sign_min SZero   SZero   = SZero"
  | "sign_min SNonNeg SNonNeg = SNonNeg"
  | "sign_min SPos    SPos    = SPos"
  | "sign_min STop    STop    = STop"
  | "sign_min SNeg    SNonPos = SNeg"    | "sign_min SNonPos SNeg    = SNeg"
  | "sign_min SNeg    SZero   = SNeg"    | "sign_min SZero   SNeg    = SNeg"
  | "sign_min SNeg    SNonNeg = SNeg"    | "sign_min SNonNeg SNeg    = SNeg"
  | "sign_min SNeg    SPos    = SNeg"    | "sign_min SPos    SNeg    = SNeg"
  | "sign_min SNeg    STop    = SNeg"    | "sign_min STop    SNeg    = SNeg"
  | "sign_min SNonPos SZero   = SNonPos" | "sign_min SZero   SNonPos = SNonPos"
  | "sign_min SNonPos SNonNeg = SNonPos" | "sign_min SNonNeg SNonPos = SNonPos"
  | "sign_min SNonPos SPos    = SNonPos" | "sign_min SPos    SNonPos = SNonPos"
  | "sign_min SNonPos STop    = SNonPos" | "sign_min STop    SNonPos = SNonPos"
  | "sign_min SZero   SNonNeg = SZero"   | "sign_min SNonNeg SZero   = SZero"
  | "sign_min SZero   SPos    = SZero"   | "sign_min SPos    SZero   = SZero"
  | "sign_min SZero   STop    = SNonPos" | "sign_min STop    SZero   = SNonPos"
  | "sign_min SNonNeg SPos    = SNonNeg" | "sign_min SPos    SNonNeg = SNonNeg"
  | "sign_min SNonNeg STop    = STop"    | "sign_min STop    SNonNeg = STop"
  | "sign_min SPos    STop    = STop"    | "sign_min STop    SPos    = STop"

fun sign_max :: "sign => sign => sign" where
    "sign_max SBot    _       = SBot"
  | "sign_max _       SBot    = SBot"
  | "sign_max SNeg    SNeg    = SNeg"
  | "sign_max SNonPos SNonPos = SNonPos"
  | "sign_max SZero   SZero   = SZero"
  | "sign_max SNonNeg SNonNeg = SNonNeg"
  | "sign_max SPos    SPos    = SPos"
  | "sign_max STop    STop    = STop"
  | "sign_max SNeg    SNonPos = SNonPos" | "sign_max SNonPos SNeg    = SNonPos"
  | "sign_max SNeg    SZero   = SZero"   | "sign_max SZero   SNeg    = SZero"
  | "sign_max SNeg    SNonNeg = SNonNeg" | "sign_max SNonNeg SNeg    = SNonNeg"
  | "sign_max SNeg    SPos    = SPos"    | "sign_max SPos    SNeg    = SPos"
  | "sign_max SNeg    STop    = STop"    | "sign_max STop    SNeg    = STop"
  | "sign_max SNonPos SZero   = SZero"   | "sign_max SZero   SNonPos = SZero"
  | "sign_max SNonPos SNonNeg = SNonNeg" | "sign_max SNonNeg SNonPos = SNonNeg"
  | "sign_max SNonPos SPos    = SPos"    | "sign_max SPos    SNonPos = SPos"
  | "sign_max SNonPos STop    = STop"    | "sign_max STop    SNonPos = STop"
  | "sign_max SZero   SNonNeg = SNonNeg" | "sign_max SNonNeg SZero   = SNonNeg"
  | "sign_max SZero   SPos    = SPos"    | "sign_max SPos    SZero   = SPos"
  | "sign_max SZero   STop    = SNonNeg" | "sign_max STop    SZero   = SNonNeg"
  | "sign_max SNonNeg SPos    = SPos"    | "sign_max SPos    SNonNeg = SPos"
  | "sign_max SNonNeg STop    = SNonNeg" | "sign_max STop    SNonNeg = SNonNeg"
  | "sign_max SPos    STop    = SPos"    | "sign_max STop    SPos    = SPos"

lemma sign_min_sound:
  assumes "i ∈ gamma_sign a" "j ∈ gamma_sign b"
  shows "min i j ∈ gamma_sign (sign_min a b)"
  using assms by (cases a; cases b; auto)

lemma sign_max_sound:
  assumes "i ∈ gamma_sign a" "j ∈ gamma_sign b"
```

```
  shows "max i j ∈ gamma_sign (sign_max a b)"
  using assms by (cases a; cases b; auto)

lemma sign_min_mono1:
  "a1 ≤ a2 ⟹ sign_min a1 b ≤ sign_min a2 (b::sign)"
  unfolding less_eq_sign_def
  by (cases a1; cases a2; cases b; simp)

lemma sign_min_mono2:
  "b1 ≤ b2 ⟹ sign_min a b1 ≤ sign_min a (b2::sign)"
  unfolding less_eq_sign_def
  by (cases a; cases b1; cases b2; simp)

lemma sign_min_combine_mono:
  "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ sign_min a1 b1 ≤ sign_min a2 (b2::sign)"
  by (meson order.trans sign_min_mono1 sign_min_mono2)

lemma sign_max_mono1:
  "a1 ≤ a2 ⟹ sign_max a1 b ≤ sign_max a2 (b::sign)"
  unfolding less_eq_sign_def
  by (cases a1; cases a2; cases b; simp)

lemma sign_max_mono2:
  "b1 ≤ b2 ⟹ sign_max a b1 ≤ sign_max a (b2::sign)"
  unfolding less_eq_sign_def
  by (cases a; cases b1; cases b2; simp)

lemma sign_max_combine_mono:
  "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ sign_max a1 b1 ≤ sign_max a2 (b2::sign)"
  by (meson order.trans sign_max_mono1 sign_max_mono2)
```

Special-call dispatch

`special_sign` havocs on `Nondet_Int` and applies `sign_min`/`sign_max` otherwise. `special_sign_eq_transfer` shows it equals the generic transfer of the `mono_minmax_ops` interpretation at `sign_special_ops`.

```
fun special_sign ::
    "special_call => vname => (vname => sign) => (vname => sign)"
where
  "special_sign Nondet_Int x d = d(x := STop)"
| "special_sign (Min a b) x d = d(x := sign_min (aval_sign a d) (aval_sign b d))"
| "special_sign (Max a b) x d = d(x := sign_max (aval_sign a d) (aval_sign b d))"

definition sign_special_ops :: "sign special_ops" where
  "sign_special_ops = (| special_min = sign_min, special_max = sign_max |)"

interpretation sign_special: mono_minmax_ops sign_special_ops aval_sign
  by unfold_locales
     (auto simp: sign_special_ops_def gamma_sign_top
           intro: sign_min_sound sign_max_sound sign_min_combine_mono sign_max_combine_mono)

lemma sign_special_ops_min [simp]: "special_min sign_special_ops = sign_min"
  by (simp add: sign_special_ops_def)

lemma sign_special_ops_max [simp]: "special_max sign_special_ops = sign_max"
  by (simp add: sign_special_ops_def)

lemma special_sign_eq_transfer: "special_sign sc x d = sign_special.special_transfer sc x d"
  by (cases sc) (simp_all add: top_sign_def)

end
```

11.5 Sign_Numeric_Queries

```
theory Sign_Numeric_Queries
  imports Sign_Backward
begin
```

Sign interpretation of the generic numeric-query interface

`sound_numeric_queries` (`Voblint_Domain.Numeric_Queries`) is the reusable interface; this theory supplies its Sign instance. None of the four query functions is a hand-built table: all four are Sign's instance of the generic derivations in `sound_inverse_ops`'s own context — `sign_less_true`/`sign_less_false` read off `inv_less_sign`, `sign_eq_true` reads off `sign_less_false` in both directions, and `sign_eq_false` reads off `(⊓)` collapsing to `SBot` — so their soundness is `sound_inverse_ops`'s, instantiated by `sign_backward_domain`.

`sign_less_true_eq`/`sign_less_false_eq`/`sign_eq_true_eq`/`sign_eq_false_eq` below restate each derived predicate as an explicit truth table over `inv_less_sign`/`(⊓)`. They are the code equations of the four predicates, and they guard precision: a future change to `inv_less_sign`/`inv_eq_sign`/`(⊓)` that silently narrows or widens what these four predicates classify breaks one of these four proofs, at the seven-element lattice, rather than surfacing only as a precision regression in a downstream analysis.

Comparison judgments

`sign_less_true a b` holds when every concrete pair `i ∈ gamma_sign a`, `j ∈ gamma_sign b` satisfies `i < j`; `sign_less_false a b` when every such pair satisfies `¬ i < j`. The two are not complements of each other: overlapping abstractions (e.g. `SNonPos` against `SNonNeg`) make both false, meaning unknown. `SBot` on either side makes both vacuously true.

```
definition sign_less_true :: "sign ⇒ sign ⇒ bool" where
  "sign_less_true = sound_inverse_ops.less_true inv_less_sign"

lemma sign_less_true_sound:
  assumes "sign_less_true a b" and "i ∈ gamma_sign a" and "j ∈ gamma_sign b"
  shows "i < j"
  using assms sound_inverse_ops.less_true_sound[OF
      sign_backward_domain[THEN mono_refinement.axioms(1), THEN sound_refinement.axioms(4)],
      of a b i j]
  by (simp add: sign_less_true_def)

definition sign_less_false :: "sign ⇒ sign ⇒ bool" where
  "sign_less_false = sound_inverse_ops.less_false inv_less_sign"

lemma sign_less_false_sound:
  assumes "sign_less_false a b" and "i ∈ gamma_sign a" and "j ∈ gamma_sign b"
  shows "¬ i < j"
  using assms sound_inverse_ops.less_false_sound[OF
      sign_backward_domain[THEN mono_refinement.axioms(1), THEN sound_refinement.axioms(4)],
      of a b i j]
  by (simp add: sign_less_false_def)

lemma sign_less_true_eq [code]: "sign_less_true a b ⟷
  (fst (inv_less_sign False a b) = SBot ∨ snd (inv_less_sign False a b) = SBot)"
  by (cases a; cases b;
      simp add: sign_less_true_def sound_inverse_ops.less_true_def[OF
        sign_backward_domain[THEN mono_refinement.axioms(1), THEN sound_refinement.axioms(4)]]
              bot_sign_def is_bottom_sign_def)

lemma sign_less_false_eq [code]: "sign_less_false a b ⟷
  (fst (inv_less_sign True a b) = SBot ∨ snd (inv_less_sign True a b) = SBot)"
  by (cases a; cases b;
      simp add: sign_less_false_def sound_inverse_ops.less_false_def[OF
        sign_backward_domain[THEN mono_refinement.axioms(1), THEN sound_refinement.axioms(4)]]
              bot_sign_def is_bottom_sign_def)
```

Equality judgments

Only `SZero` concretizes to a singleton, so equality is provable exactly there; two abstractions are provably unequal exactly when their concretizations are disjoint. Neither table is hand-built: `sign_eq_true` is Sign's instance of the `eq_true` derivation off `less_false` in both directions (integer trichotomy), and `sign_eq_false` is Sign's instance of the `eq_false` derivation off `(⊓)` collapsing to `SBot` (disjoint concretizations).

```
definition sign_eq_true :: "sign ⇒ sign ⇒ bool" where
```

```
  "sign_eq_true = sound_inverse_ops.eq_true inv_less_sign"
```

The derived definition unfolds through `less_false`'s global constant, so the code equation is stated over the Sign-level name directly.

```
lemma sign_eq_true_eq [code]:
  "sign_eq_true a b ⟷ (sign_less_false a b ∧ sign_less_false b a)"
  by (simp add: sign_eq_true_def sign_less_false_def
    sound_inverse_ops.eq_true_def[OF
        sign_backward_domain[THEN mono_refinement.axioms(1), THEN sound_refinement.axioms(4)]])

lemma sign_eq_true_sound:
  assumes "sign_eq_true a b" and "i ∈ gamma_sign a" and "j ∈ gamma_sign b"
  shows "i = j"
  using assms sound_inverse_ops.eq_true_sound[OF
      sign_backward_domain[THEN mono_refinement.axioms(1), THEN sound_refinement.axioms(4)],
      of a b i j]
  by (simp add: sign_eq_true_def)

definition sign_eq_false :: "sign ⇒ sign ⇒ bool" where
  "sign_eq_false = sound_inverse_ops.eq_false inf"

lemma sign_eq_false_eq [code]: "sign_eq_false a b ⟷ a ⊓ b = SBot"
  by (cases a; cases b;
      simp add: sign_eq_false_def bot_sign_def sound_inverse_ops.eq_false_def[OF
        sign_backward_domain[THEN mono_refinement.axioms(1), THEN sound_refinement.axioms(4)]]
        is_bottom_sign_def)

lemma sign_eq_false_sound:
  assumes "sign_eq_false a b" and "i ∈ gamma_sign a" and "j ∈ gamma_sign b"
  shows "i ≠ j"
  using assms sound_inverse_ops.eq_false_sound[OF
      sign_backward_domain[THEN mono_refinement.axioms(1), THEN sound_refinement.axioms(4)],
      of a b i j]
  by (simp add: sign_eq_false_def)
```

Goblint-style optional-Boolean queries

`sign_less`/`sign_eq` are the generic packaging of the four judgments above, so one interpretation both introduces them and discharges the query interface. The **defines** clause is what keeps them ordinary top-level constants, with the code equation the check layer needs.

```
global_interpretation sign_numeric_queries:
  numeric_query_judgments sign_less_true sign_less_false sign_eq_true sign_eq_false
  defines sign_less = sign_numeric_queries.query_less
    and sign_eq = sign_numeric_queries.query_eq
  by unfold_locales
     (auto intro: sign_less_true_sound sign_eq_true_sound
           dest: sign_less_false_sound sign_eq_false_sound)


end
```

11.6 Sign_Transfer

```
theory Sign_Transfer
  imports
    Sign_Special
    Sign_Numeric_Queries
    "Voblint_Nonrelational.Nonrelational_Transfer"
begin
```

What each kind of CFG edge does to a sign store

A sign store maps every variable to one of the seven signs. How an edge of a compiled graph moves such a store is not a Sign question: an assignment re-evaluates its right-hand side and overwrites the target, a guard runs the backward branch, a return writes into the return variable, procedure entry resets the callee frame and binds the formals, and skip, body entry and check observation leave the store alone. That is `sound_nonrelational_ops`, proved sound and monotone once for any domain that gives one abstract value per variable.

So this theory only states Sign's bundle of primitive choices and certifies it. `sign_ops` collects `aval_sign`, the comparison queries `sign_less`/`sign_eq`, `sign_refine_ops`, `sign_special_ops` and `STop`; each capability is already certified in the theory that introduces it. The guard filters, the branch, the check classifier and every transfer function are derived from the bundle by the one interpretation below, which names the results Sign's callers apply. `Sign_Exec`'s executable mirror runs on the same bundle.

`special_sign` is the one name that does not come from the interpretation. It is Sign's own `fun` in `Voblint_Analysis_Sign.Sign_S` written out case by case because a reader wants to see the three special calls, so the interpretation rewrites the locale's dispatch to it rather than introducing a second name for the same function.

```
lemma sign_check_domain: "sound_check_query sign_less sign_eq gamma_state aval_sign"
  by unfold_locales (rule sign_arith.aval_abs_sound)

definition sign_ops :: "sign nonrelational_ops" where
  "sign_ops = (| n_aval = aval_sign, n_query = (|q_less = sign_less, q_eq = sign_eq|),
                 n_refine = sign_refine_ops, n_special = sign_special_ops |)"

lemma sign_ops_simps [simp]:
  "n_aval sign_ops = aval_sign"
  "q_less (n_query sign_ops) = sign_less"
  "q_eq (n_query sign_ops) = sign_eq"
  "n_refine sign_ops = sign_refine_ops"
  "n_special sign_ops = sign_special_ops"
  by (simp_all add: sign_ops_def)

global_interpretation sign_tf: mono_nonrelational_ops sign_ops
  rewrites "(top :: sign) = STop"
    and "sound_minmax_ops.special_transfer (n_special sign_ops) (n_aval sign_ops) = special_sign"
  defines assign_sign = sign_tf.assign
    and skip_sign = sign_tf.skip
    and body_sign = sign_tf.body
    and event_sign = sign_tf.event
    and return_sign = sign_tf.ret
    and enter_frame_sign_for = sign_tf.enter_frame_for
    and enter_sign_for = sign_tf.enter_for
    and enter_sign_ci_for = sign_tf.enter_ci_for
    and sign_tf_abs = sign_tf.tf_abs
    and afilter_sign = sign_tf.backward.afilter
    and feasible_sign = sign_tf.backward.feasible
    and bfilter_sign = sign_tf.backward.bfilter
    and branch_sign = sign_tf.backward.branch
    and bfilter_lifted_sign = sign_tf.backward.bfilter_lifted
    and branch_lifted_sign = sign_tf.backward.branch_lifted
    and bfilter_sign_st_lift = sign_tf.backward.bfilter_st_lift
    and sign_truthy_query = sign_tf.check.truthy_query
    and sign_check_query = sign_tf.check.check_query
    and sign_classify_check = sign_tf.check.classify_check
    and sign_eval_answer = sign_tf.check.eval_answer
    and sign_checks_proven = sign_tf.check.abstract_checks_proven
proof -
  show "mono_nonrelational_ops sign_ops"
  proof (rule mono_nonrelational_opsI, unfold sign_ops_simps sign_refine_ops_simps)
    show "mono_minmax_ops sign_special_ops aval_sign"
      by (rule sign_special.mono_minmax_ops_axioms)
    show "mono_refinement inf aval_sign sign_tobool
            inv_less_sign inv_eq_sign inv_conservative inv_conservative inv_conservative"
      by (rule sign_backward_domain)
    show "sound_check_query sign_less sign_eq gamma_state aval_sign"
```

```
      by (rule sign_check_domain)
  qed
qed (simp add: top_sign_def, intro ext, simp add: special_sign_eq_transfer)
```

No fact is renamed here. The transfer functions get Sign-prefixed names above because they are constants a caller applies, exported to OCaml and named in registration data; the theorems about them stay under `sign_tf.`, which is where a reader looks to find out that they are the generic ones rather than Sign's own. `skip_sign`'s soundness is `sign_tf.skip_sound`, the branch's is `sign_tf.backward.branch_sound`, and the framework's transfer contract at Sign is `sign_tf.is_sound_nonrelational_transfer`.

end

11.7 Sign_Exec

theory Sign_Exec
  imports
    Sign_Transfer
begin

Sign on the executable carrier

The executable step and procedure entry are `Voblint_Nonrelational.Nonrelational_Ops`'s generic constructions at the bundle, `generic_tf_st_for sign_ops` and `generic_enter_st_for sign_ops`, and their agreement with the abstract transfer is `sign_tf.tf_st_for_commute` and `sign_tf.enter_st_for_commute`: nothing about them is the domain's to state.

The state a run starts in: a declared global holds `SZero`, a local `STop`.

abbreviation cinit_sign_st :: "sign default_st" **where**
  "cinit_sign_st ≡ initial_default_st STop SZero"

end

11.8 Sign_Sound

theory Sign_Sound
  imports
    Sign_Exec
begin

What the initial abstract state describes

Sign's specification and its soundness are not restated here: `sign_tf.spec_exec` is sound against `λd g. γ (ϱ_𝒢 d)` by `sign_tf.sound_exec` and `sign_tf.entry_cover_exec`, which every non-relational bundle gets from `sound_nonrelational_ops`.

The initial-state contract, owned here rather than reproved at each assembly instance: every concrete store a run may start in is described by `cinit_sign_st`. It holds because a declared global starts at zero, which `SZero` describes exactly, and a local starts unconstrained, which `STop` describes trivially — so it is a fact about Sign's initial state and its concretization, and about nothing else.

lemma sign_cinit_gamma:
  "cinit_stores 𝒢 ⊆ default_st_gamma 𝒢 cinit_sign_st"
  by (auto simp: cinit_stores_def gamma_state_def default_st_gamma_initial)

end

11.9 Sign_Classify

theory Sign_Classify
  imports
    Sign_Exec
begin

Sign instance of the generic check-discharge interface

The check classifier is derived from Sign's bundle: the Boolean recursion over `exp` (`Not`, `And`, `Or`), the three-way classification, and the node-indexed bridge to `checks_proven` come from the bundle's queries `sign_less`/`sign_eq` and evaluator, through the `sign_tf` interpretation in `Voblint_Analysis_Sign.Sign_Transfer`. Their facts stay under `sign_tf.check.`; `classify_check`'s two soundness directions are `sign_tf.check.classify_check_proved` and `sign_tf.check.classify_check_refuted`. `Sign_Regression` in the Sign examples exercises them.

end

11.10 Sign_Analyses

```
theory Sign_Analyses
  imports
    Sign_Sound
    Sign_Classify
    "Voblint_Nonrelational.Nonrel_Registration"
begin
```

Registering Sign at every context and update rule

Sign runs through the shared D/G pipeline at the unit context. The CLI runs it as a field of the combined state of `MCP_Analyses`, whose component and soundness the unit registration supplies. Each registration leaves the rule that merges a value side-effected into a global as a parameter `r`. The equation system, the solve, the result table and every soundness endpoint come from the interpreted locale; this theory only names the domain's own implementation and facts.

At the unit context

```
global_interpretation sign_rule:
    nonrel_unit_rule sign_ops cinit_sign_st r
  for r
  by (intro nonrel_unit_ruleI nonrel_registrationI
      sign_tf.sound_nonrelational_ops_axioms sign_cinit_gamma)

end
```

Chapter 12

Voblint_Analysis_Interval

12.1 Interval_Warrowing

```
theory Interval_Warrowing
  imports "Voblint_Domain.Interval_Lattice"
begin
```

Interval widening and narrowing

The interval lattice has infinite ascending chains, so the solver needs widening to terminate and narrowing to recover precision afterwards. This theory defines both and instantiates intervals as a numeric domain.

Widening

Standard interval widening: keep a bound if it did not move outward, otherwise push it to the infinite end. Every widen-ascending chain therefore stabilises after at most two steps (each bound takes at most one outward jump).

```
fun widen_ivl_core :: "ivl => ivl => ivl" where
    "widen_ivl_core (Ivl l1 u1) (Ivl l2 u2) =
       Ivl (if l1 ≤ l2 then l1 else MinInf)
           (if u2 ≤ u1 then u1 else PlusInf)"

lemma a_le_widen_ivl_core: "(a :: ivl) ≤ widen_ivl_core a b"
proof (cases a; cases b)
  fix l1 u1 l2 u2 :: eint
  assume "a = Ivl l1 u1" "b = Ivl l2 u2"
  then show "a ≤ widen_ivl_core a b"
    unfolding less_eq_ivl_def by auto
qed

lemma b_le_widen_ivl_core: "(b :: ivl) ≤ widen_ivl_core a b"
proof (cases a; cases b)
  fix l1 u1 l2 u2 :: eint
  assume "a = Ivl l1 u1" "b = Ivl l2 u2"
  then show "b ≤ widen_ivl_core a b"
    unfolding less_eq_ivl_def
    using eint_le_linear[of l1 l2] eint_le_linear[of u2 u1]
    by auto
qed
```

Widening termination: every widen-ascending chain stabilises. At each step each bound either stays or jumps to `MinInf` / `PlusInf`, and once a bound reaches the infinite end it remains there. After at most two transitions the chain is constant.

```
lemma widen_ivl_core_terminates:
  assumes "∀i. widen_ivl_core (f i) (f (Suc i)) = f (Suc i)"
  shows "∃n. ∀j. n ≤ j ⟶ f j = f n"
proof -
  define lb where "lb i = ivl_lower (f i)" for i
  define ub where "ub i = ivl_upper (f i)" for i
  have f_eq: "f i = Ivl (lb i) (ub i)" for i
    unfolding lb_def ub_def by (cases "f i") simp
  have step:
    "lb (Suc i) = lb i ∨ lb (Suc i) = MinInf"
    "ub (Suc i) = ub i ∨ ub (Suc i) = PlusInf"
    for i
  proof -
    have "widen_ivl_core (Ivl (lb i) (ub i)) (Ivl (lb (Suc i)) (ub (Suc i))) =
          Ivl (lb (Suc i)) (ub (Suc i))"
      using assms[rule_format, of i] f_eq[of i] f_eq[of "Suc i"] by simp
    then have eqs:
      "lb (Suc i) = (if eint_le (lb i) (lb (Suc i)) then lb i else MinInf)"
      "ub (Suc i) = (if eint_le (ub (Suc i)) (ub i) then ub i else PlusInf)"
      by simp_all
    from eqs(1) show "lb (Suc i) = lb i ∨ lb (Suc i) = MinInf"
      by (auto split: if_splits)
    from eqs(2) show "ub (Suc i) = ub i ∨ ub (Suc i) = PlusInf"
      by (auto split: if_splits)
  qed
  have lb_stay: "lb i = MinInf ⟹ lb (Suc i) = MinInf" for i
    using step(1) by metis
  have ub_stay: "ub i = PlusInf ⟹ ub (Suc i) = PlusInf" for i
    using step(2) by metis
  have lb_stay_le: "lb i = MinInf ⟹ i ≤ j ⟹ lb j = MinInf" for i j
```

```
  proof (induct j)
    case 0 thus ?case by simp
  next
    case (Suc j) thus ?case using lb_stay le_SucE by metis
  qed
  have ub_stay_le: "ub i = PlusInf ⟹ i ≤ j ⟹ ub j = PlusInf" for i j
  proof (induct j)
    case 0 thus ?case by simp
  next
    case (Suc j) thus ?case using ub_stay le_SucE by metis
  qed
  have lb_const_if_no_MinInf:
    "(∀k. lb k ≠ MinInf) ⟹ lb j = lb 0" for j
  proof (induct j)
    case 0 thus ?case by simp
  next
    case (Suc j) thus ?case using step(1)[of j] by auto
  qed
  have ub_const_if_no_PlusInf:
    "(∀k. ub k ≠ PlusInf) ⟹ ub j = ub 0" for j
  proof (induct j)
    case 0 thus ?case by simp
  next
    case (Suc j) thus ?case using step(2)[of j] by auto
  qed
  obtain nl where nl: "∀j ≥ nl. lb j = lb nl"
  proof (cases "∃k. lb k = MinInf")
    case True
    then obtain k where "lb k = MinInf" by blast
    then have "∀j ≥ k. lb j = MinInf" using lb_stay_le by blast
    then have "∀j ≥ k. lb j = lb k" using ‹lb k = MinInf› by simp
    then show ?thesis by (rule that)
  next
    case False
    then have "∀k. lb k ≠ MinInf" by blast
    then have "∀j. lb j = lb 0" using lb_const_if_no_MinInf by blast
    then have "∀j ≥ 0. lb j = lb 0" by blast
    then show ?thesis by (rule that)
  qed
  obtain nu where nu: "∀j ≥ nu. ub j = ub nu"
  proof (cases "∃k. ub k = PlusInf")
    case True
    then obtain k where "ub k = PlusInf" by blast
    then have "∀j ≥ k. ub j = PlusInf" using ub_stay_le by blast
    then have "∀j ≥ k. ub j = ub k" using ‹ub k = PlusInf› by simp
    then show ?thesis by (rule that)
  next
    case False
    then have "∀k. ub k ≠ PlusInf" by blast
    then have "∀j. ub j = ub 0" using ub_const_if_no_PlusInf by blast
    then have "∀j ≥ 0. ub j = ub 0" by blast
    then show ?thesis by (rule that)
  qed
  let ?n = "max nl nu"
  have "∀j. ?n ≤ j ⟶ f j = f ?n"
  proof (intro allI impI)
    fix j assume jn: "?n ≤ j"
    have lb_eq: "lb j = lb ?n"
    proof -
      from jn have "nl ≤ j" "nl ≤ ?n" by auto
      then have "lb j = lb nl" "lb ?n = lb nl"
        using nl by blast+
      then show ?thesis by simp
    qed
    have ub_eq: "ub j = ub ?n"
    proof -
      from jn have "nu ≤ j" "nu ≤ ?n" by auto
      then have "ub j = ub nu" "ub ?n = ub nu"
        using nu by blast+
      then show ?thesis by simp
    qed
    show "f j = f ?n"
      using f_eq[of j] f_eq[of ?n] lb_eq ub_eq by simp
  qed
  then show ?thesis by blast
qed
```

Type-class widening for TD warrowing solver

Standard interval narrowing: when the recomputed value `b` refines the widened value `a` (`b ≤ a`), keep `a`'s finite bounds and only fill an infinite bound of `a` from `b`. A bound moves off an infinite endpoint at most once, so the narrowing descent is finite. Combined with the backward guard filters this recovers loop bounds under *every* update rule (a bounded local counter reads `[0, 20]` under `join`, `per_origin` and `warrow` alike). It does not help a *flow-insensitive global*: there the guard never bounds the value written back to the slot, so `[0, +inf]` is a genuine fixpoint and narrowing has nothing smaller to descend to.

```
fun narrow_ivl_td :: "ivl ⇒ ivl ⇒ ivl" where
  "narrow_ivl_td (Ivl l1 u1) (Ivl l2 u2) =
     Ivl (if l1 = MinInf then l2 else l1) (if u1 = PlusInf then u2 else u1)"

instantiation ivl :: warrowing begin
```

Widening carries the standard bot-law `bot ∇ x = x`, `x ∇ bot = x`: since ⊥ is the empty interval `Ivl PlusInf MinInf`, an unguarded `widen_ivl_core` from bot would jump straight to the top interval, topping any unknown on its first stabilisation. The guard keeps the first contribution exact.

```
  definition "((a :: ivl) ∇ b) =
     (if a = bot then b else if b = bot then a else widen_ivl_core a b)"
  definition "((a :: ivl) Δ b) = narrow_ivl_td a b"
instance proof intro_classes
  fix a b :: ivl
  show "a ≤ (a ∇ b)"
  proof (cases "a = bot")
    case True thus ?thesis by (simp add: widen_ivl_def)
  next
    case False thus ?thesis
      by (cases "b = bot") (simp_all add: widen_ivl_def a_le_widen_ivl_core)
  qed
  show "b ≤ (a ∇ b)"
  proof (cases "a = bot")
    case True thus ?thesis by (simp add: widen_ivl_def)
  next
    case False thus ?thesis
      by (cases "b = bot") (simp_all add: widen_ivl_def b_le_widen_ivl_core)
  qed
  show "b ≤ a ⟹ b ≤ (a Δ b)"
    unfolding narrow_ivl_def
    by (cases a; cases b) (auto simp: less_eq_ivl_def split: if_splits)
  show "b ≤ a ⟹ (a Δ b) ≤ a"
    unfolding narrow_ivl_def
    by (cases a; cases b) (auto simp: less_eq_ivl_def split: if_splits)
qed
end
```

Numeric domain instance

`numeric_domain` extends `executable_domain`, which includes the solver's `warrowing`, so the instance follows the widening and narrowing above.

```
instantiation ivl :: numeric_domain begin
definition gamma_abs_ivl [simp]: "numeric_domain_class.gamma (a :: ivl) = gamma_ivl a"
definition is_empty_ivl [simp]: "is_empty (a :: ivl) = is_bottom_ivl a"
definition to_string_ivl [simp]:
  "to_string (a :: ivl) = (if is_top_ivl a then sym_top else string_of_ivl a)"
instance proof intro_classes
  show "numeric_domain_class.gamma (bot :: ivl) = {}"
    by (simp add: gamma_ivl_bot)
next
  show "numeric_domain_class.gamma (top :: ivl) = UNIV"
    by (simp add: gamma_ivl_top top_ivl_def)
next
  fix a b :: ivl
  assume H: "a ≤ b"
  show "numeric_domain_class.gamma a ⊆ numeric_domain_class.gamma b"
  proof -
    have "gamma_ivl a ⊆ gamma_ivl b" using H by (rule gamma_ivl_mono)
    then show ?thesis by simp
  qed
next
  fix a :: ivl
  show "is_empty a ⟷ numeric_domain_class.gamma a = {}"
    by (simp add: is_bottom_ivl_correct)
```

qed
end

end

12.2 Interval_Arithmetic

```
theory Interval_Arithmetic
  imports Interval_Warrowing
begin
```

Adding, subtracting and multiplying whole ranges

Addition and subtraction of intervals move the two endpoints independently: the sum of `[a, b]` and `[c, d]` runs from `a + c` to `b + d`. Multiplication cannot, because a negative factor swaps which endpoint is larger, so `ivl_times_core` multiplies all four corner pairs and keeps their minimum and maximum. Every operation first `normalize_ivl`s its arguments — an interval whose lower bound exceeds its upper one denotes no number at all and collapses to `bot` — which is what keeps an unreachable operand from producing a spurious range.

Each operation is proved sound (concrete values drawn from the operands land in the computed interval) and monotone (widening either operand widens the result). Those two facts are what `Interval_Backward` later needs to interpret the shared `sound_arith_ops` locale at Interval.

```
instantiation ivl :: plus begin
fun plus_ivl :: "ivl => ivl => ivl" where
    "plus_ivl (Ivl l1 u1) (Ivl l2 u2) =
       (case (normalize_ivl (Ivl l1 u1), normalize_ivl (Ivl l2 u2)) of
          (Ivl a b, Ivl c d) ⇒ normalize_ivl (Ivl (a + c) (b + d)))"
instance ..
end

instantiation ivl :: minus begin
fun minus_ivl :: "ivl => ivl => ivl" where
    "minus_ivl (Ivl l1 u1) (Ivl l2 u2) =
       (case (normalize_ivl (Ivl l1 u1), normalize_ivl (Ivl l2 u2)) of
          (Ivl a b, Ivl c d) ⇒ normalize_ivl (Ivl (a - d) (b - c)))"
instance ..
end
```

`ivl_min`/`ivl_max` abstract the two-argument `Min`/`Max` special calls: componentwise `min`/`max` of the bounds, exact whenever both operands are non-empty (`min`/`max` of two non-empty ranges is again non-empty, unlike multiplication's corner case), normalized like `plus_ivl`/`minus_ivl` so an empty operand still collapses to the canonical $\bot$.

```
fun ivl_min :: "ivl => ivl => ivl" where
    "ivl_min (Ivl l1 u1) (Ivl l2 u2) = normalize_ivl (Ivl (min l1 l2) (min u1 u2))"

fun ivl_max :: "ivl => ivl => ivl" where
    "ivl_max (Ivl l1 u1) (Ivl l2 u2) = normalize_ivl (Ivl (max l1 l2) (max u1 u2))"

lemma ivl_min_sound:
  assumes "i ∈ gamma_ivl a" "j ∈ gamma_ivl b"
  shows "min i j ∈ gamma_ivl (ivl_min a b)"
  using assms
  by (cases a; cases b)
     (auto simp: normalize_ivl_gamma Fin_min intro: eint_min_mono)

lemma ivl_max_sound:
  assumes "i ∈ gamma_ivl a" "j ∈ gamma_ivl b"
  shows "max i j ∈ gamma_ivl (ivl_max a b)"
  using assms
  by (cases a; cases b)
     (auto simp: normalize_ivl_gamma Fin_max intro: eint_max_mono)

lemma ivl_min_mono1: "a1 ≤ a2 ⟹ ivl_min a1 b ≤ ivl_min a2 (b::ivl)"
proof -
  assume le: "a1 ≤ a2"
  obtain x1 x2 where a1: "a1 = Ivl x1 x2" by (rule ivl_exhaustE)
  obtain x1a x2a where a2: "a2 = Ivl x1a x2a" by (rule ivl_exhaustE)
  obtain x1b x2b where b: "b = Ivl x1b x2b" by (rule ivl_exhaustE)
  from le a1 a2 have h: "eint_le x1a x1" "eint_le x2 x2a"
    by (simp_all add: less_eq_ivl_def)
  have "Ivl (min x1 x1b) (min x2 x2b) ≤ Ivl (min x1a x1b) (min x2a x2b)"
    unfolding less_eq_ivl_def using h by (auto intro: eint_min_mono)
  then show ?thesis unfolding a1 a2 b ivl_min.simps by (rule normalize_ivl_mono)
qed

lemma ivl_min_mono2: "b1 ≤ b2 ⟹ ivl_min a b1 ≤ ivl_min a (b2::ivl)"
proof -
  assume le: "b1 ≤ b2"
```

```
  obtain x1 x2 where a: "a = Ivl x1 x2" by (rule ivl_exhaustE)
  obtain x1a x2a where b1: "b1 = Ivl x1a x2a" by (rule ivl_exhaustE)
  obtain x1b x2b where b2: "b2 = Ivl x1b x2b" by (rule ivl_exhaustE)
  from le b1 b2 have h: "eint_le x1b x1a" "eint_le x2a x2b"
    by (simp_all add: less_eq_ivl_def)
  have "Ivl (min x1 x1a) (min x2 x2a) ≤ Ivl (min x1 x1b) (min x2 x2b)"
    unfolding less_eq_ivl_def using h by (auto intro: eint_min_mono)
  then show ?thesis unfolding a b1 b2 ivl_min.simps by (rule normalize_ivl_mono)
qed

lemma ivl_min_combine_mono:
  "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ ivl_min a1 b1 ≤ ivl_min a2 (b2::ivl)"
  by (meson order.trans ivl_min_mono1 ivl_min_mono2)

lemma ivl_max_mono1: "a1 ≤ a2 ⟹ ivl_max a1 b ≤ ivl_max a2 (b::ivl)"
proof -
  assume le: "a1 ≤ a2"
  obtain x1 x2 where a1: "a1 = Ivl x1 x2" by (rule ivl_exhaustE)
  obtain x1a x2a where a2: "a2 = Ivl x1a x2a" by (rule ivl_exhaustE)
  obtain x1b x2b where b: "b = Ivl x1b x2b" by (rule ivl_exhaustE)
  from le a1 a2 have h: "eint_le x1a x1" "eint_le x2 x2a"
    by (simp_all add: less_eq_ivl_def)
  have "Ivl (max x1 x1b) (max x2 x2b) ≤ Ivl (max x1a x1b) (max x2a x2b)"
    unfolding less_eq_ivl_def using h by (auto intro: eint_max_mono)
  then show ?thesis unfolding a1 a2 b ivl_max.simps by (rule normalize_ivl_mono)
qed

lemma ivl_max_mono2: "b1 ≤ b2 ⟹ ivl_max a b1 ≤ ivl_max a (b2::ivl)"
proof -
  assume le: "b1 ≤ b2"
  obtain x1 x2 where a: "a = Ivl x1 x2" by (rule ivl_exhaustE)
  obtain x1a x2a where b1: "b1 = Ivl x1a x2a" by (rule ivl_exhaustE)
  obtain x1b x2b where b2: "b2 = Ivl x1b x2b" by (rule ivl_exhaustE)
  from le b1 b2 have h: "eint_le x1b x1a" "eint_le x2a x2b"
    by (simp_all add: less_eq_ivl_def)
  have "Ivl (max x1 x1a) (max x2 x2a) ≤ Ivl (max x1 x1b) (max x2 x2b)"
    unfolding less_eq_ivl_def using h by (auto intro: eint_max_mono)
  then show ?thesis unfolding a b1 b2 ivl_max.simps by (rule normalize_ivl_mono)
qed

lemma ivl_max_combine_mono:
  "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ ivl_max a1 b1 ≤ ivl_max a2 (b2::ivl)"
  by (meson order.trans ivl_max_mono1 ivl_max_mono2)

lemma int_mult_in_corners_lo:
  fixes l1 u1 l2 u2 i j :: int
  assumes "l1 ≤ i" "i ≤ u1" "l2 ≤ j" "j ≤ u2"
  shows "min (l1*l2) (min (l1*u2) (min (u1*l2) (u1*u2))) ≤ i*j"
proof -
  consider (A) "0 ≤ i" "0 ≤ l2"
    | (B) "0 ≤ i" "l2 < 0"
    | (C) "i < 0" "0 ≤ u2"
    | (D) "i < 0" "u2 < 0"
    by linarith
  then show ?thesis
  proof cases
    case A
    have "l1*l2 ≤ i*l2"
      using assms(1) A(2)
      by (simp add: mult_right_mono)
    also have "... ≤ i*j"
      by (simp add: A(1) assms(3) mult_left_mono)
    finally show ?thesis by simp
  next
    case B
    have "u1*l2 ≤ i*l2"
      using assms(2) B(2) by simp
    also have "... ≤ i*j"
      using assms(3) B(1) by (rule mult_left_mono)
    finally show ?thesis by simp
  next
    case C
    have "l1*u2 ≤ i*u2"
      using assms(1) C(2) by (rule mult_right_mono)
    also have "... ≤ i*j"
      using assms(4) C(1) by simp
```

```
    finally show ?thesis by simp
  next
    case D
    have "u1*u2 ≤ i*u2"
      using assms(2) D(2) by simp
    also have "... ≤ i*j"
      using assms(4) D(1) by simp
    finally show ?thesis by simp
  qed
qed

lemma int_mult_in_corners_hi:
  fixes l1 u1 l2 u2 i j :: int
  assumes "l1 ≤ i" "i ≤ u1" "l2 ≤ j" "j ≤ u2"
  shows "i * j ≤ max (l1*l2) (max (l1*u2) (max (u1*l2) (u1*u2)))"
proof -
  have negated:
    "min ((-u1)*l2) (min ((-u1)*u2) (min ((-l1)*l2) ((-l1)*u2)))
       ≤ (-i)*j"
    by (rule int_mult_in_corners_lo) (use assms in simp_all)
  from negated show ?thesis
    by (simp add: min_def max_def; linarith)
qed
```

Precise multiplication on non-empty intervals. For all-finite operands the product range is the min / max over the four corner products (the box product attains its extrema at the corners); any infinite bound falls back to `ivl_top`. An empty operand yields $\bot$ – the proper bottom handling that keeps the operator monotone (mapping empties to `ivl_top` would destroy the ordering).

```
fun ivl_times_core :: "ivl => ivl => ivl" where
    "ivl_times_core (Ivl (Fin l1) (Fin u1)) (Ivl (Fin l2) (Fin u2)) =
       Ivl (Fin (min (l1*l2) (min (l1*u2) (min (u1*l2) (u1*u2)))))
           (Fin (max (l1*l2) (max (l1*u2) (max (u1*l2) (u1*u2)))))"
  | "ivl_times_core _ _ = ivl_top"

instantiation ivl :: times begin
definition times_ivl :: "ivl => ivl => ivl" where
  "times_ivl a b = (if ivl_nonempty a ∧ ivl_nonempty b then ivl_times_core a b else bot)"
instance ..
end
```

`times_ivl_def` is deliberately not `[simp]`: its conditional body would make `simp` split on `ivl_nonempty` before `ivl_times_sound` or `ivl_times_mono` can fire. A proof that must unfold `*` directly adds it explicitly.

```
lemma ivl_plus_sound:
  assumes "i ∈ gamma_ivl a" "j ∈ gamma_ivl b"
  shows "i + j ∈ gamma_ivl (a + b)"
proof (cases a; cases b)
  fix l1 u1 l2 u2 :: eint
  assume "a = Ivl l1 u1" "b = Ivl l2 u2"
  with assms have bnds:
    "eint_le l1 (Fin i)" "eint_le (Fin i) u1"
    "eint_le l2 (Fin j)" "eint_le (Fin j) u2"
    by auto
  show "i + j ∈ gamma_ivl (a + b)"
    unfolding ‹a = Ivl l1 u1› ‹b = Ivl l2 u2›
    using bnds normalize_ivl_def
    by (cases l1; cases l2; cases u1; cases u2) auto
qed

lemma ivl_minus_sound:
  assumes "i ∈ gamma_ivl a" "j ∈ gamma_ivl b"
  shows "i - j ∈ gamma_ivl (a - b)"
proof (cases a; cases b)
  fix l1 u1 l2 u2 :: eint
  assume "a = Ivl l1 u1" "b = Ivl l2 u2"
  with assms have bnds:
    "eint_le l1 (Fin i)" "eint_le (Fin i) u1"
    "eint_le l2 (Fin j)" "eint_le (Fin j) u2"
    by auto
  show "i - j ∈ gamma_ivl (a - b)"
    unfolding ‹a = Ivl l1 u1› ‹b = Ivl l2 u2›
    using bnds normalize_ivl_def
    by (cases l1; cases l2; cases u1; cases u2) auto
qed
```

```
lemma ivl_times_sound:
  assumes "i ∈ gamma_ivl a" "j ∈ gamma_ivl b"
  shows "i * j ∈ gamma_ivl (a * b)"
proof -
  from assms have ne: "ivl_nonempty a" "ivl_nonempty b"
    by (auto intro: gamma_ivl_nonempty)
  hence eq: "a * b = ivl_times_core a b" by (simp add: times_ivl_def)
  show ?thesis
  proof (cases a; cases b)
    fix l1 u1 l2 u2 assume ab: "a = Ivl l1 u1" "b = Ivl l2 u2"
    show ?thesis
    proof (cases l1; cases u1; cases l2; cases u2)
      fix n1 m1 n2 m2 assume fin: "l1 = Fin n1" "u1 = Fin m1" "l2 = Fin n2" "u2 = Fin m2"
      from assms ab fin have bnds: "n1 ≤ i" "i ≤ m1" "n2 ≤ j" "j ≤ m2" by auto
      show ?thesis using eq ab fin
        int_mult_in_corners_lo[OF bnds] int_mult_in_corners_hi[OF bnds] by simp
    qed (use eq ab ne(1) ne(2) in ‹simp_all add: ivl_top_def›)
  qed
qed
```

Both operators route their operands through `normalize_ivl`; these rewrites expose that shape without the constructor-pattern gate, so monotonicity factors through `normalize_ivl`'s monotonicity and the pointwise `eint` bounds.

```
lemma plus_ivl_norm:
  "a + b = (case (normalize_ivl a, normalize_ivl b) of
              (Ivl l1 u1, Ivl l2 u2) ⇒ normalize_ivl (Ivl (l1 + l2) (u1 + u2)))"
  by (cases a; cases b) simp

lemma minus_ivl_norm:
  "a - b = (case (normalize_ivl a, normalize_ivl b) of
              (Ivl l1 u1, Ivl l2 u2) ⇒ normalize_ivl (Ivl (l1 - u2) (u1 - l2)))"
  by (cases a; cases b) simp

lemma ivl_plus_mono:
  assumes "a1 ≤ a2" "b1 ≤ b2" shows "a1 + b1 ≤ a2 + (b2::ivl)"
proof -
  obtain pa qa where na1: "normalize_ivl a1 = Ivl pa qa" by (rule ivl_exhaustE)
  obtain ra sa where na2: "normalize_ivl a2 = Ivl ra sa" by (rule ivl_exhaustE)
  obtain pb qb where nb1: "normalize_ivl b1 = Ivl pb qb" by (rule ivl_exhaustE)
  obtain rb sb where nb2: "normalize_ivl b2 = Ivl rb sb" by (rule ivl_exhaustE)
  from normalize_ivl_mono[OF assms(1)] na1 na2 have A: "Ivl pa qa ≤ Ivl ra sa" by simp
  from normalize_ivl_mono[OF assms(2)] nb1 nb2 have B: "Ivl pb qb ≤ Ivl rb sb" by simp
  have "Ivl (pa + pb) (qa + qb) ≤ Ivl (ra + rb) (sa + sb)"
    using A B by (auto simp: less_eq_ivl_def eint_plus_mono)
  moreover have "a1 + b1 = normalize_ivl (Ivl (pa + pb) (qa + qb))"
    by (simp add: plus_ivl_norm na1 nb1)
  moreover have "a2 + b2 = normalize_ivl (Ivl (ra + rb) (sa + sb))"
    by (simp add: plus_ivl_norm na2 nb2)
  ultimately show ?thesis by (metis normalize_ivl_mono)
qed

lemma ivl_minus_mono:
  assumes "a1 ≤ a2" "b1 ≤ b2" shows "a1 - b1 ≤ a2 - (b2::ivl)"
proof -
  obtain pa qa where na1: "normalize_ivl a1 = Ivl pa qa" by (rule ivl_exhaustE)
  obtain ra sa where na2: "normalize_ivl a2 = Ivl ra sa" by (rule ivl_exhaustE)
  obtain pb qb where nb1: "normalize_ivl b1 = Ivl pb qb" by (rule ivl_exhaustE)
  obtain rb sb where nb2: "normalize_ivl b2 = Ivl rb sb" by (rule ivl_exhaustE)
  from normalize_ivl_mono[OF assms(1)] na1 na2 have A: "Ivl pa qa ≤ Ivl ra sa" by simp
  from normalize_ivl_mono[OF assms(2)] nb1 nb2 have B: "Ivl pb qb ≤ Ivl rb sb" by simp
  have "Ivl (pa - qb) (qa - pb) ≤ Ivl (ra - sb) (sa - rb)"
    using A B by (auto simp: less_eq_ivl_def eint_minus_mono)
  moreover have "a1 - b1 = normalize_ivl (Ivl (pa - qb) (qa - pb))"
    by (simp add: minus_ivl_norm na1 nb1)
  moreover have "a2 - b2 = normalize_ivl (Ivl (ra - sb) (sa - rb))"
    by (simp add: minus_ivl_norm na2 nb2)
  ultimately show ?thesis by (metis normalize_ivl_mono)
qed
```

Monotonicity of the precise corner product. Over a larger box the corner minimum can only drop and the corner maximum can only rise (each old corner is a point of the new box), so widening either operand widens the product.

```
lemma corner_min_mono:
  fixes a1 b1 c1 d1 a2 b2 c2 d2 :: int
```

```
  assumes "a2 ≤ a1" "b1 ≤ b2" "c2 ≤ c1" "d1 ≤ d2" "a1 ≤ b1" "c1 ≤ d1"
  shows "min (a2*c2) (min (a2*d2) (min (b2*c2) (b2*d2)))
       ≤ min (a1*c1) (min (a1*d1) (min (b1*c1) (b1*d1)))"
proof -
  let ?M = "min (a2*c2) (min (a2*d2) (min (b2*c2) (b2*d2)))"
  have d1: "a1 ≤ b2" "c1 ≤ d2" "a2 ≤ b1" "c2 ≤ d1" using assms by auto
  have "?M ≤ a1*c1" by (rule int_mult_in_corners_lo[OF assms(1) d1(1) assms(3) d1(2)])
  moreover have "?M ≤ a1*d1" by (rule int_mult_in_corners_lo[OF assms(1) d1(1) d1(4) assms(4)])
  moreover have "?M ≤ b1*c1" by (rule int_mult_in_corners_lo[OF d1(3) assms(2) assms(3) d1(2)])
  moreover have "?M ≤ b1*d1" by (rule int_mult_in_corners_lo[OF d1(3) assms(2) d1(4) assms(4)])
  ultimately show ?thesis by simp
qed

lemma corner_max_mono:
  fixes a1 b1 c1 d1 a2 b2 c2 d2 :: int
  assumes "a2 ≤ a1" "b1 ≤ b2" "c2 ≤ c1" "d1 ≤ d2" "a1 ≤ b1" "c1 ≤ d1"
  shows "max (a1*c1) (max (a1*d1) (max (b1*c1) (b1*d1)))
       ≤ max (a2*c2) (max (a2*d2) (max (b2*c2) (b2*d2)))"
proof -
  let ?M = "max (a2*c2) (max (a2*d2) (max (b2*c2) (b2*d2)))"
  have d1: "a1 ≤ b2" "c1 ≤ d2" "a2 ≤ b1" "c2 ≤ d1" using assms by auto
  have "a1*c1 ≤ ?M" by (rule int_mult_in_corners_hi[OF assms(1) d1(1) assms(3) d1(2)])
  moreover have "a1*d1 ≤ ?M" by (rule int_mult_in_corners_hi[OF assms(1) d1(1) d1(4) assms(4)])
  moreover have "b1*c1 ≤ ?M" by (rule int_mult_in_corners_hi[OF d1(3) assms(2) assms(3) d1(2)])
  moreover have "b1*d1 ≤ ?M" by (rule int_mult_in_corners_hi[OF d1(3) assms(2) d1(4) assms(4)])
  ultimately show ?thesis by simp
qed

lemma ivl_times_core_top:
  "¬ (∃la ua lb ub. a = Ivl (Fin la) (Fin ua) ∧ b = Ivl (Fin lb) (Fin ub))
   ⟹ ivl_times_core a b = ivl_top"
  by (cases "(a,b)" rule: ivl_times_core.cases) auto

lemma ivl_nonempty_le_fin:
  assumes "ivl_nonempty a" "a ≤ Ivl (Fin l2) (Fin u2)"
  shows "∃l1 u1. a = Ivl (Fin l1) (Fin u1)"
proof (cases a)
  case (Ivl l u)
  with assms show ?thesis by (cases l; cases u; auto simp: less_eq_ivl_def)
qed

lemma ivl_times_core_mono:
  assumes ne: "ivl_nonempty a1" "ivl_nonempty b1"
      and le: "a1 ≤ a2" "b1 ≤ b2"
  shows "ivl_times_core a1 b1 ≤ ivl_times_core a2 b2"
proof (cases "∃la ua lb ub. a2 = Ivl (Fin la) (Fin ua) ∧ b2 = Ivl (Fin lb) (Fin ub)")
  case False
  then have "ivl_times_core a2 b2 = ivl_top" by (rule ivl_times_core_top)
  thus ?thesis by (simp add: ivl_le_top)
next
  case True
  then obtain la2 ua2 lb2 ub2 where
    a2: "a2 = Ivl (Fin la2) (Fin ua2)" and b2: "b2 = Ivl (Fin lb2) (Fin ub2)" by blast
  from ivl_nonempty_le_fin[OF ne(1)] le(1) a2 obtain la1 ua1 where
    a1: "a1 = Ivl (Fin la1) (Fin ua1)" by metis
  from ivl_nonempty_le_fin[OF ne(2)] le(2) b2 obtain lb1 ub1 where
    b1: "b1 = Ivl (Fin lb1) (Fin ub1)" by metis
  from le a1 a2 b1 b2 have ord:
    "la2 ≤ la1" "ua1 ≤ ua2" "lb2 ≤ lb1" "ub1 ≤ ub2"
    by (auto simp: less_eq_ivl_def)
  from ne a1 b1 have nemp: "la1 ≤ ua1" "lb1 ≤ ub1" by auto
  show ?thesis
    unfolding a1 a2 b1 b2 ivl_times_core.simps less_eq_ivl_def
    using corner_min_mono[OF ord(1) ord(2) ord(3) ord(4) nemp(1) nemp(2)]
          corner_max_mono[OF ord(1) ord(2) ord(3) ord(4) nemp(1) nemp(2)]
    by simp
qed

lemma ivl_times_mono:
  assumes "a1 ≤ a2" "b1 ≤ b2"
  shows "a1 * b1 ≤ a2 * (b2::ivl)"
proof (cases "ivl_nonempty a1 ∧ ivl_nonempty b1")
  case False
  then have "a1 * b1 = bot" by (auto simp add: times_ivl_def)
  thus ?thesis by simp
next
```

```
  case True
  hence ne2: "ivl_nonempty a2" "ivl_nonempty b2"
    using assms ivl_nonempty_mono by blast+
  have "a1 * b1 = ivl_times_core a1 b1"
    using True by (simp add: times_ivl_def)
  moreover have "a2 * b2 = ivl_times_core a2 b2"
    using ne2 by (simp add: times_ivl_def)
  ultimately show ?thesis
    using ivl_times_core_mono[OF _ _ assms] True by simp
qed
```

Truncating division

C division truncates toward zero, so `c_div` and `c_mod` differ from HOL's flooring `div`/`mod` on negative operands. These lemmas bound `c_mod` by both operands and show `c_div` monotone in the dividend, the facts interval division rests on.

```
lemma c_mod_modulus_bound:
  assumes "b ≠ 0"
  shows "abs (c_mod a b) < abs b"
proof -
  have "0 ≤ abs a mod abs b" "abs a mod abs b < abs b"
    using assms by (auto intro: pos_mod_sign pos_mod_bound)
  then show ?thesis unfolding c_mod_abs
    by (auto simp: sgn_if abs_mult)
qed


lemma c_mod_dividend_bound:
  "abs (c_mod a b) ≤ abs a"
  unfolding c_mod_abs
  using zmod_le_nonneg_dividend[of "abs a" "abs b"]
  by (cases "b = 0") (auto simp: sgn_if abs_mult intro: pos_mod_sign)


lemma c_div_nonneg_divisor:
  assumes "0 < b"
  shows "c_div a b = (if a < 0 then - ((-a) div b) else a div b)"
  using assms by (auto simp: c_div_def sgn_if)


lemma c_div_mono_dividend:
  assumes "a1 ≤ a2" "0 < b"
  shows "c_div a1 b ≤ c_div a2 b"
proof (cases "a1 < 0 ∧ 0 ≤ a2")
  case True
  have "c_div a1 b ≤ 0" "0 ≤ c_div a2 b"
    using True assms c_div_nonpos(1) c_div_nonneg(1) by auto
  then show ?thesis by linarith
next
  case False
  show ?thesis unfolding c_div_nonneg_divisor[OF assms(2)]
    using False assms
    by (auto split: if_splits intro: zdiv_mono1)
qed


lemma c_div_antimono_divisor:
  assumes "0 ≤ a" "0 < b1" "b1 ≤ b2"
  shows "c_div a b2 ≤ c_div a b1"
  using assms
  by (simp add: c_div_nonneg_divisor zdiv_mono2)


lemma c_div_positive_divisor_bounds:
  assumes "0 < l" "l ≤ b" "b ≤ u"
  shows "min (c_div a l) (c_div a u) ≤ c_div a b ∧
    c_div a b ≤ max (c_div a l) (c_div a u)"
proof (cases "0 ≤ a")
  case True
  have "c_div a u ≤ c_div a b" "c_div a b ≤ c_div a l"
    using assms True by (auto intro: c_div_antimono_divisor)
  then show ?thesis by auto
next
  case False
  have "c_div (-a) u ≤ c_div (-a) b" "c_div (-a) b ≤ c_div (-a) l"
```

```
    using assms False by (auto intro: c_div_antimono_divisor)
  then show ?thesis
    by (simp_all add: c_div_neg_dividend min_le_iff_disj le_max_iff_disj)
qed


lemma c_div_positive_corners:
  assumes "l ≤ i" "i ≤ u" "0 < c" "c ≤ j" "j ≤ d"
  shows "min (c_div l c) (c_div l d) ≤ c_div i j"
    and "c_div i j ≤ max (c_div u c) (c_div u d)"
proof -
  have "min (c_div l c) (c_div l d) ≤ c_div l j"
    using c_div_positive_divisor_bounds[OF assms(3-5), THEN conjunct1] .
  also have "c_div l j ≤ c_div i j"
    using assms by (auto intro: c_div_mono_dividend)
  finally show "min (c_div l c) (c_div l d) ≤ c_div i j" .
  have "c_div i j ≤ c_div u j"
    using assms by (auto intro: c_div_mono_dividend)
  also have "c_div u j ≤ max (c_div u c) (c_div u d)"
    using c_div_positive_divisor_bounds[OF assms(3-5), THEN conjunct2] .
  finally show "c_div i j ≤ max (c_div u c) (c_div u d)" .
qed


fun ivl_div_positive_core :: "ivl ⇒ ivl ⇒ ivl" where
  "ivl_div_positive_core (Ivl (Fin l) (Fin u)) (Ivl (Fin c) (Fin d)) =
    (if 0 < c then
      Ivl (Fin (min (c_div l c) (c_div l d))) (Fin (max (c_div u c) (c_div u d)))
     else ivl_top)"
| "ivl_div_positive_core a b = (if b = Ivl (Fin 1) (Fin 1) then a else ivl_top)"

definition ivl_div_positive :: "ivl ⇒ ivl ⇒ ivl" where
  "ivl_div_positive a b =
    (if ivl_nonempty a ∧ ivl_nonempty b
     then normalize_ivl (ivl_div_positive_core a b) else bot)"

lemma ivl_div_positive_core_sound:
  assumes "i ∈ gamma_ivl a" "j ∈ gamma_ivl b"
  shows "c_div i j ∈ gamma_ivl (ivl_div_positive_core a b)"
  using assms
  by (cases "(a, b)" rule: ivl_div_positive_core.cases)
     (auto simp: gamma_ivl_top intro: c_div_positive_corners)

lemma ivl_div_positive_sound:
  "i ∈ gamma_ivl a ⟹ j ∈ gamma_ivl b ⟹
    c_div i j ∈ gamma_ivl (ivl_div_positive a b)"
  unfolding ivl_div_positive_def
  by (auto simp: normalize_ivl_gamma
      intro: ivl_div_positive_core_sound dest: gamma_ivl_nonempty)

lemma ivl_div_positive_core_one:
  "ivl_div_positive_core a (Ivl (Fin 1) (Fin 1)) = a"
  by (cases "(a, Ivl (Fin 1) (Fin 1))" rule: ivl_div_positive_core.cases) auto

lemma ivl_div_positive_core_top:
  "¬ (∃l u c d. a = Ivl (Fin l) (Fin u) ∧ b = Ivl (Fin c) (Fin d) ∧ 0 < c)
    ⟹ b ≠ Ivl (Fin 1) (Fin 1) ⟹ ivl_div_positive_core a b = ivl_top"
  by (cases "(a, b)" rule: ivl_div_positive_core.cases) auto

lemma ivl_div_positive_core_mono:
  assumes ne: "ivl_nonempty a1" "ivl_nonempty b1"
    and le: "a1 ≤ a2" "b1 ≤ b2"
  shows "ivl_div_positive_core a1 b1 ≤ ivl_div_positive_core a2 b2"
proof (cases "b2 = Ivl (Fin 1) (Fin 1)")
  case True
  have b1: "b1 = Ivl (Fin 1) (Fin 1)"
    using ivl_nonempty_le_fin[OF ne(2) le(2)[unfolded True]] le(2) ne(2)
    by (auto simp: True less_eq_ivl_def)
  show ?thesis using le(1) by (simp add: True b1 ivl_div_positive_core_one)
next
  case not_one: False
  show ?thesis
  proof (cases "∃l u c d. a2 = Ivl (Fin l) (Fin u) ∧ b2 = Ivl (Fin c) (Fin d) ∧ 0 < c")
    case False
    then show ?thesis using not_one by (simp add: ivl_div_positive_core_top ivl_le_top)
  next
```

```
    case True
    then obtain l2 u2 c2 d2 where
      a2: "a2 = Ivl (Fin l2) (Fin u2)" and b2: "b2 = Ivl (Fin c2) (Fin d2)"
      and pos: "0 < c2" by blast
    obtain l1 u1 where a1: "a1 = Ivl (Fin l1) (Fin u1)"
      using ivl_nonempty_le_fin[OF ne(1) le(1)[unfolded a2]] by blast
    obtain c1 d1 where b1: "b1 = Ivl (Fin c1) (Fin d1)"
      using ivl_nonempty_le_fin[OF ne(2) le(2)[unfolded b2]] by blast
    have bounds: "l2 ≤ l1" "u1 ≤ u2" "c2 ≤ c1" "d1 ≤ d2" "l1 ≤ u1" "c1 ≤ d1"
      using le ne unfolding a1 a2 b1 b2 by (auto simp: less_eq_ivl_def)
    have lo:
      "min (c_div l2 c2) (c_div l2 d2) ≤ c_div l1 c1"
      "min (c_div l2 c2) (c_div l2 d2) ≤ c_div l1 d1"
      using bounds pos by (auto intro: c_div_positive_corners(1))
    have hi:
      "c_div u1 c1 ≤ max (c_div u2 c2) (c_div u2 d2)"
      "c_div u1 d1 ≤ max (c_div u2 c2) (c_div u2 d2)"
      using bounds pos by (auto intro: c_div_positive_corners(2))
    show ?thesis unfolding a1 a2 b1 b2
      using lo hi pos bounds by (simp add: less_eq_ivl_def)
  qed
qed


lemma ivl_div_positive_mono:
  assumes "a1 ≤ a2" "b1 ≤ b2"
  shows "ivl_div_positive a1 b1 ≤ ivl_div_positive a2 b2"
  using assms
  unfolding ivl_div_positive_def
  by (auto intro: normalize_ivl_mono ivl_div_positive_core_mono
      dest: ivl_nonempty_mono)

definition ivl_positive_part :: "ivl ⇒ ivl" where
  "ivl_positive_part a = intersect_ivl a (Ivl (Fin 1) PlusInf)"

definition ivl_zero_part :: "ivl ⇒ ivl" where
  "ivl_zero_part a = intersect_ivl a (Ivl (Fin 0) (Fin 0))"

definition ivl_div :: "ivl ⇒ ivl ⇒ ivl" where
  "ivl_div a b =
    (if ivl_nonempty a ∧ ivl_nonempty b then
      ivl_div_positive a (ivl_positive_part b) ⊔
      ivl_div_positive (Ivl (Fin 0) (Fin 0) - a)
        (ivl_positive_part (Ivl (Fin 0) (Fin 0) - b)) ⊔ ivl_zero_part b
     else bot)"


lemma ivl_div_join_left:
  "x ∈ gamma_ivl a ⟹ x ∈ gamma_ivl (a ⊔ b)"
  using gamma_ivl_mono[OF le_supI1] by blast

lemma ivl_div_join_right:
  "x ∈ gamma_ivl b ⟹ x ∈ gamma_ivl (a ⊔ b)"
  using gamma_ivl_mono[OF le_supI2] by blast

lemma ivl_div_sound:
  assumes "i ∈ gamma_ivl a" "j ∈ gamma_ivl b"
  shows "c_div i j ∈ gamma_ivl (ivl_div a b)"
proof -
  have ne: "ivl_nonempty a" "ivl_nonempty b"
    using assms by (auto dest: gamma_ivl_nonempty)
  show ?thesis
  proof (cases "j = 0")
    case True
    then have "c_div i j ∈ gamma_ivl (ivl_zero_part b)"
      unfolding ivl_zero_part_def using assms True
      by (intro intersect_ivl_gamma) auto
    then show ?thesis unfolding ivl_div_def using ne
      by (auto intro: ivl_div_join_right)
  next
    case nz: False
    show ?thesis
    proof (cases "0 < j")
      case True
      have "j ∈ gamma_ivl (ivl_positive_part b)"
        unfolding ivl_positive_part_def using assms True
```

```
      by (intro intersect_ivl_gamma) auto
    then have "c_div i j ∈ gamma_ivl (ivl_div_positive a (ivl_positive_part b))"
      using assms(1) by (auto intro: ivl_div_positive_sound)
    then show ?thesis unfolding ivl_div_def using ne
      by (auto intro: ivl_div_join_left)
  next
    case False
    have ni: "-i ∈ gamma_ivl (Ivl (Fin 0) (Fin 0) - a)"
      using ivl_minus_sound[of 0 _ i a] assms(1) by simp
    have nj: "-j ∈ gamma_ivl (ivl_positive_part (Ivl (Fin 0) (Fin 0) - b))"
      unfolding ivl_positive_part_def
      apply (rule intersect_ivl_gamma)
       apply (use ivl_minus_sound[of 0 "Ivl (Fin 0) (Fin 0)" j b] assms(2) in simp)
      using False nz by auto
    have "c_div i j ∈ gamma_ivl
      (ivl_div_positive (Ivl (Fin 0) (Fin 0) - a)
        (ivl_positive_part (Ivl (Fin 0) (Fin 0) - b)))"
      using ivl_div_positive_sound[OF ni nj]
      by (simp add: c_div_neg_dividend c_div_neg_divisor)
    then show ?thesis unfolding ivl_div_def using ne
      by (auto intro: ivl_div_join_left
          ivl_div_join_right)
  qed
 qed
qed

lemma ivl_div_mono:
 assumes "a1 ≤ a2" "b1 ≤ b2"
 shows "ivl_div a1 b1 ≤ ivl_div a2 b2"
proof -
 have pos: "ivl_positive_part b1 ≤ ivl_positive_part b2"
   unfolding ivl_positive_part_def
   by (intro intersect_ivl_mono assms order_refl)
 have neg_a: "Ivl (Fin 0) (Fin 0) - a1 ≤ Ivl (Fin 0) (Fin 0) - a2"
   by (intro ivl_minus_mono assms order_refl)
 have neg_b: "ivl_positive_part (Ivl (Fin 0) (Fin 0) - b1) ≤
   ivl_positive_part (Ivl (Fin 0) (Fin 0) - b2)"
   unfolding ivl_positive_part_def
   by (intro intersect_ivl_mono ivl_minus_mono assms order_refl)
 have zero: "ivl_zero_part b1 ≤ ivl_zero_part b2"
   unfolding ivl_zero_part_def
   by (intro intersect_ivl_mono assms order_refl)
 have joined:
   "ivl_div_positive a1 (ivl_positive_part b1) ⊔
     ivl_div_positive (Ivl (Fin 0) (Fin 0) - a1)
       (ivl_positive_part (Ivl (Fin 0) (Fin 0) - b1)) ⊔ ivl_zero_part b1 ≤
    ivl_div_positive a2 (ivl_positive_part b2) ⊔
     ivl_div_positive (Ivl (Fin 0) (Fin 0) - a2)
       (ivl_positive_part (Ivl (Fin 0) (Fin 0) - b2)) ⊔ ivl_zero_part b2"
   by (intro sup_mono ivl_div_positive_mono assms pos neg_a neg_b zero)
 show ?thesis unfolding ivl_div_def
   using joined ivl_nonempty_mono assms by auto
qed
```

Remainder bounds

`ivl_mod_bound` bounds a remainder by the divisor alone. When the divisor interval excludes zero, `abs (c_mod i j) < abs j` gives a symmetric interval one inside the largest divisor magnitude; otherwise the bound is `ivl_top`.

```
fun ivl_mod_bound :: "ivl ⇒ ivl" where
 "ivl_mod_bound (Ivl (Fin l) (Fin u)) =
   (if 0 < l ∨ u < 0 then
     Ivl (Fin (1 - max (abs l) (abs u))) (Fin (max (abs l) (abs u) - 1))
    else ivl_top)"
| "ivl_mod_bound _ = ivl_top"

lemma abs_between_bounds:
 fixes l i u :: int
 assumes "l ≤ i" "i ≤ u"
 shows "abs i ≤ max (abs l) (abs u)"
 using assms by (auto simp: abs_if)

lemma ivl_mod_bound_sound:
 assumes "j ∈ gamma_ivl b"
 shows "c_mod i j ∈ gamma_ivl (ivl_mod_bound b)"
proof (cases b rule: ivl_mod_bound.cases)
```

```
  case (1 l u)
  with assms have bounds: "l ≤ j" "j ≤ u" by simp_all
  have magnitude: "abs j ≤ max (abs l) (abs u)"
    by (rule abs_between_bounds[OF bounds])
  show ?thesis unfolding 1 ivl_mod_bound.simps
    using bounds magnitude c_mod_modulus_bound[of j i]
    by (auto simp: gamma_ivl_top; linarith)
qed (simp_all add: gamma_ivl_top)

lemma ivl_mod_bound_mono:
  assumes "ivl_nonempty b1" "b1 ≤ b2"
  shows "ivl_mod_bound b1 ≤ ivl_mod_bound b2"
proof (cases b2 rule: ivl_mod_bound.cases)
  case (1 l2 u2)
  obtain l1 u1 where b1: "b1 = Ivl (Fin l1) (Fin u1)"
    using ivl_nonempty_le_fin[OF assms(1) assms(2)[unfolded 1]] by blast
  have bounds: "l2 ≤ l1" "u1 ≤ u2" "l1 ≤ u1"
    using assms unfolding 1 b1 by (auto simp: less_eq_ivl_def)
  have "max (abs l1) (abs u1) ≤ max (abs l2) (abs u2)"
    using abs_between_bounds[of l2 l1 u2] abs_between_bounds[of l2 u1 u2] bounds by auto
  then show ?thesis unfolding 1 b1 ivl_mod_bound.simps
    using bounds by (auto simp: less_eq_ivl_def ivl_top_def)
qed (simp_all add: ivl_le_top)

lemma ivl_mod_dividend_sound:
  assumes "i ∈ gamma_ivl a"
  shows "c_mod i j ∈ gamma_ivl (a ⊔ Ivl (Fin 0) (Fin 0))"
proof -
  have range: "min 0 i ≤ c_mod i j ∧ c_mod i j ≤ max 0 i"
    using c_mod_dividend_bound[of i j] c_mod_nonneg[of i j] c_mod_nonpos[of i j]
    by (auto simp: abs_if min_def max_def split: if_splits; linarith)
  obtain l u where a: "a = Ivl l u" by (cases a)
  show ?thesis using assms range
    unfolding a sup_ivl_def
    by (cases l; cases u; auto simp: min_def max_def split: if_splits)
qed

definition ivl_mod :: "ivl ⇒ ivl ⇒ ivl" where
  "ivl_mod a b =
    (if ivl_nonempty a ∧ ivl_nonempty b then
      intersect_ivl (a - ivl_div a b * b)
        (intersect_ivl (a ⊔ Ivl (Fin 0) (Fin 0)) (ivl_mod_bound b))
     else bot)"

lemma ivl_mod_sound:
  assumes "i ∈ gamma_ivl a" "j ∈ gamma_ivl b"
  shows "c_mod i j ∈ gamma_ivl (ivl_mod a b)"
proof -
  have ne: "ivl_nonempty a" "ivl_nonempty b"
    using assms by (auto dest: gamma_ivl_nonempty)
  have residual: "i - c_div i j * j ∈ gamma_ivl (a - ivl_div a b * b)"
    by (intro ivl_minus_sound ivl_times_sound ivl_div_sound assms)
  have "c_mod i j ∈ gamma_ivl (a - ivl_div a b * b)"
    using residual by (cases "j = 0") (simp_all add: c_mod_def)
  then show ?thesis unfolding ivl_mod_def
    using assms(1,2) intersect_ivl_gamma ivl_mod_bound_sound ivl_mod_dividend_sound ne(1,2)
    by presburger
qed

lemma ivl_mod_mono:
  assumes "a1 ≤ a2" "b1 ≤ b2"
  shows "ivl_mod a1 b1 ≤ ivl_mod a2 b2"
proof (cases "ivl_nonempty a1 ∧ ivl_nonempty b1")
  case False
  then show ?thesis unfolding ivl_mod_def by (simp only: False if_False bot_least)
next
  case True
  have ne: "ivl_nonempty a2" "ivl_nonempty b2"
    using True assms ivl_nonempty_mono by blast+
  show ?thesis unfolding ivl_mod_def
    apply (simp only: True ne simp_thms if_True)
    by (intro intersect_ivl_mono ivl_minus_mono ivl_times_mono ivl_div_mono
        sup_mono ivl_mod_bound_mono assms order_refl) (use True in auto)
qed
```

Results are canonical

Every public interval operation returns a canonical representative: a non-empty interval, or $\bot$ itself. No operation manufactures a fresh inverted bound pair such as `Ivl (Fin 2) (Fin 1)`, so structural equality on results distinguishes exactly the abstract values they denote.

```
lemma normalize_ivl_plus [simp]: "normalize_ivl (a + b) = a + b"
  by (simp add: plus_ivl_norm split: ivl.splits prod.splits)

lemma normalize_ivl_minus [simp]: "normalize_ivl (a - b) = a - b"
  by (simp add: minus_ivl_norm split: ivl.splits prod.splits)

lemma normalize_ivl_ivl_min [simp]: "normalize_ivl (ivl_min a b) = ivl_min a b"
  by (cases a; cases b) simp

lemma normalize_ivl_ivl_max [simp]: "normalize_ivl (ivl_max a b) = ivl_max a b"
  by (cases a; cases b) simp

lemma is_bottom_ivl_times_core [simp]:
  "¬ is_bottom_ivl (ivl_times_core x y)"
proof (induction x y rule: ivl_times_core.induct)
  case (1 l1 u1 l2 u2)
  show ?case
    by (auto simp: is_bottom_ivl_def min_le_iff_disj le_max_iff_disj)
qed (simp_all add: is_bottom_ivl_def ivl_top_def)

lemma normalize_ivl_times [simp]: "normalize_ivl (a * b) = a * b"
  by (cases "ivl_nonempty a ∧ ivl_nonempty b") (auto simp: times_ivl_def)

end
```

12.3 Interval_Backward

```
theory Interval_Backward
  imports Interval_Arithmetic "Voblint_Nonrelational.Exec_Backward"
    "Voblint_Nonrelational.Abstract_Arithmetic" Interval_Numeric_Queries
begin
```

Interval backward filtering

The interval comparison queries, the evaluator `aval_ivl`, and the inverse operators that narrow interval operands against a guard's outcome, packaged as `ivl_refine_ops` with its refinement certificate.

Comparison and truthiness queries

`Voblint_Nonrelational.Abstract_Arithmetic`'s `sound_arith_ops` locale expects three-valued `bool option` queries: `Some True`/`Some False` when the bound-based table decides it, `None` otherwise. The check layer's `interval_less`/`interval_eq` are exactly those queries, so the evaluator uses them too. `interval_tobool` is truthiness against the point interval `[0,0]`.

```
lemma interval_less_unfold:
  "interval_less a b =
     (if interval_less_true a b then Some True
      else if interval_less_false a b then Some False
      else None)"
  by (fact interval_numeric_queries.query_less_def)

lemma interval_eq_unfold:
  "interval_eq a b =
     (if interval_eq_true a b then Some True
      else if interval_eq_false a b then Some False
      else None)"
  by (fact interval_numeric_queries.query_eq_def)

definition interval_tobool :: "ivl ⇒ bool option" where
  "interval_tobool a =
     (if interval_eq_false a (Ivl (Fin 0) (Fin 0)) then Some True
      else if interval_eq_true a (Ivl (Fin 0) (Fin 0)) then Some False
      else None)"

lemma interval_less_ivl_sound:
  assumes "interval_less p q = Some b" and "i ∈ gamma_ivl p" and "j ∈ gamma_ivl q"
  shows "(i < j) = b"
  using assms unfolding interval_less_unfold
  by (auto split: if_splits dest: interval_less_true_sound interval_less_false_sound)

lemma interval_eq_ivl_sound:
  assumes "interval_eq p q = Some b" and "i ∈ gamma_ivl p" and "j ∈ gamma_ivl q"
  shows "(i = j) = b"
  using assms unfolding interval_eq_unfold
  by (auto split: if_splits dest: interval_eq_true_sound interval_eq_false_sound)

lemma interval_tobool_sound:
  assumes "interval_tobool p = Some b" and "i ∈ gamma_ivl p"
  shows "truthy i = b"
proof -
  have z: "(0::int) ∈ gamma_ivl (Ivl (Fin 0) (Fin 0))" by simp
  show ?thesis
    using assms z
    unfolding interval_tobool_def truthy_def
    by (auto split: if_splits dest: interval_eq_false_sound interval_eq_true_sound)
qed

lemma interval_less_mono:
  assumes hp: "¬ is_empty (p1::ivl)" and hq: "¬ is_empty q1"
      and hpm: "p1 ≤ p2" and hqm: "q1 ≤ q2"
      and hwide: "interval_less p2 q2 = Some b"
  shows "interval_less p1 q1 = Some b"
proof -
  obtain l1 u1 where p1_def: "p1 = Ivl l1 u1" by (cases p1)
  obtain l1' u1' where p2_def: "p2 = Ivl l1' u1'" by (cases p2)
  obtain l2 u2 where q1_def: "q1 = Ivl l2 u2" by (cases q1)
  obtain l2' u2' where q2_def: "q2 = Ivl l2' u2'" by (cases q2)
  have ne_p1: "l1 ≤ u1" using hp unfolding p1_def is_empty_ivl is_bottom_ivl_def by auto
  have ne_q1: "l2 ≤ u2" using hq unfolding q1_def is_empty_ivl is_bottom_ivl_def by auto
```

```
  have bnds: "l1' ≤ l1" "u1 ≤ u1'" "l2' ≤ l2" "u2 ≤ u2'"
    using hpm hqm unfolding p1_def p2_def q1_def q2_def less_eq_ivl_def by auto
  have ne_p2: "l1' ≤ u1'" and ne_q2: "l2' ≤ u2'"
    using bnds ne_p1 ne_q1 by order+
  show ?thesis
  proof (cases "interval_less_true p2 q2")
    case True
    then have "u1' < l2'" using p2_def q2_def ne_p2 ne_q2 by simp
    then have "u1 < l2" using bnds by order
    then have p1q1: "interval_less_true p1 q1" using p1_def q1_def ne_p1 ne_q1 by simp
    have "interval_less p2 q2 = Some True" using True unfolding interval_less_unfold by simp
    with hwide have "b = True" by simp
    then show ?thesis using p1q1 unfolding interval_less_unfold by simp
  next
    case False
    then have hlf: "interval_less_false p2 q2"
      using hwide unfolding interval_less_unfold by (auto split: if_splits)
    then have "u2' ≤ l1'" using p2_def q2_def ne_p2 ne_q2 by simp
    then have hle: "u2 ≤ l1" using bnds by order
    then have p1q1f: "interval_less_false p1 q1" using p1_def q1_def ne_p1 ne_q1 by simp
    have p1q1nt: "¬ interval_less_true p1 q1"
    proof
      assume "interval_less_true p1 q1"
      then have "u1 < l2" using p1_def q1_def ne_p1 ne_q1 by simp
      with hle ne_p1 ne_q1 show False by order
    qed
    have "interval_less p2 q2 = Some False" using False hlf unfolding interval_less_unfold by simp
    with hwide have "b = False" by simp
    then show ?thesis using p1q1f p1q1nt unfolding interval_less_unfold by simp
  qed
qed


lemma interval_eq_mono:
  assumes hp: "¬ is_empty (p1::ivl)" and hq: "¬ is_empty q1"
      and hpm: "p1 ≤ p2" and hqm: "q1 ≤ q2"
      and hwide: "interval_eq p2 q2 = Some b"
  shows "interval_eq p1 q1 = Some b"
proof -
  obtain l1 u1 where p1_def: "p1 = Ivl l1 u1" by (cases p1)
  obtain l1' u1' where p2_def: "p2 = Ivl l1' u1'" by (cases p2)
  obtain l2 u2 where q1_def: "q1 = Ivl l2 u2" by (cases q1)
  obtain l2' u2' where q2_def: "q2 = Ivl l2' u2'" by (cases q2)
  have ne_p1: "l1 ≤ u1" using hp unfolding p1_def is_empty_ivl is_bottom_ivl_def by auto
  have ne_q1: "l2 ≤ u2" using hq unfolding q1_def is_empty_ivl is_bottom_ivl_def by auto
  have bnds: "l1' ≤ l1" "u1 ≤ u1'" "l2' ≤ l2" "u2 ≤ u2'"
    using hpm hqm unfolding p1_def p2_def q1_def q2_def less_eq_ivl_def by auto
  have ne_p2: "l1' ≤ u1'" and ne_q2: "l2' ≤ u2'"
    using bnds ne_p1 ne_q1 by order+
  show ?thesis
    using hwide ne_p1 ne_q1 ne_p2 ne_q2 bnds
    unfolding interval_eq_unfold p1_def p2_def q1_def q2_def
    by (auto split: if_splits; order)
qed

lemma interval_tobool_mono:
  assumes hp: "¬ is_empty (p1::ivl)" and hpm: "p1 ≤ p2"
      and hwide: "interval_tobool p2 = Some b"
  shows "interval_tobool p1 = Some b"
proof -
  obtain l1 u1 where p1_def: "p1 = Ivl l1 u1" by (cases p1)
  obtain l1' u1' where p2_def: "p2 = Ivl l1' u1'" by (cases p2)
  have ne_p1: "l1 ≤ u1" using hp unfolding p1_def is_empty_ivl is_bottom_ivl_def by auto
  have bnds: "l1' ≤ l1" "u1 ≤ u1'"
    using hpm unfolding p1_def p2_def less_eq_ivl_def by auto
  have ne_p2: "l1' ≤ u1'" using bnds ne_p1 by order
  show ?thesis
    using hwide ne_p1 ne_p2 bnds
    unfolding interval_tobool_def p1_def p2_def
    by (auto split: if_splits; order)
qed
```

Abstract expression evaluation

`aval_ivl` evaluates an expression bottom-up with the interval operations above. A comparison is `bot` when either operand is empty; otherwise it is the literal its three-valued query decides, or the join of `0` and `1` on `None`.

These are the equations `sound_arith_ops` expects of an evaluator.

```
fun aval_ivl :: "exp => (vname => ivl) => ivl" where
    "aval_ivl (N n)        d = ivl_of_int n"
  | "aval_ivl (V x)        d = d x"
  | "aval_ivl (Plus  a b)  d = aval_ivl a d + aval_ivl b d"
  | "aval_ivl (Minus a b)  d = aval_ivl a d - aval_ivl b d"
  | "aval_ivl (Times a b)  d = aval_ivl a d * aval_ivl b d"
  | "aval_ivl (Div a b)  d = ivl_div (aval_ivl a d) (aval_ivl b d)"
  | "aval_ivl (Mod a b)  d = ivl_mod (aval_ivl a d) (aval_ivl b d)"
  | "aval_ivl (Less a b)   d =
       (if is_empty (aval_ivl a d) ∨ is_empty (aval_ivl b d) then bot
        else of_bool_option ivl_of_int (interval_less (aval_ivl a d) (aval_ivl b d)))"
  | "aval_ivl (LessEq a b)   d =
       (if is_empty (aval_ivl a d) ∨ is_empty (aval_ivl b d) then bot
        else of_bool_option ivl_of_int
               (map_option HOL.Not (interval_less (aval_ivl b d) (aval_ivl a d))))"
  | "aval_ivl (Greater a b)   d =
       (if is_empty (aval_ivl a d) ∨ is_empty (aval_ivl b d) then bot
        else of_bool_option ivl_of_int (interval_less (aval_ivl b d) (aval_ivl a d)))"
  | "aval_ivl (GreaterEq a b)   d =
       (if is_empty (aval_ivl a d) ∨ is_empty (aval_ivl b d) then bot
        else of_bool_option ivl_of_int
               (map_option HOL.Not (interval_less (aval_ivl a d) (aval_ivl b d))))"
  | "aval_ivl (NotEq a b) d =
       (if is_empty (aval_ivl a d) ∨ is_empty (aval_ivl b d) then bot
        else of_bool_option ivl_of_int
               (map_option HOL.Not (interval_eq (aval_ivl a d) (aval_ivl b d))))"
  | "aval_ivl (exp.Eq a b) d =
       (if is_empty (aval_ivl a d) ∨ is_empty (aval_ivl b d) then bot
        else of_bool_option ivl_of_int (interval_eq (aval_ivl a d) (aval_ivl b d)))"
  | "aval_ivl (exp.Not a)  d =
       (if is_empty (aval_ivl a d) then bot
        else of_bool_option ivl_of_int (map_option HOL.Not (interval_tobool (aval_ivl a d))))"
  | "aval_ivl (And a b)    d =
       (if is_empty (aval_ivl a d) ∨ is_empty (aval_ivl b d) then bot
        else of_bool_option ivl_of_int
               (and_opt (interval_tobool (aval_ivl a d)) (interval_tobool (aval_ivl b d))))"
  | "aval_ivl (Or a b)     d =
       (if is_empty (aval_ivl a d) ∨ is_empty (aval_ivl b d) then bot
        else of_bool_option ivl_of_int
               (or_opt (interval_tobool (aval_ivl a d)) (interval_tobool (aval_ivl b d))))"

interpretation ivl_arith: mono_arith_ops
    aval_ivl ivl_of_int "(+)" "(-)" "(*)" ivl_div ivl_mod
    interval_less interval_eq interval_tobool
  by unfold_locales
     (simp_all add: ivl_plus_sound ivl_minus_sound ivl_times_sound ivl_div_sound ivl_mod_sound
                    ivl_plus_mono ivl_minus_mono ivl_times_mono ivl_div_mono ivl_mod_mono
                    interval_less_ivl_sound interval_eq_ivl_sound
                    interval_tobool_sound[unfolded truthy_def]
                    interval_less_mono interval_eq_mono interval_tobool_mono
                    sup_ivl_def)
```

Backward inverse operators

Inverse operators for backward analysis over intervals. `inv_less_ivl` performs precise interval narrowing: on the true branch of `n1 < n2`, the upper bound of `n1` tightens to one below the upper bound of `n2`, and the lower bound of `n2` tightens to one above the lower bound of `n1`. On the false branch (`n1` $\geq$ `n2`), `n1s` lower bound tightens to the lower bound of `n2`, and `n2s` upper bound tightens to the upper bound of `n1`. Plus/minus/times instantiate the shared `inv_conservative` (identity) instead of a per-domain no-op.

```
fun inv_less_ivl :: "bool => ivl => ivl => ivl * ivl" where
    "inv_less_ivl True  (Ivl l1 u1) (Ivl l2 u2) =
       (Ivl l1 u1 ⊓ Ivl MinInf (u2 - Fin 1),
        Ivl l2 u2 ⊓ Ivl (l1 + Fin 1) PlusInf)"
  | "inv_less_ivl False (Ivl l1 u1) (Ivl l2 u2) =
       (Ivl l1 u1 ⊓ Ivl l2 PlusInf,
        Ivl l2 u2 ⊓ Ivl MinInf u1)"

lemma inv_less_ivl_n1_ub:
  "n2 ∈ gamma_ivl (Ivl l2 u2) ⟹ n1 < n2
   ⟹ n1 ∈ gamma_ivl (Ivl MinInf (u2 - Fin 1))"
  by (cases u2; auto; linarith)

lemma inv_less_ivl_n2_lb:
```

```
  "n1 ∈ gamma_ivl (Ivl l1 u1) ⟹ n1 < n2
   ⟹ n2 ∈ gamma_ivl (Ivl (l1 + Fin 1) PlusInf)"
  by (cases l1; auto; linarith)

lemma inv_less_ivl_n1_ge_lb:
  "n2 ∈ gamma_ivl (Ivl l2 u2) ⟹ ¬ n1 < n2
   ⟹ n1 ∈ gamma_ivl (Ivl l2 PlusInf)"
  by (cases l2; auto; linarith)

lemma inv_less_ivl_n2_le_ub:
  "n1 ∈ gamma_ivl (Ivl l1 u1) ⟹ ¬ n1 < n2
   ⟹ n2 ∈ gamma_ivl (Ivl MinInf u1)"
  by (cases u1; auto; linarith)

lemma inv_less_ivl_sound:
  assumes g1: "n1 ∈ gamma_ivl a1" and g2: "n2 ∈ gamma_ivl a2" and eq: "(n1 < n2) = res"
  shows "n1 ∈ gamma_ivl (fst (inv_less_ivl res a1 a2))
       ∧ n2 ∈ gamma_ivl (snd (inv_less_ivl res a1 a2))"
proof -
  obtain l1 u1 where ha1: "a1 = Ivl l1 u1" by (rule ivl_exhaustE)
  obtain l2 u2 where ha2: "a2 = Ivl l2 u2" by (rule ivl_exhaustE)
  show ?thesis
  proof (cases res)
    case True
    have lt: "n1 < n2" using eq True by simp
    have p1: "n1 ∈ gamma_ivl (Ivl l1 u1 ⊓ Ivl MinInf (u2 - Fin 1))"
      by (rule meet_ivl_gamma[OF g1[unfolded ha1] inv_less_ivl_n1_ub[OF g2[unfolded ha2] lt]])
    have p2: "n2 ∈ gamma_ivl (Ivl l2 u2 ⊓ Ivl (l1 + Fin 1) PlusInf)"
      by (rule meet_ivl_gamma[OF g2[unfolded ha2] inv_less_ivl_n2_lb[OF g1[unfolded ha1] lt]])
    show ?thesis using p1 p2 by (simp add: True ha1 ha2)
  next
    case False
    have nlt: "¬ n1 < n2" using eq False by simp
    have p1: "n1 ∈ gamma_ivl (Ivl l1 u1 ⊓ Ivl l2 PlusInf)"
      by (rule meet_ivl_gamma[OF g1[unfolded ha1] inv_less_ivl_n1_ge_lb[OF g2[unfolded ha2] nlt]])
    have p2: "n2 ∈ gamma_ivl (Ivl l2 u2 ⊓ Ivl MinInf u1)"
      by (rule meet_ivl_gamma[OF g2[unfolded ha2] inv_less_ivl_n2_le_ub[OF g1[unfolded ha1] nlt]])
    show ?thesis using p1 p2 by (simp add: False ha1 ha2)
  qed
qed
```

Backward inverse operator for equality

`inv_eq_ivl` narrows on a guard `e1 = e2` known true or false. The true branch narrows both operands to their intersection, the same argument as the sign instance via `meet_ivl_gamma`. The false branch is the sound identity: a precise refinement is possible in specific cases (e.g. excluding a known point value from one bound of the other operand when that point sits exactly at that bound), but `ivl`'s order makes boundary-exclusion tests incompatible with the monotonicity shape required by the backward-domain interface: widening an operand can invalidate a test that matched its old endpoint. The false branch therefore remains the identity. This is a documented precision gap, not a soundness one: `bfilter`'s `Eq _ _ False` case under this instance narrows exactly as much for Interval as it already does today (not at all), while Sign gains real precision from its own instance.

```
fun inv_eq_ivl :: "bool => ivl => ivl => ivl * ivl" where
    "inv_eq_ivl True  a1 a2 = (a1 ⊓ a2, a1 ⊓ a2)"
  | "inv_eq_ivl False a1 a2 = (a1, a2)"

lemma inv_eq_ivl_sound:
  assumes "n1 ∈ gamma_ivl a1" and "n2 ∈ gamma_ivl a2" and "(n1 = n2) = res"
  shows "n1 ∈ gamma_ivl (fst (inv_eq_ivl res a1 a2))
       ∧ n2 ∈ gamma_ivl (snd (inv_eq_ivl res a1 a2))"
proof (cases res)
  case True
  then have "n1 = n2" using assms(3) by simp
  then have "n1 ∈ gamma_ivl a2" using assms(2) by simp
  then have "n1 ∈ gamma_ivl (a1 ⊓ a2)" using meet_ivl_gamma[OF assms(1)] by simp
  then show ?thesis using True ‹n1 = n2› by simp
next
  case False
  then show ?thesis using assms(1,2) by simp
qed

lemma inv_eq_ivl_mono:
  assumes A1: "a1 ≤ (a1' :: ivl)" and A2: "a2 ≤ a2'"
  shows
    "le_pair (inv_eq_ivl r a1 a2) (inv_eq_ivl r a1' a2')"
```

```
proof (cases r)
  case True
  have "a1 ⊓ a2 ≤ a1' ⊓ a2'" by (rule inf_mono[OF A1 A2])
  then show ?thesis using True by simp
next
  case False
  then show ?thesis using A1 A2 by simp
qed


lemma inv_less_ivl_mono:
  assumes a1: "(a1 :: ivl) ≤ a1'" and a2: "(a2 :: ivl) ≤ a2'"
  shows "le_pair (inv_less_ivl res a1 a2) (inv_less_ivl res a1' a2')"
proof -
  obtain l1 u1 where ha1: "a1 = Ivl l1 u1" by (rule ivl_exhaustE)
  obtain l2 u2 where ha2: "a2 = Ivl l2 u2" by (rule ivl_exhaustE)
  obtain l1' u1' where ha1': "a1' = Ivl l1' u1'" by (rule ivl_exhaustE)
  obtain l2' u2' where ha2': "a2' = Ivl l2' u2'" by (rule ivl_exhaustE)
  from a1[unfolded ha1 ha1' less_eq_ivl_def] have ord1: "eint_le l1' l1" "eint_le u1 u1'" by auto
  from a2[unfolded ha2 ha2' less_eq_ivl_def] have ord2: "eint_le l2' l2" "eint_le u2 u2'" by auto
  show ?thesis
  proof (cases res)
    case True
    then have r: "res = True" by simp
    have aux1: "Ivl MinInf (u2 - Fin 1) ≤ Ivl MinInf (u2' - Fin (1::int))"
      by (simp add: less_eq_ivl_def eint_minus_mono[OF ord2(2) eint_le_refl])
    have aux2: "Ivl (l1 + Fin 1) PlusInf ≤ Ivl (l1' + Fin (1::int)) PlusInf"
      by (simp add: less_eq_ivl_def eint_plus_mono[OF ord1(1) eint_le_refl])
    show ?thesis
      unfolding r ha1 ha2 ha1' ha2' inv_less_ivl.simps fst_conv snd_conv
    proof (intro conjI)
      show "Ivl l1 u1 ⊓ Ivl MinInf (u2 - Fin 1) ≤ Ivl l1' u1' ⊓ Ivl MinInf (u2' - Fin 1)"
        by (intro inf_mono a1[unfolded ha1 ha1'] aux1)
      show "Ivl l2 u2 ⊓ Ivl (l1 + Fin 1) PlusInf ≤ Ivl l2' u2' ⊓ Ivl (l1' + Fin 1) PlusInf"
        by (intro inf_mono a2[unfolded ha2 ha2'] aux2)
    qed
  next
    case False
    then have r: "res = False" by simp
    have aux3: "Ivl l2 PlusInf ≤ Ivl l2' PlusInf"
      by (simp add: less_eq_ivl_def ord2(1))
    have aux4: "Ivl MinInf u1 ≤ Ivl MinInf u1'"
      by (simp add: less_eq_ivl_def ord1(2))
    show ?thesis
      unfolding r ha1 ha2 ha1' ha2' inv_less_ivl.simps fst_conv snd_conv
    proof (intro conjI)
      show "Ivl l1 u1 ⊓ Ivl l2 PlusInf ≤ Ivl l1' u1' ⊓ Ivl l2' PlusInf"
        by (intro inf_mono a1[unfolded ha1 ha1'] aux3)
      show "Ivl l2 u2 ⊓ Ivl MinInf u1 ≤ Ivl l2' u2' ⊓ Ivl MinInf u1'"
        by (intro inf_mono a2[unfolded ha2 ha2'] aux4)
    qed
  qed
qed
```

The refinement operations and their certificate

Interval's refinement choices: the normalising intersection, the bound-tightening comparison inverses, and the conservative identity for arithmetic. The guard filters and the branch are derived from this record and the evaluator in `Interval_Transfer`.

The intersection is `intersect_ivl`, not the lattice (⊓). The certificate only requires it to preserve concretizations, to be reductive in both arguments, and to be monotone — never that it is the greatest lower bound of the representation order — and the normalising variant additionally keeps every filtered state canonical, so an infeasible guard stores ⊥ instead of a reversed bound pair. (⊓) itself cannot be normalised without losing the greatest-lower-bound law; `meet_ivl_normalized_breaks_greatest` is that counterexample.

```
definition ivl_refine_ops :: "ivl refine_ops" where
  "ivl_refine_ops =
     ⦇r_tobool = interval_tobool, r_inv_less = inv_less_ivl, r_inv_eq = inv_eq_ivl,
      r_inv_plus = inv_conservative, r_inv_minus = inv_conservative,
      r_inv_times = inv_conservative, r_intersect = intersect_ivl⦈"

lemma ivl_refine_ops_simps [simp]:
  "r_tobool ivl_refine_ops = interval_tobool"
  "r_inv_less ivl_refine_ops = inv_less_ivl"
  "r_inv_eq ivl_refine_ops = inv_eq_ivl"
```

```
  "r_inv_plus ivl_refine_ops = inv_conservative"
  "r_inv_minus ivl_refine_ops = inv_conservative"
  "r_inv_times ivl_refine_ops = inv_conservative"
  "r_intersect ivl_refine_ops = intersect_ivl"
  by (simp_all add: ivl_refine_ops_def)
```

The certificate discharges soundness, monotonicity, and reductiveness together against `mono_refinement`; each `inv_*`'s mono/reductive obligation is one `le_pair` fact, built from the componentwise per-operator lemmas above.

```
lemma ivl_backward_domain:
  "mono_refinement intersect_ivl aval_ivl interval_tobool
     inv_less_ivl inv_eq_ivl inv_conservative inv_conservative inv_conservative"
proof unfold_locales
  fix n :: int and a b :: ivl
  assume "n ∈ γ a" and "n ∈ γ b"
  then have "n ∈ gamma_ivl a" and "n ∈ gamma_ivl b" by simp_all
  then show "n ∈ γ (intersect_ivl a b)" using intersect_ivl_gamma by simp
qed (simp_all add: inv_less_ivl_sound inv_eq_ivl_sound
       interval_tobool_sound[unfolded truthy_def] intersect_ivl_mono ivl_arith.aval_dom_mono
       inv_less_ivl_mono inv_eq_ivl_mono inv_conservative_def
       intersect_ivl_le1 intersect_ivl_le2 interval_tobool_mono
     del: intersect_ivl_def)

end
```

12.4 Interval_Transfer

```
theory Interval_Transfer
  imports
    Interval_Special
    "Voblint_Nonrelational.Nonrelational_Transfer"
begin
```

What each kind of CFG edge does to an interval store

An interval store gives every variable a lower and an upper bound. How an edge of a compiled graph moves such a store is not an Interval question — an assignment re-evaluates its right-hand side and overwrites the target, a guard runs the backward branch, a return writes into the return variable, procedure entry resets the callee frame to `ivl_top` and binds the formals, and skip, body entry and check observation leave the store alone. That is `sound_nonrelational_ops`, proved sound and monotone once for every domain that gives one abstract value per variable.

What is Interval's own are the primitive choices. `ivl_ops` collects the evaluator, the bound-comparison queries, `ivl_refine_ops` — whose comparison inverses make `x < 8` tighten `x`'s upper bound — the special calls, and `ivl_top`. The one interpretation below derives the guard filters, the branch, the check classifier and the transfer from that bundle, and rewrites the locale's special-call dispatch to `special_ivl`, Interval's own `fun`, rather than introducing a second name for it.

```
lemma interval_check_domain:
  "sound_check_query interval_less interval_eq gamma_state aval_ivl"
  by unfold_locales (rule ivl_arith.aval_abs_sound)

definition ivl_ops :: "ivl nonrelational_ops" where
  "ivl_ops = (| n_aval = aval_ivl, n_query = (|q_less = interval_less, q_eq = interval_eq|),
                n_refine = ivl_refine_ops, n_special = ivl_special_ops |)"

lemma ivl_ops_simps [simp]:
  "n_aval ivl_ops = aval_ivl"
  "q_less (n_query ivl_ops) = interval_less"
  "q_eq (n_query ivl_ops) = interval_eq"
  "n_refine ivl_ops = ivl_refine_ops"
  "n_special ivl_ops = ivl_special_ops"
  by (simp_all add: ivl_ops_def)

global_interpretation ivl_tf: mono_nonrelational_ops ivl_ops
  rewrites "(top :: ivl) = ivl_top"
    and "sound_minmax_ops.special_transfer (n_special ivl_ops) (n_aval ivl_ops) = special_ivl"
  defines assign_ivl = ivl_tf.assign
    and skip_ivl = ivl_tf.skip
    and body_ivl = ivl_tf.body
    and event_ivl = ivl_tf.event
    and return_ivl = ivl_tf.ret
    and enter_frame_ivl_for = ivl_tf.enter_frame_for
    and enter_ivl_for = ivl_tf.enter_for
    and enter_ivl_ci_for = ivl_tf.enter_ci_for
    and ivl_tf_abs = ivl_tf.tf_abs
    and afilter_ivl = ivl_tf.backward.afilter
    and feasible_ivl = ivl_tf.backward.feasible
    and bfilter_ivl = ivl_tf.backward.bfilter
    and branch_ivl = ivl_tf.backward.branch
    and bfilter_lifted_ivl = ivl_tf.backward.bfilter_lifted
    and branch_lifted_ivl = ivl_tf.backward.branch_lifted
    and interval_truthy_query = ivl_tf.check.truthy_query
    and interval_check_query = ivl_tf.check.check_query
    and interval_classify_check = ivl_tf.check.classify_check
    and interval_eval_answer = ivl_tf.check.eval_answer
    and interval_checks_proven = ivl_tf.check.abstract_checks_proven
proof -
  show "mono_nonrelational_ops ivl_ops"
  proof (rule mono_nonrelational_opsI, unfold ivl_ops_simps ivl_refine_ops_simps)
    show "mono_minmax_ops ivl_special_ops aval_ivl"
      by (rule ivl_special.mono_minmax_ops_axioms)
    show "mono_refinement intersect_ivl aval_ivl interval_tobool
            inv_less_ivl inv_eq_ivl inv_conservative inv_conservative inv_conservative"
      by (rule ivl_backward_domain)
    show "sound_check_query interval_less interval_eq gamma_state aval_ivl"
      by (rule interval_check_domain)
  qed
qed (simp_all add: special_ivl_eq_transfer fun_eq_iff top_ivl_def)
```

No fact is renamed. The transfer functions get Interval-prefixed names above because they are constants a caller applies, exported to OCaml and named in registration data; the theorems about them stay under `ivl_tf.`, which is where a reader looks to find out that they are the generic ones rather than Interval's own. `skip_ivl`'s soundness is `ivl_tf.skip_sound`, the branch's is `ivl_tf.backward.branch_sound`, and the framework's transfer contract at Interval is `ivl_tf.is_sound_nonrelational_transfer`.

end

12.5 Interval_Domain

```
theory Interval_Domain
  imports Interval_Transfer
begin
```

Integrated Interval domain

This theory provides one import surface for interval bounds, lattice structure, widening, narrowing, arithmetic, backward filtering, transfers, and printing.

Executable examples

```
lemma "(Fin 3 :: eint) + Fin (-1) = Fin 2" by eval
lemma "(PlusInf :: eint) + Fin 100 = PlusInf" by eval
lemma "(Fin 5 :: eint) - Fin 3 = Fin 2" by eval
lemma "(Fin 0 :: eint) - PlusInf = MinInf" by eval

lemma "Ivl (Fin 1) (Fin 3) + Ivl (Fin 2) (Fin 5) = Ivl (Fin 3) (Fin 8)" by eval
lemma "Ivl (Fin 5) (Fin 10) - Ivl (Fin 1) (Fin 3) = Ivl (Fin 2) (Fin 9)" by eval
lemma "Ivl (Fin (-2)) (Fin 3) * Ivl (Fin (-1)) (Fin 4) = Ivl (Fin (- 8)) (Fin 12)" by eval

lemma "join_ivl (Ivl (Fin 1) (Fin 3)) (Ivl (Fin 2) (Fin 5)) = Ivl (Fin 1) (Fin 5)" by eval
lemma "widen_ivl_core (Ivl (Fin 0) (Fin 1)) (Ivl (Fin 0) (Fin 2)) = Ivl (Fin 0) PlusInf" by eval
lemma "widen_ivl_core (Ivl (Fin 1) (Fin 3)) (Ivl (Fin 0) (Fin 3)) = Ivl MinInf (Fin 3)" by eval
lemma "Ivl (Fin 0) (Fin 10) ⊓ Ivl (Fin 3) (Fin 7) = Ivl (Fin 3) (Fin 7)" by eval

lemma "string_of_eint MinInf = STR ''-<infinity>''" by eval
lemma "string_of_eint PlusInf = STR ''+<infinity>''" by eval
lemma "string_of_ivl (Ivl (Fin (-3)) PlusInf) = STR ''[-3,+<infinity>]''" by eval

lemma "aval_ivl (Minus (V (STR ''x'')) (N 1))
      ((λ_. ivl_top)((STR ''x'') := Ivl (Fin 5) (Fin 10)))
    = Ivl (Fin 4) (Fin 9)"
  by eval
lemma "(bfilter_ivl (Less (V (STR ''x'')) (N 5)) True
         ((λ_. ivl_top)((STR ''x'') := Ivl (Fin 0) (Fin 10)))) (STR ''x'')
    = Ivl (Fin 0) (Fin 4)"
  by eval

end
```

12.6 Interval_Exec

```
theory Interval_Exec
  imports
    Interval_Domain
begin
```

Interval on the executable carrier

The executable step and procedure entry are `Voblint_Nonrelational.Nonrelational_Ops`'s generic constructions at the bundle, `generic_tf_st_for ivl_ops` and `generic_enter_st_for ivl_ops`, and their agreement with the abstract transfer is `ivl_tf.tf_st_for_commute` and `ivl_tf.enter_st_for_commute`: nothing about them is the domain's to state.

The state a run starts in: a declared global holds `[0,0]`, a local is unbounded.

```
abbreviation cinit_ivl_st :: "ivl default_st" where
  "cinit_ivl_st ≡ initial_default_st (Ivl MinInf PlusInf) (Ivl (Fin 0) (Fin 0))"

end
```

12.7 Interval_Special

```
theory Interval_Special
  imports Interval_Backward "Voblint_Nonrelational.Special_Ops"
begin
```

Interval: special-call semantics

`ivl_min`/`ivl_max` (in `Interval_Arithmetic`) exist solely as the abstract implementation of the `Min`/`Max` special calls, mirroring how Sign's `sign_min`/`sign_max` exist solely for the same reason.

Special-call dispatch

```
fun special_ivl ::
    "special_call => vname => (vname => ivl) => (vname => ivl)"
where
  "special_ivl Nondet_Int x d = d(x := ivl_top)"
| "special_ivl (Min a b) x d = d(x := ivl_min (aval_ivl a d) (aval_ivl b d))"
| "special_ivl (Max a b) x d = d(x := ivl_max (aval_ivl a d) (aval_ivl b d))"

definition ivl_special_ops :: "ivl special_ops" where
  "ivl_special_ops = (| special_min = ivl_min, special_max = ivl_max |)"

interpretation ivl_special: mono_minmax_ops ivl_special_ops aval_ivl
  by unfold_locales
     (auto simp: ivl_special_ops_def top_ivl_def gamma_ivl_top
           intro: ivl_min_sound ivl_max_sound ivl_min_combine_mono ivl_max_combine_mono)

lemma ivl_special_ops_min [simp]: "special_min ivl_special_ops = ivl_min"
  by (simp add: ivl_special_ops_def)

lemma ivl_special_ops_max [simp]: "special_max ivl_special_ops = ivl_max"
  by (simp add: ivl_special_ops_def)

lemma special_ivl_eq_transfer: "special_ivl sc x d = ivl_special.special_transfer sc x d"
  by (cases sc) (simp_all add: top_ivl_def)

end
```

12.8 Interval_Numeric_Queries

```
theory Interval_Numeric_Queries
  imports Interval_Warrowing "Voblint_Domain.Backward_Numeric_Queries"
begin
```

Interval interpretation of the generic numeric-query interface

Second-domain validation for `sound_numeric_queries` (`Voblint_Domain.Numeric_Queries`): a real, minimal Interval instance, interpreted without touching that generic theory. Entailment/refutation of `<`/`=` reduces to comparing bounds: two intervals' `<` relation is provable exactly when one interval's upper bound is strictly below the other's lower bound; `=` exactly when both are the same singleton point; $\neq$ exactly when the intervals are disjoint. Empty intervals (`l > u`) make every judgment vacuously true, the same role `SBot` plays for Sign.

```
lemma gamma_ivlD:
  assumes "x ∈ gamma_ivl (Ivl l u)"
  shows "l ≤ Fin x" and "Fin x ≤ u"
  using assms
  by auto

lemma ivl_upper_lower_less:
  assumes "Fin i ≤ u" and "u < l" and "l ≤ Fin j"
  shows "i < j"
  using le_less_trans[OF assms(1) less_le_trans[OF assms(2,3)]]
  by (simp add: less_eint_def less_eq_eint_def)

lemma ivl_upper_lower_less_eq:
  assumes "Fin i ≤ u" and "u ≤ l" and "l ≤ Fin j"
  shows "i ≤ j"
  using order_trans[OF assms(1) order_trans[OF assms(2,3)]]
  by (simp add: less_eq_eint_def)

lemma ivl_upper_lower_not_less_eq:
  assumes "Fin j ≤ u" and "u ≤ l" and "l ≤ Fin i"
  shows "¬ i < j"
  using ivl_upper_lower_less_eq[OF assms]
  by auto


lemma ivl_separated_not_equal:
  assumes "Fin i ≤ u1"
      and "u1 < l2"
      and "l2 ≤ Fin j"
  shows "i ≠ j"
  using ivl_upper_lower_less[OF assms]
  by auto

fun interval_less_true :: "ivl ⇒ ivl ⇒ bool" where
  "interval_less_true (Ivl l1 u1) (Ivl l2 u2) =
     (¬ l1 ≤ u1 ∨ ¬ l2 ≤ u2 ∨ u1 < l2)"

lemma interval_less_true_sound:
  assumes query: "interval_less_true a b"
      and ai: "i ∈ gamma_ivl a"
      and bj: "j ∈ gamma_ivl b"
  shows "i < j"
proof -
  obtain l1 u1 where a_def: "a = Ivl l1 u1"
    by (cases a)
  obtain l2 u2 where b_def: "b = Ivl l2 u2"
    by (cases b)

  from ai have i_bounds:
    "l1 ≤ Fin i" "Fin i ≤ u1"
    unfolding a_def
    by (auto dest: gamma_ivlD)

  from bj have j_bounds:
    "l2 ≤ Fin j" "Fin j ≤ u2"
    unfolding b_def
    by (auto dest: gamma_ivlD)

  from i_bounds j_bounds have nonempty:
    "l1 ≤ u1" "l2 ≤ u2"
    by order+
```

```
  with query have separated: "u1 < l2"
    unfolding a_def b_def
    by simp

  show ?thesis
    using i_bounds(2) separated j_bounds(1)
    by (rule ivl_upper_lower_less)
qed


fun interval_less_false :: "ivl ⇒ ivl ⇒ bool" where
  "interval_less_false (Ivl l1 u1) (Ivl l2 u2) =
     (¬ l1 ≤ u1 ∨ ¬ l2 ≤ u2 ∨ u2 ≤ l1)"

lemma interval_less_false_sound:
  assumes query: "interval_less_false a b"
      and ai: "i ∈ gamma_ivl a"
      and bj: "j ∈ gamma_ivl b"
  shows "¬ i < j"
proof -
  obtain l1 u1 where a_def: "a = Ivl l1 u1"
    by (cases a)
  obtain l2 u2 where b_def: "b = Ivl l2 u2"
    by (cases b)

  from ai have i_bounds:
    "l1 ≤ Fin i" "Fin i ≤ u1"
    unfolding a_def
    by (auto dest: gamma_ivlD)

  from bj have j_bounds:
    "l2 ≤ Fin j" "Fin j ≤ u2"
    unfolding b_def
    by (auto dest: gamma_ivlD)

  from i_bounds j_bounds have nonempty:
    "l1 ≤ u1" "l2 ≤ u2"
    by order+

  with query have separated: "u2 ≤ l1"
    unfolding a_def b_def
    by simp

  show ?thesis
    using j_bounds(2) separated i_bounds(1)
    by (rule ivl_upper_lower_not_less_eq)
qed

fun interval_eq_true :: "ivl ⇒ ivl ⇒ bool" where
  "interval_eq_true (Ivl l1 u1) (Ivl l2 u2) =
     (¬ l1 ≤ u1 ∨ ¬ l2 ≤ u2 ∨ (l1 = u1 ∧ l2 = u2 ∧ l1 = l2))"

lemma interval_eq_true_sound:
  assumes query: "interval_eq_true a b"
      and ai: "i ∈ gamma_ivl a"
      and bj: "j ∈ gamma_ivl b"
  shows "i = j"
proof -
  obtain l1 u1 where a_def: "a = Ivl l1 u1"
    by (cases a)
  obtain l2 u2 where b_def: "b = Ivl l2 u2"
    by (cases b)

  from ai have i_bounds:
    "l1 ≤ Fin i" "Fin i ≤ u1"
    unfolding a_def
    by (auto dest: gamma_ivlD)

  from bj have j_bounds:
    "l2 ≤ Fin j" "Fin j ≤ u2"
    unfolding b_def
    by (auto dest: gamma_ivlD)

  from i_bounds j_bounds have nonempty:
    "l1 ≤ u1" "l2 ≤ u2"
```

```
    by (meson order_trans)+

  with query have singleton:
    "l1 = u1" "l2 = u2" "l1 = l2"
    unfolding a_def b_def
    by auto

  from ai singleton have "Fin i = l1"
    unfolding a_def
    using i_bounds(1,2) by order

  moreover from bj singleton have "Fin j = l1"
    unfolding b_def
    using j_bounds(1,2) by order

  ultimately show ?thesis
    by blast
qed

fun interval_eq_false :: "ivl ⇒ ivl ⇒ bool" where
  "interval_eq_false (Ivl l1 u1) (Ivl l2 u2) =
     (¬ l1 ≤ u1 ∨ ¬ l2 ≤ u2 ∨ u1 < l2 ∨ u2 < l1)"

lemma interval_eq_false_sound:
  assumes query: "interval_eq_false a b"
      and ai: "i ∈ gamma_ivl a"
      and bj: "j ∈ gamma_ivl b"
  shows "i ≠ j"
proof -
  obtain l1 u1 where a_def: "a = Ivl l1 u1"
    by (cases a)
  obtain l2 u2 where b_def: "b = Ivl l2 u2"
    by (cases b)

  from ai have i_bounds:
    "l1 ≤ Fin i" "Fin i ≤ u1"
    unfolding a_def
    by (auto dest: gamma_ivlD)

  from bj have j_bounds:
    "l2 ≤ Fin j" "Fin j ≤ u2"
    unfolding b_def
    by (auto dest: gamma_ivlD)

  from i_bounds j_bounds have nonempty:
    "l1 ≤ u1" "l2 ≤ u2"
    by (meson order_trans)+

  with query have separated:
    "u1 < l2 ∨ u2 < l1"
    unfolding a_def b_def
    by force

  then show ?thesis
    using i_bounds(1,2) ivl_separated_not_equal j_bounds(1,2) by blast
qed
```

Goblint-style optional-Boolean queries

`interval_less`/`interval_eq` are the generic packaging of the four bound-comparison judgments above, so one interpretation both introduces them and discharges the query interface. The **defines** clause is what keeps them ordinary top-level constants, with the code equation the check layer needs.

```
global_interpretation interval_numeric_queries:
  numeric_query_judgments
    interval_less_true interval_less_false interval_eq_true interval_eq_false
  defines interval_less = interval_numeric_queries.query_less
    and interval_eq = interval_numeric_queries.query_eq
  by unfold_locales
     (auto intro: interval_less_true_sound interval_eq_true_sound
           dest: interval_less_false_sound interval_eq_false_sound)
```

Semantic intersection versus raw lattice meet

Unlike Sign, whose `SBot` is the lattice's unique empty representative, the raw `ivl` type has infinitely many non-canonical empty representations (any `Ivl l u` with `l > u`, documented at `normalize_ivl`), and only `Ivl PlusInf MinInf` is literally `bot`. Testing the raw `(⊓)` against `bot` therefore under-approximates disjointness: two disjoint, non-empty finite intervals meet to an empty-but-non-canonical result. The witness below is disjoint by `interval_eq_false`'s sound table, yet its representation-level meet is not `bot`.

The generic `eq_false` derived in `sound_refinement`'s own context (`Voblint_Domain.Backward_Numeric_Queries`) instead tests semantic intersection via `is_empty`. `intersect_ivl` returns a value `is_empty` classifies as empty on the same witness, so the generic derivation classifies it correctly regardless of whether that value is the canonical ⊥. Interval also exposes its hand-tuned query table; both operations agree on this disjointness witness. `interval_eq_true` is independent of either operation because it derives equality from `interval_less_false` in both directions.

```
lemma interval_meet_of_witness_not_bot:
  "Ivl (Fin 1) (Fin 2) ⊓ Ivl (Fin 5) (Fin 6) ≠ bot"
  by (simp add: bot_ivl_def)

lemma interval_intersect_of_witness_bot:
  "intersect_ivl (Ivl (Fin 1) (Fin 2)) (Ivl (Fin 5) (Fin 6)) = bot"
  by eval

end
```

12.9 Interval_Point_Digest

theory Interval_Point_Digest
  imports Interval_Domain
begin

Interval point abstraction

The point abstraction maps an integer to its singleton interval.

definition ivl_decode :: "Int.int ⇒ ivl" **where**
  "ivl_decode v = Ivl (Fin v) (Fin v)"

end

12.10 Interval_Sound

theory Interval_Sound
  imports
    Interval_Exec
begin

What the initial abstract state describes

Interval's specification and its soundness are not restated here: `ivl_tf.spec_exec` is sound against `λd g. γ (ϱ_𝒢 d)` by `ivl_tf.sound_exec` and `ivl_tf.entry_cover_exec`, which every non-relational bundle gets from `sound_nonrelational_ops`.

The initial-state contract, owned here rather than reproved at each of the four assembly instances: every concrete store a run may start in is described by `cinit_ivl_st`. A declared global starts at zero, which the singleton interval describes exactly; a local starts unconstrained, which the unbounded interval describes trivially. The solver discipline does not enter into it, which is why one lemma serves all four.

lemma interval_cinit_gamma:
  "cinit_stores 𝒢 ⊆ default_st_gamma 𝒢 cinit_ivl_st"
  by (auto simp: cinit_stores_def gamma_state_def default_st_gamma_initial)

end

12.11 Interval_Classify

theory `Interval_Classify`
  imports
    `Interval_Exec`
begin

Interval instance of the generic check-discharge interface

The check classifier is derived from Interval's bundle, mirroring `Sign_Classify`: the Boolean recursion over `exp` (`Not`, `And`, `Or`), the three-way classification, and the node-indexed bridge to `checks_proven` come from the bundle's bound-comparison queries `interval_less`/`interval_eq` and evaluator, through the `ivl_tf` interpretation in `Voblint_Analysis_Interval.Interval_Transfer`.

The classifier sits below `Interval_Analyses`: each registration there discharges its `ClProved`/`ClRefuted` obligations with `ivl_tf.check.classify_check_proved` and `ivl_tf.check.classify_check_refuted`.

end

12.12 Interval_Analyses

```
theory Interval_Analyses
  imports
    Interval_Sound
    Interval_Classify
    "Voblint_Nonrelational.Nonrel_Registration"
begin
```

Registering Interval at every context and update rule

Interval runs through the shared D/G pipeline at the unit context, keyed by the abstract values a callee's formals hold on entry, and keyed by a bounded call string. The CLI runs it as a field of the combined state of `MCP_Analyses`, whose component and soundness the unit registration supplies. Each registration leaves the rule that merges a value side-effected into a global as a parameter `r`, and the call-string one also its bound `k`. The equation system, the solve, the result table and every soundness endpoint come from the interpreted locale; this theory only names the domain's own implementation and facts.

At the unit context

```
global_interpretation interval_rule:
    nonrel_unit_rule ivl_ops cinit_ivl_st r
  for r
  by (intro nonrel_unit_ruleI nonrel_registrationI
      ivl_tf.sound_nonrelational_ops_axioms interval_cinit_gamma)
```

At the entry-state context

A callee is keyed by the abstract values its formals hold on entry.

```
global_interpretation interval_es_rule:
    nonrel_es_rule ivl_ops cinit_ivl_st r
  for r
  by (intro nonrel_es_ruleI nonrel_registrationI
      ivl_tf.sound_nonrelational_ops_axioms interval_cinit_gamma)
```

At the call-string context

A callee is keyed by its last `k` call sites.

```
global_interpretation interval_cs_rule:
    nonrel_cs_rule ivl_ops cinit_ivl_st k r
  for k r
  by (intro nonrel_cs_ruleI nonrel_registrationI
      ivl_tf.sound_nonrelational_ops_axioms interval_cinit_gamma)

end
```

Chapter 13

Voblint_Analysis_Parity

13.1 Parity_Warrowing

```
theory Parity_Warrowing
  imports "Voblint_Domain.Parity_Lattice"
begin
```

Parity warrowing and domain instance

Parity is finite, so widening is join and narrowing keeps the left argument.

```
instantiation parity :: warrowing begin
  definition "((a :: parity) ∇ b) = join_parity a b"
  definition "((a :: parity) △ b) = a"
instance proof
  fix a b :: parity
  show "a ≤ (a ∇ b)" unfolding less_eq_parity_def widen_parity_def by (rule join_parity_ub1)
  show "b ≤ (a ∇ b)" unfolding less_eq_parity_def widen_parity_def by (rule join_parity_ub2)
  show "b ≤ a ⟹ b ≤ (a △ b)" unfolding narrow_parity_def by simp
  show "b ≤ a ⟹ (a △ b) ≤ a"
    unfolding narrow_parity_def by (simp add: less_eq_parity_def)
qed
end

instantiation parity :: numeric_domain begin
definition gamma_abs_parity [simp]: "numeric_domain_class.gamma (a :: parity) = gamma_parity a"
definition is_empty_parity [simp]: "is_empty (a :: parity) = is_bottom_parity a"
definition to_string_parity [simp]:
  "to_string (a :: parity) = (if is_top_parity a then sym_top else string_of_parity a)"
instance proof
  show "numeric_domain_class.gamma (bot :: parity) = {}" unfolding bot_parity_def by simp
next
  show "numeric_domain_class.gamma (top :: parity) = UNIV" by (simp add: gamma_parity_top)
next
  fix a b :: parity
  assume H: "a ≤ b"
  have "gamma_parity a ⊆ gamma_parity b"
    using H unfolding less_eq_parity_def by (rule gamma_parity_mono)
  then show "numeric_domain_class.gamma a ⊆ numeric_domain_class.gamma b" by simp
next
  fix a :: parity
  show "is_empty a ⟷ numeric_domain_class.gamma a = {}"
    by (simp add: is_bottom_parity_correct)
qed
end

end
```

13.2 Parity_Domain

```
theory Parity_Domain
  imports Parity_Warrowing "Voblint_Nonrelational.Abstract_Arithmetic"
begin
```

Parity arithmetic and expression evaluation

Parity arithmetic, comparison queries and the evaluator `aval_parity`, with the soundness and monotonicity facts `sound_arith_ops` asks of a domain.

Abstract arithmetic

Addition and subtraction follow the parity rules of `int`; a product is even as soon as one factor is. Any bottom operand yields `PBot`.

```
instantiation parity :: plus begin
fun plus_parity :: "parity => parity => parity" where
    "plus_parity PBot  _     = PBot"
  | "plus_parity _     PBot  = PBot"
  | "plus_parity PEven PEven = PEven"
  | "plus_parity POdd  POdd  = PEven"
  | "plus_parity PEven POdd  = POdd"
  | "plus_parity POdd  PEven = POdd"
  | "plus_parity _     _     = PTop"
instance ..
end

instantiation parity :: minus begin
fun minus_parity :: "parity => parity => parity" where
    "minus_parity PBot  _     = PBot"
  | "minus_parity _     PBot  = PBot"
  | "minus_parity PEven PEven = PEven"
  | "minus_parity POdd  POdd  = PEven"
  | "minus_parity PEven POdd  = POdd"
  | "minus_parity POdd  PEven = POdd"
  | "minus_parity _     _     = PTop"
instance ..
end

instantiation parity :: times begin
fun times_parity :: "parity => parity => parity" where
    "times_parity PBot  _     = PBot"
  | "times_parity _     PBot  = PBot"
  | "times_parity PEven _     = PEven"
  | "times_parity _     PEven = PEven"
  | "times_parity POdd  POdd  = POdd"
  | "times_parity _     _     = PTop"
instance ..
end

lemma parity_plus_sound:
  "i ∈ gamma_parity a ⟹ j ∈ gamma_parity b ⟹ i + j ∈ gamma_parity (a + b)"
  by (cases a; cases b; auto)

lemma parity_minus_sound:
  "i ∈ gamma_parity a ⟹ j ∈ gamma_parity b ⟹ i - j ∈ gamma_parity (a - b)"
  by (cases a; cases b; auto)

lemma parity_times_sound:
  "i ∈ gamma_parity a ⟹ j ∈ gamma_parity b ⟹ i * j ∈ gamma_parity (a * b)"
  by (cases a; cases b; auto)
```

Monotonicity of arithmetic

Each operation is monotone in each argument by case analysis; the `_combine_mono` lemmas join the two one-sided facts.

```
lemma parity_plus_mono1: "a1 ≤ a2 ⟹ a1 + b ≤ a2 + (b::parity)"
  unfolding less_eq_parity_def by (cases a1; cases a2; cases b; simp)
lemma parity_plus_mono2: "b1 ≤ b2 ⟹ a + b1 ≤ a + (b2::parity)"
  unfolding less_eq_parity_def by (cases a; cases b1; cases b2; simp)
lemma parity_minus_mono1: "a1 ≤ a2 ⟹ a1 - b ≤ a2 - (b::parity)"
  unfolding less_eq_parity_def by (cases a1; cases a2; cases b; simp)
```

```
lemma parity_minus_mono2: "b1 ≤ b2 ⟹ a - b1 ≤ a - (b2::parity)"
  unfolding less_eq_parity_def by (cases a; cases b1; cases b2; simp)
lemma parity_times_mono1: "a1 ≤ a2 ⟹ a1 * b ≤ a2 * (b::parity)"
  unfolding less_eq_parity_def by (cases a1; cases a2; cases b; simp)
lemma parity_times_mono2: "b1 ≤ b2 ⟹ a * b1 ≤ a * (b2::parity)"
  unfolding less_eq_parity_def by (cases a; cases b1; cases b2; simp)

lemma parity_plus_combine_mono: "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ a1 + b1 ≤ a2 + (b2::parity)"
  by (meson order.trans parity_plus_mono1 parity_plus_mono2)

lemma parity_minus_combine_mono: "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ a1 - b1 ≤ a2 - (b2::parity)"
  by (meson order.trans parity_minus_mono1 parity_minus_mono2)

lemma parity_times_combine_mono: "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ a1 * b1 ≤ a2 * (b2::parity)"
  by (meson order.trans parity_times_mono1 parity_times_mono2)
```

Comparison and truthiness queries

Parity carries no ordering information, so `parity_lt` is always `None`. Equality is decidable exactly when the two operands have opposite parity – an even value can never equal an odd one – and truthiness is decidable exactly for `POdd`, since every odd integer is nonzero.

```
fun parity_lt :: "parity ⇒ parity ⇒ bool option" where
  "parity_lt _ _ = None"

fun parity_eqb :: "parity ⇒ parity ⇒ bool option" where
    "parity_eqb PEven POdd = Some False"
  | "parity_eqb POdd PEven = Some False"
  | "parity_eqb _ _ = None"

fun parity_tobool :: "parity ⇒ bool option" where
    "parity_tobool POdd = Some True"
  | "parity_tobool _ = None"

lemma parity_lt_sound:
  "parity_lt a b = Some c ⟹ i ∈ gamma_parity a ⟹ j ∈ gamma_parity b ⟹ (i < j) = c"
  by simp

lemma parity_eqb_sound:
  "parity_eqb a b = Some c ⟹ i ∈ gamma_parity a ⟹ j ∈ gamma_parity b ⟹ (i = j) = c"
  by (cases a; cases b; auto)

lemma parity_tobool_sound:
  "parity_tobool a = Some c ⟹ i ∈ gamma_parity a ⟹ (i ≠ 0) = c"
  by (cases a; auto)

lemma parity_lt_mono:
  "¬ is_empty (a1::parity) ⟹ ¬ is_empty b1 ⟹ a1 ≤ a2 ⟹ b1 ≤ b2 ⟹
   parity_lt a2 b2 = Some c ⟹ parity_lt a1 b1 = Some c"
  by simp

lemma parity_eqb_mono:
  "¬ is_empty (a1::parity) ⟹ ¬ is_empty b1 ⟹ a1 ≤ a2 ⟹ b1 ≤ b2 ⟹
   parity_eqb a2 b2 = Some c ⟹ parity_eqb a1 b1 = Some c"
  unfolding is_empty_parity is_bottom_parity_def less_eq_parity_def
  by (cases a1; cases a2; cases b1; cases b2; simp)

lemma parity_tobool_mono:
  "¬ is_empty (a1::parity) ⟹ a1 ≤ a2 ⟹ parity_tobool a2 = Some c ⟹ parity_tobool a1 = Some c"
  unfolding is_empty_parity is_bottom_parity_def less_eq_parity_def
  by (cases a1; cases a2; simp)


definition parity_div :: "parity ⇒ parity ⇒ parity" where
  "parity_div a b = (if a = PBot ∨ b = PBot then PBot else PTop)"

definition parity_mod :: "parity ⇒ parity ⇒ parity" where
  "parity_mod a b =
    (if a = PBot ∨ b = PBot then PBot else if b = PEven then a else PTop)"

lemma parity_div_sound:
  "i ∈ gamma_parity a ⟹ j ∈ gamma_parity b ⟹
    c_div i j ∈ gamma_parity (parity_div a b)"
  by (cases a; cases b; simp add: parity_div_def)

lemma parity_mod_sound:
```

```
  "i ∈ gamma_parity a ⟹ j ∈ gamma_parity b ⟹
    c_mod i j ∈ gamma_parity (parity_mod a b)"
  by (cases a; cases b; auto simp: parity_mod_def c_mod_def)

lemma parity_div_mono:
  "a1 ≤ a2 ⟹ b1 ≤ b2 ⟹ parity_div a1 b1 ≤ parity_div a2 b2"
  by (cases a1; cases a2; cases b1; cases b2;
      simp add: parity_div_def less_eq_parity_def)

lemma parity_mod_mono:
  "a1 ≤ a2 ⟹ b1 ≤ b2 ⟹ parity_mod a1 b1 ≤ parity_mod a2 b2"
  by (cases a1; cases a2; cases b1; cases b2;
      simp add: parity_mod_def less_eq_parity_def)
```

Abstract expression evaluation

`aval_parity` evaluates an expression bottom-up with the parity operations above. A comparison is `bot` when either operand is empty; otherwise it is the literal its three-valued query decides, or the join of `0` and `1` on `None`. These are the equations `sound_arith_ops` expects of an evaluator.

```
fun aval_parity :: "exp => (vname => parity) => parity" where
    "aval_parity (N n)       d = parity_of_int n"
  | "aval_parity (V v)       d = d v"
  | "aval_parity (Plus  a b) d = aval_parity a d + aval_parity b d"
  | "aval_parity (Minus a b) d = aval_parity a d - aval_parity b d"
  | "aval_parity (Times a b) d = aval_parity a d * aval_parity b d"
  | "aval_parity (Div a b) d = parity_div (aval_parity a d) (aval_parity b d)"
  | "aval_parity (Mod a b) d = parity_mod (aval_parity a d) (aval_parity b d)"
  | "aval_parity (Less a b)  d =
       (if is_empty (aval_parity a d) ∨ is_empty (aval_parity b d) then bot
        else of_bool_option parity_of_int (parity_lt (aval_parity a d) (aval_parity b d)))"
  | "aval_parity (LessEq a b)  d =
       (if is_empty (aval_parity a d) ∨ is_empty (aval_parity b d) then bot
        else of_bool_option parity_of_int
               (map_option HOL.Not (parity_lt (aval_parity b d) (aval_parity a d))))"
  | "aval_parity (Greater a b)  d =
       (if is_empty (aval_parity a d) ∨ is_empty (aval_parity b d) then bot
        else of_bool_option parity_of_int (parity_lt (aval_parity b d) (aval_parity a d)))"
  | "aval_parity (GreaterEq a b)  d =
       (if is_empty (aval_parity a d) ∨ is_empty (aval_parity b d) then bot
        else of_bool_option parity_of_int
               (map_option HOL.Not (parity_lt (aval_parity a d) (aval_parity b d))))"
  | "aval_parity (NotEq a b) d =
       (if is_empty (aval_parity a d) ∨ is_empty (aval_parity b d) then bot
        else of_bool_option parity_of_int
               (map_option HOL.Not (parity_eqb (aval_parity a d) (aval_parity b d))))"
  | "aval_parity (exp.Eq a b) d =
       (if is_empty (aval_parity a d) ∨ is_empty (aval_parity b d) then bot
        else of_bool_option parity_of_int (parity_eqb (aval_parity a d) (aval_parity b d)))"
  | "aval_parity (exp.Not a)  d =
       (if is_empty (aval_parity a d) then bot
        else of_bool_option parity_of_int (map_option HOL.Not (parity_tobool (aval_parity a d))))"
  | "aval_parity (And a b)    d =
       (if is_empty (aval_parity a d) ∨ is_empty (aval_parity b d) then bot
        else of_bool_option parity_of_int
               (and_opt (parity_tobool (aval_parity a d)) (parity_tobool (aval_parity b d))))"
  | "aval_parity (Or a b)     d =
       (if is_empty (aval_parity a d) ∨ is_empty (aval_parity b d) then bot
        else of_bool_option parity_of_int
               (or_opt (parity_tobool (aval_parity a d)) (parity_tobool (aval_parity b d))))"

interpretation parity_arith: mono_arith_ops
    aval_parity parity_of_int "(+)" "(-)" "(*)" parity_div parity_mod
    parity_lt parity_eqb parity_tobool
  apply unfold_locales
  apply (simp_all add: parity_of_int_gamma parity_plus_sound parity_minus_sound parity_times_sound
    parity_div_sound parity_mod_sound
                       parity_plus_combine_mono parity_minus_combine_mono parity_times_combine_mono
                         parity_div_mono parity_mod_mono
                       parity_lt_sound parity_eqb_sound parity_tobool_sound[unfolded truthy_def]
                       sup_parity_def
                  del: parity_lt.simps parity_eqb.simps parity_tobool.simps)
  apply (blast intro: parity_lt_mono[unfolded is_empty_parity]
                      parity_eqb_mono[unfolded is_empty_parity]
                      parity_tobool_mono[unfolded is_empty_parity])+
  done
```

end

13.3 Parity_Special

```
theory Parity_Special
  imports Parity_Domain "Voblint_Nonrelational.Special_Ops"
begin
```

Parity: special-call semantics

`parity_min`/`parity_max` exist solely as the abstract implementation of the `Min`/`Max` special calls; see the analogous note on Sign's `sign_min`/ `sign_max`. Unlike Sign and Interval, parity's result is not simply "top unless we're being lazy": `min x y` and `max x y` both evaluate to exactly one of their two arguments, never a newly synthesized value, so when both arguments share a known parity the result provably shares it too – `parity_min PEven PEven = PEven`, not `PTop`. Only a parity mismatch (or an unknown operand) loses the information, because then either argument could be the one selected.

```
fun parity_min :: "parity => parity => parity" where
    "parity_min PBot _        = PBot"
  | "parity_min _    PBot     = PBot"
  | "parity_min PEven PEven   = PEven"
  | "parity_min POdd  POdd    = POdd"
  | "parity_min _    _        = PTop"

fun parity_max :: "parity => parity => parity" where
    "parity_max PBot _        = PBot"
  | "parity_max _    PBot     = PBot"
  | "parity_max PEven PEven   = PEven"
  | "parity_max POdd  POdd    = POdd"
  | "parity_max _    _        = PTop"

lemma parity_min_sound:
  assumes "i ∈ gamma_parity a" "j ∈ gamma_parity b"
  shows "min i j ∈ gamma_parity (parity_min a b)"
  using assms by (cases a; cases b; auto simp: min_def)

lemma parity_max_sound:
  assumes "i ∈ gamma_parity a" "j ∈ gamma_parity b"
  shows "max i j ∈ gamma_parity (parity_max a b)"
  using assms by (cases a; cases b; auto simp: max_def)

lemma parity_min_mono1:
  "a1 ≤ a2 ⟹ parity_min a1 b ≤ parity_min a2 (b::parity)"
  unfolding less_eq_parity_def
  by (cases a1; cases a2; cases b; simp)

lemma parity_min_mono2:
  "b1 ≤ b2 ⟹ parity_min a b1 ≤ parity_min a (b2::parity)"
  unfolding less_eq_parity_def
  by (cases a; cases b1; cases b2; simp)

lemma parity_min_combine_mono:
  "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ parity_min a1 b1 ≤ parity_min a2 (b2::parity)"
  by (meson order.trans parity_min_mono1 parity_min_mono2)

lemma parity_max_mono1:
  "a1 ≤ a2 ⟹ parity_max a1 b ≤ parity_max a2 (b::parity)"
  unfolding less_eq_parity_def
  by (cases a1; cases a2; cases b; simp)

lemma parity_max_mono2:
  "b1 ≤ b2 ⟹ parity_max a b1 ≤ parity_max a (b2::parity)"
  unfolding less_eq_parity_def
  by (cases a; cases b1; cases b2; simp)

lemma parity_max_combine_mono:
  "⟦a1 ≤ a2; b1 ≤ b2⟧ ⟹ parity_max a1 b1 ≤ parity_max a2 (b2::parity)"
  by (meson order.trans parity_max_mono1 parity_max_mono2)
```

Special-call dispatch

`special_parity` havocs on `Nondet_Int` and applies `parity_min`/`parity_max` otherwise. `special_parity_eq_transfer` shows it equals the generic transfer of the `mono_minmax_ops` interpretation at `parity_special_ops`.

```
fun special_parity ::
    "special_call => vname => (vname => parity) => (vname => parity)"
where
```

```
  "special_parity Nondet_Int x d = d(x := PTop)"
| "special_parity (Min a b) x d = d(x := parity_min (aval_parity a d) (aval_parity b d))"
| "special_parity (Max a b) x d = d(x := parity_max (aval_parity a d) (aval_parity b d))"

definition parity_special_ops :: "parity special_ops" where
  "parity_special_ops = (| special_min = parity_min, special_max = parity_max |)"

interpretation parity_special: mono_minmax_ops parity_special_ops aval_parity
  by unfold_locales
     (auto simp: parity_special_ops_def gamma_parity_top
           intro: parity_min_sound parity_max_sound parity_min_combine_mono parity_max_combine_mono)

lemma parity_special_ops_min [simp]: "special_min parity_special_ops = parity_min"
  by (simp add: parity_special_ops_def)

lemma parity_special_ops_max [simp]: "special_max parity_special_ops = parity_max"
  by (simp add: parity_special_ops_def)

lemma special_parity_eq_transfer:
  "special_parity sc x d = parity_special.special_transfer sc x d"
  by (cases sc) (simp_all add: top_parity_def)

end
```

13.4 Parity_Backward

```
theory Parity_Backward
  imports
    Parity_Domain
    "Voblint_Nonrelational.Exec_Backward"
begin
```

Parity backward refinement

A guard tells Parity something whenever it pins a sum, a difference, a product or an equality. Parity of a sum or difference is the exclusive or of its operands' parities, so knowing the result and one operand fixes the other; an odd product forces both factors odd; and an equality that held gives both sides the meet of their parities. An order comparison says nothing about parity, so its inverse is the conservative no-op.

Inverse operators

```
definition inv_eq_parity :: "bool ⇒ parity ⇒ parity ⇒ parity × parity" where
  "inv_eq_parity res a b = (if res then (a ⊓ b, a ⊓ b) else (a, b))"

definition inv_plus_parity :: "parity ⇒ parity ⇒ parity ⇒ parity × parity" where
  "inv_plus_parity r a b = (a ⊓ (r - b), b ⊓ (r - a))"

definition inv_minus_parity :: "parity ⇒ parity ⇒ parity ⇒ parity × parity" where
  "inv_minus_parity r a b = (a ⊓ (r + b), b ⊓ (a - r))"
```

The parity both factors of a product with result parity `r` must have.

```
fun factor_parity :: "parity ⇒ parity" where
    "factor_parity PBot = PBot"
  | "factor_parity POdd = POdd"
  | "factor_parity _ = PTop"

definition inv_times_parity :: "parity ⇒ parity ⇒ parity ⇒ parity × parity" where
  "inv_times_parity r a b = (a ⊓ factor_parity r, b ⊓ factor_parity r)"

lemma inv_eq_parity_sound:
  "n1 ∈ gamma_parity a1 ⟹ n2 ∈ gamma_parity a2 ⟹ (n1 = n2) = res
   ⟹ n1 ∈ gamma_parity (fst (inv_eq_parity res a1 a2))
     ∧ n2 ∈ gamma_parity (snd (inv_eq_parity res a1 a2))"
  by (auto simp: inv_eq_parity_def)

lemma inv_plus_parity_sound:
  assumes "n1 ∈ gamma_parity a1" and "n2 ∈ gamma_parity a2" and "n1 + n2 ∈ gamma_parity r"
  shows "n1 ∈ gamma_parity (fst (inv_plus_parity r a1 a2))
     ∧ n2 ∈ gamma_parity (snd (inv_plus_parity r a1 a2))"
  using assms parity_minus_sound[OF assms(3) assms(2)] parity_minus_sound[OF assms(3) assms(1)]
  by (simp add: inv_plus_parity_def)

lemma inv_minus_parity_sound:
  assumes "n1 ∈ gamma_parity a1" and "n2 ∈ gamma_parity a2" and "n1 - n2 ∈ gamma_parity r"
  shows "n1 ∈ gamma_parity (fst (inv_minus_parity r a1 a2))
     ∧ n2 ∈ gamma_parity (snd (inv_minus_parity r a1 a2))"
  using assms parity_plus_sound[OF assms(3) assms(2)] parity_minus_sound[OF assms(1) assms(3)]
  by (simp add: inv_minus_parity_def)

lemma factor_parity_sound:
  "n1 * n2 ∈ gamma_parity r ⟹ n1 ∈ gamma_parity (factor_parity r) ∧ n2 ∈ gamma_parity (factor_parity r)"
  by (cases r) auto

lemma inv_times_parity_sound:
  "n1 ∈ gamma_parity a1 ⟹ n2 ∈ gamma_parity a2 ⟹ n1 * n2 ∈ gamma_parity r
   ⟹ n1 ∈ gamma_parity (fst (inv_times_parity r a1 a2))
     ∧ n2 ∈ gamma_parity (snd (inv_times_parity r a1 a2))"
  using factor_parity_sound by (simp add: inv_times_parity_def)

lemma factor_parity_mono: "r1 ≤ r2 ⟹ factor_parity r1 ≤ factor_parity r2"
  by (cases r1; cases r2) (simp_all add: less_eq_parity_def)

lemma inv_eq_parity_mono:
  "x1 ≤ x2 ⟹ y1 ≤ y2 ⟹ le_pair (inv_eq_parity res x1 y1) (inv_eq_parity res x2 y2)"
  by (auto simp: inv_eq_parity_def intro: le_infI1 le_infI2)

lemma inv_plus_parity_mono:
  "r1 ≤ r2 ⟹ x1 ≤ x2 ⟹ y1 ≤ y2 ⟹ le_pair (inv_plus_parity r1 x1 y1) (inv_plus_parity r2 x2 y2)"
```

```
  by (auto simp: inv_plus_parity_def intro: le_infI1 le_infI2 parity_minus_combine_mono)

lemma inv_minus_parity_mono:
  "r1 ≤ r2 ⟹ x1 ≤ x2 ⟹ y1 ≤ y2 ⟹ le_pair (inv_minus_parity r1 x1 y1) (inv_minus_parity r2 x2 y2)"
  by (auto simp: inv_minus_parity_def
      intro: le_infI1 le_infI2 parity_plus_combine_mono parity_minus_combine_mono)

lemma inv_times_parity_mono:
  "r1 ≤ r2 ⟹ x1 ≤ x2 ⟹ y1 ≤ y2 ⟹ le_pair (inv_times_parity r1 x1 y1) (inv_times_parity r2 x2 y2)"
  by (auto simp: inv_times_parity_def intro: le_infI1 le_infI2 factor_parity_mono)
```

The refinement operations and their certificate

`parity_refine_ops` bundles the parity inverses; parity has no ordering, so `r_inv_less` is `inv_conservative`. `parity_backward_domain` certifies the bundle as a monotone refinement over `aval_parity`.

```
definition parity_refine_ops :: "parity refine_ops" where
  "parity_refine_ops =
     (|r_tobool = parity_tobool, r_inv_less = inv_conservative, r_inv_eq = inv_eq_parity,
      r_inv_plus = inv_plus_parity, r_inv_minus = inv_minus_parity,
      r_inv_times = inv_times_parity, r_intersect = inf|)"

lemma parity_refine_ops_simps [simp]:
  "r_tobool parity_refine_ops = parity_tobool"
  "r_inv_less parity_refine_ops = inv_conservative"
  "r_inv_eq parity_refine_ops = inv_eq_parity"
  "r_inv_plus parity_refine_ops = inv_plus_parity"
  "r_inv_minus parity_refine_ops = inv_minus_parity"
  "r_inv_times parity_refine_ops = inv_times_parity"
  "r_intersect parity_refine_ops = inf"
  by (simp_all add: parity_refine_ops_def)

lemma parity_backward_domain:
  "mono_refinement inf aval_parity parity_tobool
     inv_conservative inv_eq_parity inv_plus_parity inv_minus_parity inv_times_parity"
proof unfold_locales
qed (simp_all add: inv_conservative_def inv_eq_parity_sound inv_plus_parity_sound
       inv_minus_parity_sound inv_times_parity_sound parity_tobool_sound[unfolded truthy_def]
       inf_mono parity_arith.aval_dom_mono inv_eq_parity_mono inv_plus_parity_mono
       inv_minus_parity_mono inv_times_parity_mono parity_tobool_mono le_infI1 le_infI2)

end
```

13.5 Parity_Transfer

```
theory Parity_Transfer
  imports
    Parity_Backward
    Parity_Special
    Parity_Numeric_Queries
    "Voblint_Nonrelational.Nonrelational_Transfer"
begin
```

What each kind of CFG edge does to a parity store

A parity store records, for each variable, whether it is even, odd, either (`PTop`) or unreachable (`PBot`). How an edge of a compiled graph moves such a store is not a Parity question — an assignment re-evaluates its right-hand side and overwrites the target, a guard runs the backward branch, a return writes into the return variable, procedure entry resets the callee frame and binds the formals — so it is `sound_nonrelational_ops` that says it, once, for every domain that gives one abstract value per variable.

What is Parity's own are the primitive choices. `parity_ops` collects the evaluator, the comparison queries, `parity_refine_ops` — whose inverses let `x + 1 == y` with `y` even make `x` odd — the special calls, and `PTop`. The one interpretation below derives the guard filters, the branch, the check classifier and the transfer from that bundle, and rewrites the locale's special-call dispatch to `special_parity`, Parity's own `fun`, rather than introducing a second name for it.

```
lemma parity_check_domain: "sound_check_query parity_less parity_eq gamma_state aval_parity"
  by unfold_locales (rule parity_arith.aval_abs_sound)

definition parity_ops :: "parity nonrelational_ops" where
  "parity_ops = (| n_aval = aval_parity, n_query = (|q_less = parity_less, q_eq = parity_eq|),
                  n_refine = parity_refine_ops, n_special = parity_special_ops |)"

lemma parity_ops_simps [simp]:
  "n_aval parity_ops = aval_parity"
  "q_less (n_query parity_ops) = parity_less"
  "q_eq (n_query parity_ops) = parity_eq"
  "n_refine parity_ops = parity_refine_ops"
  "n_special parity_ops = parity_special_ops"
  by (simp_all add: parity_ops_def)

global_interpretation parity_tf: mono_nonrelational_ops parity_ops
  rewrites "(top :: parity) = PTop"
    and "sound_minmax_ops.special_transfer (n_special parity_ops) (n_aval parity_ops)
           = special_parity"
  defines assign_parity = parity_tf.assign
    and skip_parity = parity_tf.skip
    and body_parity = parity_tf.body
    and event_parity = parity_tf.event
    and return_parity = parity_tf.ret
    and enter_frame_parity_for = parity_tf.enter_frame_for
    and enter_parity_for = parity_tf.enter_for
    and enter_parity_ci_for = parity_tf.enter_ci_for
    and parity_tf_abs = parity_tf.tf_abs
    and afilter_parity = parity_tf.backward.afilter
    and feasible_parity = parity_tf.backward.feasible
    and bfilter_parity = parity_tf.backward.bfilter
    and branch_parity = parity_tf.backward.branch
    and parity_truthy_query = parity_tf.check.truthy_query
    and parity_check_query = parity_tf.check.check_query
    and parity_classify_check = parity_tf.check.classify_check
    and parity_eval_answer = parity_tf.check.eval_answer
    and parity_checks_proven = parity_tf.check.abstract_checks_proven
proof -
  show "mono_nonrelational_ops parity_ops"
  proof (rule mono_nonrelational_opsI, unfold parity_ops_simps parity_refine_ops_simps)
    show "mono_minmax_ops parity_special_ops aval_parity"
      by (rule parity_special.mono_minmax_ops_axioms)
    show "mono_refinement inf aval_parity parity_tobool
            inv_conservative inv_eq_parity inv_plus_parity inv_minus_parity inv_times_parity"
      by (rule parity_backward_domain)
    show "sound_check_query parity_less parity_eq gamma_state aval_parity"
      by (rule parity_check_domain)
  qed
qed (simp_all add: special_parity_eq_transfer fun_eq_iff top_parity_def)
```

No fact is renamed. The transfer functions get Parity-prefixed names above because they are constants a

caller applies, exported to OCaml and named in registration data; the theorems about them stay under `parity_tf.`, which is where a reader looks to find out that they are the generic ones rather than Parity's own. `skip_parity`'s soundness is `parity_tf.skip_sound`, the branch's is `parity_tf.backward.branch_sound`, and the framework's transfer contract at Parity is `parity_tf.is_sound_nonrelational_transfer`.

end

13.6 Parity_Exec

```
theory Parity_Exec
  imports
    Parity_Transfer
begin
```

Parity on the executable carrier

The executable step and procedure entry are `Voblint_Nonrelational.Nonrelational_Ops`'s generic constructions at the bundle, `generic_tf_st_for parity_ops` and `generic_enter_st_for parity_ops`, and their agreement with the abstract transfer is `parity_tf.tf_st_for_commute` and `parity_tf.enter_st_for_commute`: nothing about them is the domain's to state.

The state a run starts in: a declared global holds `PEven`, a local `PTop`.

```
abbreviation cinit_parity_st :: "parity default_st" where
  "cinit_parity_st ≡ initial_default_st PTop PEven"

end
```

13.7 Parity_Numeric_Queries

```
theory Parity_Numeric_Queries
  imports Parity_Domain
begin
```

Parity interpretation of the generic numeric-query interface

Third-domain validation for `sound_numeric_queries` (`Voblint_Domain.Numeric_Queries`), hand-tuned like `Voblint_Analysis_Int` rather than derived through `sound_refinement`: parity has no `inv_less` instance (guards do not refine parity), so there is nothing for the generic derivations to read off. Parity carries no order information at all, so `parity_less_true`/`parity_less_false` are provable only in the vacuous case (one side `PBot`); every genuine pair is unknown. `PBot` is the lattice's unique empty representative (unlike Interval's raw `ivl` type, which has infinitely many non-canonical empty representations), so the vacuous case is a single clean equality test, as for Sign's `SBot`.

Comparison judgments: vacuous only

`parity_less_true a b`/`parity_less_false a b` both hold exactly when one side concretizes to nothing: neither judgment can ever be established from parity alone, since knowing a value's evenness says nothing about its order against another value's evenness. The two coincide definitionally — both are the same vacuity test — which is itself the content of the "usually unknown" fact about Parity's order queries.

```
definition parity_vacuous :: "parity ⇒ parity ⇒ bool" where
  "parity_vacuous a b ⟷ (a = PBot ∨ b = PBot)"

definition parity_less_true :: "parity ⇒ parity ⇒ bool" where
  "parity_less_true = parity_vacuous"

definition parity_less_false :: "parity ⇒ parity ⇒ bool" where
  "parity_less_false = parity_vacuous"

lemma parity_less_true_sound:
  assumes "parity_less_true a b" and "i ∈ gamma_parity a" and "j ∈ gamma_parity b"
  shows "i < j"
  using assms unfolding parity_less_true_def parity_vacuous_def by (cases a; cases b; auto)

lemma parity_less_false_sound:
  assumes "parity_less_false a b" and "i ∈ gamma_parity a" and "j ∈ gamma_parity b"
  shows "¬ i < j"
  using assms unfolding parity_less_false_def parity_vacuous_def by (cases a; cases b; auto)
```

Equality judgments

`parity_eq_true` is vacuous-only for the same reason as the order queries: no parity class concretizes to a singleton (unlike Sign's `SZero` or an Interval point interval), so no non-vacuous pair is ever provably equal. `parity_eq_true a b` asks provable equality of every witness pair, so a reflexive-looking check such as `x == x` is *not* classified true by this interface either: `aval_abs` only ever hands both sides the same *abstract* value `PEven`/`POdd`/`PTop`, which still ranges over infinitely many concrete integers, so `eq_true` stays false there too — correctly: this classification layer never assumes two occurrences of the same syntactic expression carry a hidden equality constraint.

`parity_eq_true` is genuinely provable to be true for identical parity classes when combined at the check level with the compiler-independent evaluator; this file states it as it is, no more.

```
definition parity_eq_true :: "parity ⇒ parity ⇒ bool" where
  "parity_eq_true = parity_vacuous"

lemma parity_eq_true_sound:
  assumes "parity_eq_true a b" and "i ∈ gamma_parity a" and "j ∈ gamma_parity b"
  shows "i = j"
  using assms unfolding parity_eq_true_def parity_vacuous_def by (cases a; cases b; auto)
```

`parity_eq_false` is where parity earns its keep: two values of disjoint parity classes — one `PEven`, the other `POdd` — can never be equal, regardless of how imprecise Sign or Interval are about the same pair (both may see fully overlapping sign/range information). This is the one query Parity answers strictly better than a vacuity check.

```
fun parity_eq_false :: "parity ⇒ parity ⇒ bool" where
    "parity_eq_false PEven POdd = True"
  | "parity_eq_false POdd PEven = True"
  | "parity_eq_false a b = (a = PBot ∨ b = PBot)"

lemma parity_eq_false_sound:
```

```
  assumes "parity_eq_false a b" and "i ∈ gamma_parity a" and "j ∈ gamma_parity b"
  shows "i ≠ j"
  using assms by (cases a; cases b; auto)
```

Goblint-style optional-Boolean queries

`parity_less`/`parity_eq` are the generic packaging of the four judgments above, so one interpretation both introduces them and discharges the query interface. Nothing about that packaging is backward-specific: it asks only for four judgments and their soundness, which is why parity can reuse it while having no `inv_less` for the derivations beside it to read off. The **defines** clause is what keeps the two queries ordinary top-level constants, with the code equation the check layer needs.

```
global_interpretation parity_numeric_queries:
  numeric_query_judgments
    parity_less_true parity_less_false parity_eq_true parity_eq_false
  defines parity_less = parity_numeric_queries.query_less
    and parity_eq = parity_numeric_queries.query_eq
  by unfold_locales
     (auto intro: parity_less_true_sound parity_eq_true_sound
           dest: parity_less_false_sound parity_eq_false_sound)


end
```

13.8 Parity_Sound

```
theory Parity_Sound
  imports
    Parity_Exec
begin
```

What a run's initial abstract state describes

A run starts with every declared global at zero and every local unconstrained. The one lemma here says `cinit_parity_st` describes exactly that: zero is even, so `PEven` covers a declared global precisely, and `PTop` covers an unconstrained local trivially. Choosing `PEven` rather than `PTop` for globals is what makes this a fact worth owning: it is the one place the zero-initialization of globals is turned into a parity.

Parity's specification, its concretization and the soundness of the one against the other are not restated here. Those are constants and theorems of `dg_domain_exec`, and an assembly reaches them by interpreting that locale from Parity's own commute lemmas in `Voblint_Analysis_Parity.Parity_Exec` — so a per-domain copy would only rename what the locale already proves.

```
lemma parity_cinit_gamma:
  "cinit_stores 𝒢 ⊆ default_st_gamma 𝒢 cinit_parity_st"
  by (auto simp: cinit_stores_def gamma_state_def default_st_gamma_initial)


end
```

13.9 Parity_Classify

theory Parity_Classify
  imports
    Parity_Exec
begin

Parity instance of the generic check-discharge interface

The check classifier is derived from Parity's bundle, exactly as in `Voblint_Analysis_Sign.Sign_Classify`: the Boolean recursion over `exp`, the three-way classification, and the node-indexed bridge to `checks_proven` come from the comparison queries `parity_less`/`parity_eq` and the evaluator, through the `parity_tf` interpretation in `Voblint_Analysis_Parity.Parity_Transfer`. Their facts stay under `parity_tf.check.`.

end

13.10 Parity_Analyses

```
theory Parity_Analyses
  imports
    Parity_Sound
    "Voblint_Nonrelational.Nonrel_Registration"
begin
```

Registering Parity at every context and update rule

Parity runs through the shared D/G pipeline at the unit context. The CLI runs it as a field of the combined state of `MCP_Analyses`, whose component and soundness the unit registration supplies. Each registration leaves the rule that merges a value side-effected into a global as a parameter `r`. The equation system, the solve, the result table and every soundness endpoint come from the interpreted locale; this theory only names the domain's own implementation and facts.

At the unit context

```
global_interpretation parity_rule:
    nonrel_unit_rule parity_ops cinit_parity_st r
  for r
  by (intro nonrel_unit_ruleI nonrel_registrationI
      parity_tf.sound_nonrelational_ops_axioms parity_cinit_gamma)

end
```

Chapter 14

Voblint_Analysis_Congruence

14.1 Congruence_Warrowing

```
theory Congruence_Warrowing
  imports "Voblint_Domain.Congruence_Lattice" "Voblint_Exec.Default_St_Reachability"
begin
```

Type-class warrowing for the TD warrowing solver

Congruence follows Sign's and Parity's own choice (`Sign_Lattice`, `Parity_Domain`): the vendored `widening`/`narrowing` classes require only the four upper/bracket laws below, not a termination proof, so a domain whose lattice is already well behaved gets a conservative instance rather than an invented acceleration mechanism. Widening is plain join; narrowing is the trivial left projection, which discharges `narrow_ge`/`narrow_le` immediately from `b <= a` without needing anything about congruence's own structure.

```
definition narrow_congruence_td :: "congruence => congruence => congruence" where
  "narrow_congruence_td a b = a"

instantiation congruence :: warrowing
begin

definition widen_congruence :: "congruence => congruence => congruence" where
  "((a :: congruence) ▽ b) = join_congruence a b"

definition narrow_congruence :: "congruence => congruence => congruence" where
  "((a :: congruence) △ b) = narrow_congruence_td a b"

instance proof intro_classes
  fix a b :: congruence
  show "a <= (a ▽ b)"
    unfolding widen_congruence_def by (rule join_congruence_ub1)
  show "b <= (a ▽ b)"
    unfolding widen_congruence_def by (rule join_congruence_ub2)
  show "b <= a ⟹ b <= (a △ b)"
    unfolding narrow_congruence_def narrow_congruence_td_def by simp
  show "b <= a ⟹ (a △ b) <= a"
    unfolding narrow_congruence_def narrow_congruence_td_def by simp
qed

end
```

Numeric domain instance

`numeric_domain` extends `executable_domain`, which includes the solver's `warrowing`, so the instance follows the widening and narrowing above.

```
instantiation congruence :: numeric_domain
begin

definition gamma_abs_congruence [simp]:
  "numeric_domain_class.gamma (a :: congruence) = gamma_congruence a"

definition is_empty_congruence [simp]:
  "is_empty (a :: congruence) = is_bottom_congruence a"

definition to_string_congruence [simp]:
  "to_string (a :: congruence) =
     (if is_top_congruence a then sym_top else string_of_congruence a)"

instance
proof intro_classes
  show "numeric_domain_class.gamma (bot :: congruence) = {}"
    unfolding bot_congruence_def by simp
next
  show "numeric_domain_class.gamma (top :: congruence) = UNIV"
    by simp
next
  fix a b :: congruence
  assume "a <= b"
  then show "numeric_domain_class.gamma a ⊆ numeric_domain_class.gamma b"
    unfolding less_eq_congruence_iff_gamma
    by simp
next
  fix a :: congruence
  show "is_empty a ⟷ numeric_domain_class.gamma a = {}"
    by (simp add: is_bottom_congruence_correct)
```

qed

end

end

14.2 Congruence_Arithmetic

```
theory Congruence_Arithmetic
  imports Congruence_Warrowing
    "Voblint_Nonrelational.Abstract_Arithmetic"
begin
```

Congruence arithmetic

Arithmetic follows the unbounded-integer fragment of Goblint's congruence domain. Every public result passes through the normalized carrier. Goblint raises when exactly one operand is bottom; the total HOL operations instead return bottom because no concrete arithmetic result exists for an empty operand.

```
fun plus_congruence_rep ::
    "congruence_rep => congruence_rep => congruence_rep"
where
  "plus_congruence_rep None _ = None"
| "plus_congruence_rep _ None = None"
| "plus_congruence_rep (Some (c1, m1)) (Some (c2, m2)) =
     normalize_congruence_rep (Some (c1 + c2, gcd m1 m2))"

fun minus_congruence_rep ::
    "congruence_rep => congruence_rep => congruence_rep"
where
  "minus_congruence_rep None _ = None"
| "minus_congruence_rep _ None = None"
| "minus_congruence_rep (Some (c1, m1)) (Some (c2, m2)) =
     normalize_congruence_rep (Some (c1 - c2, gcd m1 m2))"
```

Multiplication expands two classes into a constant plus three variable terms. The gcd of their coefficients contains every concrete product while retaining all congruence information justified by that expansion. The resulting class is a sound over-approximation; no exactness claim is needed.

```
fun times_congruence_rep ::
    "congruence_rep => congruence_rep => congruence_rep"
where
  "times_congruence_rep None _ = None"
| "times_congruence_rep _ None = None"
| "times_congruence_rep (Some (c1, m1)) (Some (c2, m2)) =
     normalize_congruence_rep
       (Some
         (c1 * c2,
          gcd (c1 * m2)
            (gcd (m1 * c2) (m1 * m2))))"

lemma normalized_plus_congruence_rep [simp]:
  "normalized_congruence_rep (plus_congruence_rep a b)"
  by (cases a; cases b;
      auto simp only: plus_congruence_rep.simps
        normalized_congruence_rep.simps
        normalized_normalize_congruence_rep
        split: prod.splits)

lemma normalized_minus_congruence_rep [simp]:
  "normalized_congruence_rep (minus_congruence_rep a b)"
  by (cases a; cases b;
      auto simp only: minus_congruence_rep.simps
        normalized_congruence_rep.simps
        normalized_normalize_congruence_rep
        split: prod.splits)

lemma normalized_times_congruence_rep [simp]:
  "normalized_congruence_rep (times_congruence_rep a b)"
  by (cases a; cases b;
      auto simp only: times_congruence_rep.simps
        normalized_congruence_rep.simps
        normalized_normalize_congruence_rep
        split: prod.splits)

instantiation congruence :: plus
begin

lift_definition plus_congruence ::
    "congruence => congruence => congruence"
  is plus_congruence_rep
  by simp
```

```
instance ..

end

instantiation congruence :: minus
begin

lift_definition minus_congruence ::
    "congruence => congruence => congruence"
  is minus_congruence_rep
  by simp

instance ..

end

instantiation congruence :: times
begin

lift_definition times_congruence ::
    "congruence => congruence => congruence"
  is times_congruence_rep
  by simp

instance ..

end
```

Semantic soundness

Each operation's soundness reduces to one divisibility fact about the two operand classes; `congruence_binop_sound` discharges the bottom operands and the representation plumbing once.

```
lemma congruence_binop_sound:
  assumes "i ∈ gamma_congruence a" and "j ∈ gamma_congruence b"
    and "⋀c1 m1 c2 m2. Rep_congruence a = Some (c1, m1) ⟹
           Rep_congruence b = Some (c2, m2) ⟹ m1 dvd i - c1 ⟹ m2 dvd j - c2 ⟹
           k ∈ gamma_congruence r"
  shows "k ∈ gamma_congruence r"
  using assms unfolding gamma_congruence_def
  by (cases rule: congruence_rep_cases[of a]; cases rule: congruence_rep_cases[of b]) auto

lemma congruence_plus_sound:
  assumes "i ∈ gamma_congruence a" and "j ∈ gamma_congruence b"
  shows "i + j ∈ gamma_congruence (a + b)"
  using assms
proof (rule congruence_binop_sound)
  fix c1 m1 c2 m2
  assume a: "Rep_congruence a = Some (c1, m1)" and b: "Rep_congruence b = Some (c2, m2)"
    and mi: "m1 dvd i - c1" and mj: "m2 dvd j - c2"
  have "gcd m1 m2 dvd (i - c1) + (j - c2)"
    by (rule dvd_add[OF dvd_trans[OF gcd_dvd1 mi] dvd_trans[OF gcd_dvd2 mj]])
  then have "gcd m1 m2 dvd (i + j) - (c1 + c2)" by (simp add: algebra_simps)
  then show "i + j ∈ gamma_congruence (a + b)"
    unfolding gamma_congruence_def plus_congruence.rep_eq a b
    by (simp only: plus_congruence_rep.simps gamma_normalize_congruence_rep
        gamma_congruence_rep.simps mem_Collect_eq)
qed

lemma congruence_minus_sound:
  assumes "i ∈ gamma_congruence a" and "j ∈ gamma_congruence b"
  shows "i - j ∈ gamma_congruence (a - b)"
  using assms
proof (rule congruence_binop_sound)
  fix c1 m1 c2 m2
  assume a: "Rep_congruence a = Some (c1, m1)" and b: "Rep_congruence b = Some (c2, m2)"
    and mi: "m1 dvd i - c1" and mj: "m2 dvd j - c2"
  have "gcd m1 m2 dvd (i - c1) - (j - c2)"
    by (rule dvd_diff[OF dvd_trans[OF gcd_dvd1 mi] dvd_trans[OF gcd_dvd2 mj]])
  then have "gcd m1 m2 dvd (i - j) - (c1 - c2)" by (simp add: algebra_simps)
  then show "i - j ∈ gamma_congruence (a - b)"
    unfolding gamma_congruence_def minus_congruence.rep_eq a b
    by (simp only: minus_congruence_rep.simps gamma_normalize_congruence_rep
        gamma_congruence_rep.simps mem_Collect_eq)
qed
```

```
lemma congruence_product_divisible:
  fixes c1 c2 m1 m2 i j :: int
  assumes mi: "m1 dvd i - c1"
      and mj: "m2 dvd j - c2"
  shows
    "gcd (c1 * m2) (gcd (m1 * c2) (m1 * m2))
       dvd i * j - c1 * c2"
proof -
  obtain ki where ki: "i - c1 = m1 * ki"
    using mi unfolding dvd_def by blast
  obtain kj where kj: "j - c2 = m2 * kj"
    using mj unfolding dvd_def by blast
  let ?g =
    "gcd (c1 * m2) (gcd (m1 * c2) (m1 * m2))"
  have d1: "?g dvd c1 * m2"
    by (rule gcd_dvd1)
  have outer:
    "?g dvd gcd (m1 * c2) (m1 * m2)"
    by (rule gcd_dvd2)
  have d2: "?g dvd m1 * c2"
    by (rule dvd_trans[OF outer gcd_dvd1])
  have d3: "?g dvd m1 * m2"
    by (rule dvd_trans[OF outer gcd_dvd2])
  have t1: "?g dvd c1 * (j - c2)"
    using dvd_mult2[OF d1, of kj] unfolding kj by (simp add: algebra_simps)
  have t2: "?g dvd c2 * (i - c1)"
    using dvd_mult2[OF d2, of ki] unfolding ki by (simp add: algebra_simps)
  have t3: "?g dvd (i - c1) * (j - c2)"
    using dvd_mult2[OF d3, of "ki * kj"]
    unfolding ki kj by (simp add: algebra_simps)
  have combined:
    "?g dvd c1 * (j - c2) + c2 * (i - c1) +
      (i - c1) * (j - c2)"
    by (rule dvd_add[OF dvd_add[OF t1 t2] t3])
  have product:
    "c1 * (j - c2) + c2 * (i - c1) +
       (i - c1) * (j - c2) =
     i * j - c1 * c2"
    by (simp add: algebra_simps)
  show ?thesis using combined unfolding product .
qed

lemma congruence_times_sound:
  assumes "i ∈ gamma_congruence a" and "j ∈ gamma_congruence b"
  shows "i * j ∈ gamma_congruence (a * b)"
  using assms
proof (rule congruence_binop_sound)
  fix c1 m1 c2 m2
  assume a: "Rep_congruence a = Some (c1, m1)" and b: "Rep_congruence b = Some (c2, m2)"
    and mi: "m1 dvd i - c1" and mj: "m2 dvd j - c2"
  from congruence_product_divisible[OF mi mj]
  show "i * j ∈ gamma_congruence (a * b)"
    unfolding gamma_congruence_def times_congruence.rep_eq a b
    by (simp only: times_congruence_rep.simps gamma_normalize_congruence_rep
        gamma_congruence_rep.simps mem_Collect_eq)
qed
```

Monotonicity

Addition, subtraction and multiplication are monotone. At the representation level an ordering `c + m`$\mathbb{Z}$ below `c' + m'`$\mathbb{Z}$ is divisibility of the modulus and the offset difference, so `plus_congruence_rep_mono_nonbottom` reduces to `gcd` monotonicity; the lifted `congruence_plus_mono` and its siblings split on bottom.

```
lemma plus_congruence_rep_mono_nonbottom:
  fixes c1 m1 c2 m2 d1 n1 d2 n2 :: int
  assumes m: "m2 dvd m1" "m2 dvd c1 - c2"
      and n: "n2 dvd n1" "n2 dvd d1 - d2"
  shows
    "congruence_le_rep
       (plus_congruence_rep (Some (c1, m1)) (Some (d1, n1)))
       (plus_congruence_rep (Some (c2, m2)) (Some (d2, n2)))"
  unfolding congruence_le_rep_iff
  by (simp only: plus_congruence_rep.simps gamma_normalize_congruence_rep
        gamma_congruence_rep.simps congruence_class_subset_iff)
     (use gcd_mono[OF m(1) n(1)]
```

```
        dvd_add[OF dvd_trans[OF gcd_dvd1 m(2)] dvd_trans[OF gcd_dvd2 n(2)]]
      in ‹simp add: algebra_simps›)

lemma plus_congruence_rep_mono:
  assumes "congruence_le_rep a1 a2"
      and "congruence_le_rep b1 b2"
  shows
    "congruence_le_rep
       (plus_congruence_rep a1 b1)
       (plus_congruence_rep a2 b2)"
  using assms
  by (cases a1; cases a2; cases b1; cases b2;
      auto split: prod.splits
        intro: plus_congruence_rep_mono_nonbottom)

lemma congruence_plus_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows "a1 + b1 <= a2 + (b2 :: congruence)"
  unfolding less_eq_congruence_def congruence_le_def
  apply (simp only: plus_congruence.rep_eq)
  by (rule plus_congruence_rep_mono)
     (use assms in
       ‹simp_all add: less_eq_congruence_def congruence_le_def›)

lemma minus_congruence_rep_mono_nonbottom:
  fixes c1 m1 c2 m2 d1 n1 d2 n2 :: int
  assumes m: "m2 dvd m1" "m2 dvd c1 - c2"
      and n: "n2 dvd n1" "n2 dvd d1 - d2"
  shows
    "congruence_le_rep
       (minus_congruence_rep (Some (c1, m1)) (Some (d1, n1)))
       (minus_congruence_rep (Some (c2, m2)) (Some (d2, n2)))"
  unfolding congruence_le_rep_iff
  by (simp only: minus_congruence_rep.simps gamma_normalize_congruence_rep
        gamma_congruence_rep.simps congruence_class_subset_iff)
     (use gcd_mono[OF m(1) n(1)]
        dvd_diff[OF dvd_trans[OF gcd_dvd1 m(2)] dvd_trans[OF gcd_dvd2 n(2)]]
      in ‹simp add: algebra_simps›)

lemma minus_congruence_rep_mono:
  assumes "congruence_le_rep a1 a2"
      and "congruence_le_rep b1 b2"
  shows
    "congruence_le_rep
       (minus_congruence_rep a1 b1)
       (minus_congruence_rep a2 b2)"
  using assms
  by (cases a1; cases a2; cases b1; cases b2;
      auto split: prod.splits
        intro: minus_congruence_rep_mono_nonbottom)

lemma congruence_minus_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows "a1 - b1 <= a2 - (b2 :: congruence)"
  unfolding less_eq_congruence_def congruence_le_def
  apply (simp only: minus_congruence.rep_eq)
  by (rule minus_congruence_rep_mono)
     (use assms in
       ‹simp_all add: less_eq_congruence_def congruence_le_def›)

lemma times_congruence_rep_mono_nonbottom:
  fixes c1 m1 c2 m2 d1 n1 d2 n2 :: int
  assumes m: "m2 dvd m1" "m2 dvd c1 - c2"
      and n: "n2 dvd n1" "n2 dvd d1 - d2"
  shows
    "congruence_le_rep
       (times_congruence_rep (Some (c1, m1)) (Some (d1, n1)))
       (times_congruence_rep (Some (c2, m2)) (Some (d2, n2)))"
proof -
  obtain km where m1: "m1 = m2 * km"
    using m(1) unfolding dvd_def by blast
  obtain kc where c1: "c1 = c2 + m2 * kc"
  proof -
    obtain kc where "c1 - c2 = m2 * kc"
      using m(2) unfolding dvd_def by blast
    then have "c1 = c2 + m2 * kc" by simp
```

```
    then show thesis by (rule that)
  qed
  obtain kn where n1: "n1 = n2 * kn"
    using n(1) unfolding dvd_def by blast
  obtain kd where d1: "d1 = d2 + n2 * kd"
  proof -
    obtain kd where "d1 - d2 = n2 * kd"
      using n(2) unfolding dvd_def by blast
    then have "d1 = d2 + n2 * kd" by simp
    then show thesis by (rule that)
  qed
  let ?coarse =
    "gcd (c2 * n2) (gcd (m2 * d2) (m2 * n2))"
  have coarse_c: "?coarse dvd c2 * n2"
    by (rule gcd_dvd1)
  have coarse_tail:
    "?coarse dvd gcd (m2 * d2) (m2 * n2)"
    by (rule gcd_dvd2)
  have coarse_d: "?coarse dvd m2 * d2"
    by (rule dvd_trans[OF coarse_tail gcd_dvd1])
  have coarse_mn: "?coarse dvd m2 * n2"
    by (rule dvd_trans[OF coarse_tail gcd_dvd2])
  have generator1: "?coarse dvd c1 * n1"
  proof -
    have first: "?coarse dvd (c2 * n2) * kn"
      by (rule dvd_mult2[OF coarse_c])
    have second: "?coarse dvd (m2 * n2) * (kc * kn)"
      by (rule dvd_mult2[OF coarse_mn])
    have sum:
      "?coarse dvd (c2 * n2) * kn +
        (m2 * n2) * (kc * kn)"
      by (rule dvd_add[OF first second])
    show ?thesis
      using sum unfolding c1 n1 by (simp add: algebra_simps)
  qed
  have generator2: "?coarse dvd m1 * d1"
  proof -
    have first: "?coarse dvd (m2 * d2) * km"
      by (rule dvd_mult2[OF coarse_d])
    have second: "?coarse dvd (m2 * n2) * (km * kd)"
      by (rule dvd_mult2[OF coarse_mn])
    have sum:
      "?coarse dvd (m2 * d2) * km +
        (m2 * n2) * (km * kd)"
      by (rule dvd_add[OF first second])
    show ?thesis
      using sum unfolding m1 d1 by (simp add: algebra_simps)
  qed
  have generator3: "?coarse dvd m1 * n1"
  proof -
    have "?coarse dvd (m2 * n2) * (km * kn)"
      by (rule dvd_mult2[OF coarse_mn])
    then show ?thesis
      unfolding m1 n1 by (simp add: algebra_simps)
  qed
  have modulus:
    "?coarse dvd
      gcd (c1 * n1) (gcd (m1 * d1) (m1 * n1))"
  proof (rule gcd_greatest)
    show "?coarse dvd c1 * n1" by (rule generator1)
    show "?coarse dvd gcd (m1 * d1) (m1 * n1)"
      by (rule gcd_greatest[OF generator2 generator3])
  qed
  have offset: "?coarse dvd c1 * d1 - c2 * d2"
    by (rule congruence_product_divisible[OF m(2) n(2)])
  show ?thesis
    unfolding congruence_le_rep_iff
    using modulus offset
    by (simp only: times_congruence_rep.simps
        gamma_normalize_congruence_rep
        gamma_congruence_rep.simps congruence_class_subset_iff)
qed

lemma times_congruence_rep_mono:
  assumes "congruence_le_rep a1 a2"
      and "congruence_le_rep b1 b2"
```

```
  shows
    "congruence_le_rep
       (times_congruence_rep a1 b1)
       (times_congruence_rep a2 b2)"
  using assms
  by (cases a1; cases a2; cases b1; cases b2;
      auto split: prod.splits
        intro: times_congruence_rep_mono_nonbottom)

lemma congruence_times_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows "a1 * b1 <= a2 * (b2 :: congruence)"
  unfolding less_eq_congruence_def congruence_le_def
  apply (simp only: times_congruence.rep_eq)
  by (rule times_congruence_rep_mono)
     (use assms in
       ‹simp_all add: less_eq_congruence_def congruence_le_def›)
```

Comparison and truthiness queries

Congruence classes carry no ordering information, so `congruence_lt` is always `None`. Equality and truthiness are decidable exactly when both sides are singleton classes (modulus `0`): `congruence_singleton` reads that singleton constant off the normalized representation.

```
definition congruence_singleton :: "congruence => int option" where
  "congruence_singleton a =
     (case Rep_congruence a of
        Some (c, m) => (if m = 0 then Some c else None)
      | None => None)"

lemma congruence_singleton_sound:
  assumes "congruence_singleton a = Some c" and "n ∈ gamma_congruence a"
  shows "n = c"
  using assms
  unfolding congruence_singleton_def gamma_congruence_def
  by (cases "Rep_congruence a" rule: option.exhaust) (auto split: if_splits)

lemma congruence_singleton_le:
  assumes "¬ is_empty (a1::congruence)" and "a1 ≤ a2"
      and "congruence_singleton a2 = Some c"
  shows "congruence_singleton a1 = Some c"
proof -
  have rep2: "Rep_congruence a2 = Some (c, 0)"
    using assms(3) unfolding congruence_singleton_def
    by (auto split: option.splits if_splits)
  have le: "congruence_le_rep (Rep_congruence a1) (Rep_congruence a2)"
    using assms(2) unfolding less_eq_congruence_def congruence_le_def .
  show ?thesis
  proof (cases "Rep_congruence a1")
    case None
    then have "a1 = bottom_congruence"
      by (simp only: Rep_congruence_inject[symmetric] bottom_congruence.rep_eq)
    with assms(1) show ?thesis
      by (simp add: is_bottom_congruence_def bot_congruence_def)
  next
    case (Some p)
    obtain c1 m1 where p: "p = (c1, m1)" by (cases p)
    with le Some rep2 have "m1 = 0 ∧ c1 = c" by simp
    with Some p show ?thesis unfolding congruence_singleton_def by simp
  qed
qed

definition congruence_lt :: "congruence => congruence => bool option" where
  "congruence_lt a b =
     (case (congruence_singleton a, congruence_singleton b) of
        (Some c1, Some c2) => Some (c1 < c2)
      | _ => None)"

definition congruence_eqb :: "congruence => congruence => bool option" where
  "congruence_eqb a b =
     (case (congruence_singleton a, congruence_singleton b) of
        (Some c1, Some c2) => Some (c1 = c2)
      | _ => None)"

definition congruence_tobool :: "congruence => bool option" where
  "congruence_tobool a =
```

```
    (case congruence_singleton a of Some c => Some (c ≠ 0) | None => None)"

lemma congruence_lt_sound:
  assumes "congruence_lt a b = Some c" and "i ∈ gamma_congruence a" and "j ∈ gamma_congruence b"
  shows "(i < j) = c"
proof -
  obtain c1 c2 where s1: "congruence_singleton a = Some c1" and s2:
    "congruence_singleton b = Some c2"
      and c_def: "c = (c1 < c2)"
    using assms(1) unfolding congruence_lt_def by (auto split: option.splits)
  have "i = c1" using congruence_singleton_sound[OF s1 assms(2)] .
  moreover have "j = c2" using congruence_singleton_sound[OF s2 assms(3)] .
  ultimately show ?thesis using c_def by simp
qed

lemma congruence_eqb_sound:
  assumes "congruence_eqb a b = Some c" and "i ∈ gamma_congruence a" and "j ∈ gamma_congruence b"
  shows "(i = j) = c"
proof -
  obtain c1 c2 where s1: "congruence_singleton a = Some c1" and s2:
    "congruence_singleton b = Some c2"
      and c_def: "c = (c1 = c2)"
    using assms(1) unfolding congruence_eqb_def by (auto split: option.splits)
  have "i = c1" using congruence_singleton_sound[OF s1 assms(2)] .
  moreover have "j = c2" using congruence_singleton_sound[OF s2 assms(3)] .
  ultimately show ?thesis using c_def by simp
qed

lemma congruence_tobool_sound:
  assumes "congruence_tobool a = Some c" and "i ∈ gamma_congruence a"
  shows "(i ≠ 0) = c"
proof -
  obtain c1 where s1: "congruence_singleton a = Some c1" and c_def: "c = (c1 ≠ 0)"
    using assms(1) unfolding congruence_tobool_def by (auto split: option.splits)
  have "i = c1" using congruence_singleton_sound[OF s1 assms(2)] .
  then show ?thesis using c_def by simp
qed

lemma congruence_lt_mono:
  assumes "¬ is_empty (a1::congruence)" and "¬ is_empty b1" and "a1 ≤ a2" and "b1 ≤ b2"
      and "congruence_lt a2 b2 = Some c"
  shows "congruence_lt a1 b1 = Some c"
proof -
  obtain c1 c2 where s1: "congruence_singleton a2 = Some c1" and s2:
    "congruence_singleton b2 = Some c2"
      and c_def: "c = (c1 < c2)"
    using assms(5) unfolding congruence_lt_def by (auto split: option.splits)
  have "congruence_singleton a1 = Some c1"
    using congruence_singleton_le[OF assms(1,3) s1] .
  moreover have "congruence_singleton b1 = Some c2"
    using congruence_singleton_le[OF assms(2,4) s2] .
  ultimately show ?thesis unfolding congruence_lt_def using c_def by simp
qed

lemma congruence_eqb_mono:
  assumes "¬ is_empty (a1::congruence)" and "¬ is_empty b1" and "a1 ≤ a2" and "b1 ≤ b2"
      and "congruence_eqb a2 b2 = Some c"
  shows "congruence_eqb a1 b1 = Some c"
proof -
  obtain c1 c2 where s1: "congruence_singleton a2 = Some c1" and s2:
    "congruence_singleton b2 = Some c2"
      and c_def: "c = (c1 = c2)"
    using assms(5) unfolding congruence_eqb_def by (auto split: option.splits)
  have "congruence_singleton a1 = Some c1"
    using congruence_singleton_le[OF assms(1,3) s1] .
  moreover have "congruence_singleton b1 = Some c2"
    using congruence_singleton_le[OF assms(2,4) s2] .
  ultimately show ?thesis unfolding congruence_eqb_def using c_def by simp
qed

lemma congruence_tobool_mono:
  assumes "¬ is_empty (a1::congruence)" and "a1 ≤ a2" and "congruence_tobool a2 = Some c"
  shows "congruence_tobool a1 = Some c"
proof -
  obtain c1 where s1: "congruence_singleton a2 = Some c1" and c_def: "c = (c1 ≠ 0)"
    using assms(3) unfolding congruence_tobool_def by (auto split: option.splits)
```

```
  have "congruence_singleton a1 = Some c1"
    using congruence_singleton_le[OF assms(1,2) s1] .
  then show ?thesis unfolding congruence_tobool_def using c_def by simp
qed


definition congruence_exact_binop ::
    "(int ⇒ int ⇒ int) ⇒ (congruence ⇒ congruence ⇒ congruence) ⇒
      congruence ⇒ congruence ⇒ congruence" where
  "congruence_exact_binop f fallback a b =
    (if is_empty a ∨ is_empty b then bot
     else case (congruence_singleton a, congruence_singleton b) of
       (Some x, Some y) ⇒ congruence_of_int (f x y)
     | _ ⇒ fallback a b)"

lemma congruence_exact_binop_sound:
  assumes "i ∈ gamma_congruence a" "j ∈ gamma_congruence b"
    and sound: "⋀a b i j. i ∈ gamma_congruence a ⟹ j ∈ gamma_congruence b ⟹
      f i j ∈ gamma_congruence (fallback a b)"
  shows "f i j ∈ gamma_congruence (congruence_exact_binop f fallback a b)"
  using assms congruence_singleton_sound
  unfolding congruence_exact_binop_def
  by (auto simp: is_bottom_congruence_correct congruence_of_int_def
      split: option.splits dest: congruence_singleton_sound)

lemma congruence_exact_binop_mono:
  assumes "a1 ≤ a2" "b1 ≤ b2"
    and sound: "⋀a b i j. i ∈ gamma_congruence a ⟹ j ∈ gamma_congruence b ⟹
      f i j ∈ gamma_congruence (fallback a b)"
    and mono: "⋀a1 a2 b1 b2. a1 ≤ a2 ⟹ b1 ≤ b2 ⟹
      fallback a1 b1 ≤ fallback a2 b2"
  shows "congruence_exact_binop f fallback a1 b1 ≤
    congruence_exact_binop f fallback a2 b2"
proof (cases "is_empty a1 ∨ is_empty b1")
  case True
  then show ?thesis by (simp add: congruence_exact_binop_def)
next
  case False
  have ne: "¬ is_empty a1" "¬ is_empty b1" "¬ is_empty a2" "¬ is_empty b2"
    using False is_empty_antimono assms(1,2) by blast+
  have singleton_le:
    "⋀a x. congruence_singleton a = Some x ⟹
      gamma_congruence a = {x}"
    unfolding congruence_singleton_def gamma_congruence_def
    by (auto split: option.splits if_splits)
  show ?thesis
    unfolding congruence_exact_binop_def
    using ne congruence_singleton_le[OF ne(1) assms(1)]
      congruence_singleton_le[OF ne(2) assms(2)]
      assms(1,2) mono[OF assms(1,2)] sound singleton_le
    by (auto simp: less_eq_congruence_iff_gamma congruence_of_int_def
        split: option.splits)
qed


lemma gamma_congruence_exact_division:
  assumes nz: "d ≠ 0" and dc: "d dvd c" and dm: "d dvd m"
  shows "gamma_congruence (mk_congruence (c div d) (m div d)) =
    (λn. c_div n d) ` {n. m dvd n - c}"
proof -
  obtain c' m' where c: "c = d * c'" and m: "m = d * m'"
    using dc dm by (auto simp: dvd_def)
  have cancel: "⋀k. c_div (d * k) d = k"
    using nz by (simp add: c_div_exact)
  have quotients: "c div d = c'" "m div d = m'"
    using nz by (simp_all add: c m)
  show ?thesis
  proof (rule set_eqI, rule iffI)
    fix q
    assume "q ∈ gamma_congruence (mk_congruence (c div d) (m div d))"
    then obtain k where q: "q - c' = m' * k"
      unfolding quotients by (auto simp: dvd_def)
    have member: "d * q ∈ {n. m dvd n - c}"
      unfolding c m using q by (auto simp: dvd_def algebra_simps)
    show "q ∈ (λn. c_div n d) ` {n. m dvd n - c}"
      using imageI[OF member, of "λn. c_div n d"] by (simp add: cancel)
```

```
  next
    fix q
    assume "q ∈ (λn. c_div n d) ‘ {n. m dvd n - c}"
    then obtain n k where q: "q = c_div n d" and n: "n - c = m * k"
      by (auto simp: dvd_def)
    have n_eq: "n = d * (c' + m' * k)"
      using n unfolding c m by (simp add: algebra_simps)
    show "q ∈ gamma_congruence (mk_congruence (c div d) (m div d))"
      unfolding quotients q n_eq by (simp add: cancel)
  qed
qed

definition congruence_divides :: "int ⇒ congruence ⇒ bool" where
  "congruence_divides d a =
    (case Rep_congruence a of None ⇒ True | Some (c, m) ⇒ d dvd c ∧ d dvd m)"

lemma congruence_divides_iff:
  "congruence_divides d a ⟷ (∀n ∈ gamma_congruence a. d dvd n)"
proof -
  have classes: "⋀c m. (∀n. m dvd n - c ⟶ d dvd n) ⟷ d dvd c ∧ d dvd m"
    using congruence_class_subset_iff[of _ _ d 0]
    by (auto simp: subset_iff)
  show ?thesis
    unfolding congruence_divides_def gamma_congruence_def
    by (auto simp: classes split: option.splits)
qed

definition congruence_div_const :: "congruence ⇒ int ⇒ congruence" where
  "congruence_div_const a d =
    (case Rep_congruence a of None ⇒ bot
     | Some (c, m) ⇒
       if d = 0 then congruence_of_int 0
       else if congruence_divides d a then mk_congruence (c div d) (m div d)
       else top)"

lemma gamma_congruence_div_const:
  assumes "d ≠ 0" "congruence_divides d a"
  shows "gamma_congruence (congruence_div_const a d) =
    (λn. c_div n d) ‘ gamma_congruence a"
proof (cases "Rep_congruence a")
  case None
  then have "gamma_congruence a = {}" by (simp add: gamma_congruence_def)
  then show ?thesis by (simp add: congruence_div_const_def None bot_congruence_def)
next
  case (Some p)
  obtain c m where p: "p = (c, m)" by (cases p)
  have divisibility: "d dvd c" "d dvd m"
    using assms(2) unfolding congruence_divides_def Some p by simp_all
  have gamma: "gamma_congruence a = {n. m dvd n - c}"
    by (simp add: gamma_congruence_def Some p)
  show ?thesis
    using gamma_congruence_exact_division[OF assms(1) divisibility]
    by (simp add: congruence_div_const_def Some p assms gamma)
qed

lemma gamma_congruence_div_const_cases:
  "gamma_congruence (congruence_div_const a d) =
    (if is_empty a then {}
     else if d = 0 then {0}
     else if congruence_divides d a then (λn. c_div n d) ‘ gamma_congruence a
     else UNIV)"
proof (cases "is_empty a")
  case True
  then have rep: "Rep_congruence a = None"
    by (simp add: is_bottom_congruence_def bot_congruence_def)
  show ?thesis using True
    by (simp add: congruence_div_const_def rep bot_congruence_def
        is_bottom_congruence_correct gamma_congruence_def)
next
  case False
  then obtain c m where rep: "Rep_congruence a = Some (c, m)"
    by (cases "Rep_congruence a") (auto simp: is_bottom_congruence_correct gamma_congruence_def)
  show ?thesis
    using gamma_congruence_div_const[of d a]
    by (cases "d = 0"; cases "congruence_divides d a")
       (use False in ‹simp_all add: congruence_div_const_def rep congruence_of_int_def›)
```

```
qed

lemma congruence_div_const_sound:
  assumes "i ∈ gamma_congruence a"
  shows "c_div i d ∈ gamma_congruence (congruence_div_const a d)"
  using assms
  by (auto simp: gamma_congruence_div_const_cases is_bottom_congruence_correct)

lemma congruence_div_const_mono:
  assumes "a1 ≤ a2"
  shows "congruence_div_const a1 d ≤ congruence_div_const a2 d"
  using assms
  by (auto simp: less_eq_congruence_iff_gamma gamma_congruence_div_const_cases
      is_bottom_congruence_correct congruence_divides_iff)

definition congruence_div_fallback :: "congruence ⇒ congruence ⇒ congruence" where
  "congruence_div_fallback a b =
    (if is_empty a ∨ is_empty b then bot
     else case congruence_singleton b of
       Some c ⇒ congruence_div_const a c
     | None ⇒ top)"

lemma congruence_div_fallback_sound:
  assumes "i ∈ gamma_congruence a" "j ∈ gamma_congruence b"
  shows "c_div i j ∈ gamma_congruence (congruence_div_fallback a b)"
  using assms congruence_singleton_sound[OF _ assms(2)]

  unfolding congruence_div_fallback_def
  by (auto simp: is_bottom_congruence_correct split: option.splits
      intro: congruence_div_const_sound)

lemma congruence_div_fallback_mono:
  assumes "a1 ≤ a2" "b1 ≤ b2"
  shows "congruence_div_fallback a1 b1 ≤ congruence_div_fallback a2 b2"
proof (cases "is_empty a1 ∨ is_empty b1")
  case True
  then show ?thesis by (simp add: congruence_div_fallback_def)
next
  case False
  have ne: "¬ is_empty a1" "¬ is_empty b1" "¬ is_empty a2" "¬ is_empty b2"
    using False is_empty_antimono assms by blast+
  show ?thesis unfolding congruence_div_fallback_def
    using ne assms congruence_singleton_le[OF ne(2) assms(2)]
    by (auto split: option.splits intro: congruence_div_const_mono)
qed

definition congruence_div :: "congruence ⇒ congruence ⇒ congruence" where
  "congruence_div = congruence_exact_binop c_div congruence_div_fallback"

definition congruence_mod :: "congruence ⇒ congruence ⇒ congruence" where
  "congruence_mod = congruence_exact_binop c_mod (λa b. a - top * b)"

lemma congruence_mod_fallback_sound:
  assumes "i ∈ gamma_congruence a" "j ∈ gamma_congruence b"
  shows "c_mod i j ∈ gamma_congruence (a - top * b)"
proof -
  have "i - c_div i j * j ∈ gamma_congruence (a - top * b)"
    by (intro congruence_minus_sound assms congruence_times_sound) simp_all
  then show ?thesis by (cases "j = 0") (simp_all add: c_mod_def)
qed

lemma congruence_div_sound:
  "i ∈ gamma_congruence a ⟹ j ∈ gamma_congruence b ⟹
    c_div i j ∈ gamma_congruence (congruence_div a b)"
  unfolding congruence_div_def
  by (rule congruence_exact_binop_sound) (auto intro: congruence_div_fallback_sound)

lemma congruence_mod_sound:
  "i ∈ gamma_congruence a ⟹ j ∈ gamma_congruence b ⟹
    c_mod i j ∈ gamma_congruence (congruence_mod a b)"
  unfolding congruence_mod_def
  by (rule congruence_exact_binop_sound) (auto intro: congruence_mod_fallback_sound)

lemma congruence_div_mono:
  "a1 ≤ a2 ⟹ b1 ≤ b2 ⟹ congruence_div a1 b1 ≤ congruence_div a2 b2"
  unfolding congruence_div_def
```

```
  by (rule congruence_exact_binop_mono)
     (auto intro: congruence_div_fallback_sound congruence_div_fallback_mono)

lemma congruence_mod_mono:
  "a1 ≤ a2 ⟹ b1 ≤ b2 ⟹ congruence_mod a1 b1 ≤ congruence_mod a2 b2"
  unfolding congruence_mod_def
  by (rule congruence_exact_binop_mono)
     (auto intro: congruence_mod_fallback_sound congruence_minus_mono congruence_times_mono)
```

Abstract expression evaluation

`aval_congruence` evaluates an expression bottom-up with the congruence operations above. A comparison is `bot` when either operand is empty; otherwise it is the literal its three-valued query decides, or the join of `0` and `1` on `None`. These are the equations `sound_arith_ops` expects of an evaluator.

```
fun aval_congruence ::
    "exp => (vname => congruence) => congruence"
where
  "aval_congruence (N n) sigma = congruence_of_int n"
| "aval_congruence (V x) sigma = sigma x"
| "aval_congruence (Plus e1 e2) sigma =
     aval_congruence e1 sigma + aval_congruence e2 sigma"
| "aval_congruence (Minus e1 e2) sigma =
     aval_congruence e1 sigma - aval_congruence e2 sigma"
| "aval_congruence (Times e1 e2) sigma =
     aval_congruence e1 sigma * aval_congruence e2 sigma"
  | "aval_congruence (Div e1 e2) sigma = congruence_div (aval_congruence e1 sigma) (aval_congruence e2 sigma)"
  | "aval_congruence (Mod e1 e2) sigma = congruence_mod (aval_congruence e1 sigma) (aval_congruence e2 sigma)"
| "aval_congruence (Less e1 e2) sigma =
     (if is_empty (aval_congruence e1 sigma) ∨ is_empty (aval_congruence e2 sigma) then bot
      else of_bool_option congruence_of_int
             (congruence_lt (aval_congruence e1 sigma) (aval_congruence e2 sigma)))"
| "aval_congruence (LessEq e1 e2) sigma =
     (if is_empty (aval_congruence e1 sigma) ∨ is_empty (aval_congruence e2 sigma) then bot
      else of_bool_option congruence_of_int
             (map_option HOL.Not
                (congruence_lt (aval_congruence e2 sigma) (aval_congruence e1 sigma))))"
| "aval_congruence (Greater e1 e2) sigma =
     (if is_empty (aval_congruence e1 sigma) ∨ is_empty (aval_congruence e2 sigma) then bot
      else of_bool_option congruence_of_int
             (congruence_lt (aval_congruence e2 sigma) (aval_congruence e1 sigma)))"
| "aval_congruence (GreaterEq e1 e2) sigma =
     (if is_empty (aval_congruence e1 sigma) ∨ is_empty (aval_congruence e2 sigma) then bot
      else of_bool_option congruence_of_int
             (map_option HOL.Not
                (congruence_lt (aval_congruence e1 sigma) (aval_congruence e2 sigma))))"
| "aval_congruence (NotEq e1 e2) sigma =
     (if is_empty (aval_congruence e1 sigma) ∨ is_empty (aval_congruence e2 sigma) then bot
      else of_bool_option congruence_of_int
             (map_option HOL.Not
                (congruence_eqb (aval_congruence e1 sigma) (aval_congruence e2 sigma))))"
| "aval_congruence (exp.Eq e1 e2) sigma =
     (if is_empty (aval_congruence e1 sigma) ∨ is_empty (aval_congruence e2 sigma) then bot
      else of_bool_option congruence_of_int
             (congruence_eqb (aval_congruence e1 sigma) (aval_congruence e2 sigma)))"
| "aval_congruence (exp.Not e) sigma =
     (if is_empty (aval_congruence e sigma) then bot
      else of_bool_option congruence_of_int
             (map_option HOL.Not (congruence_tobool (aval_congruence e sigma))))"
| "aval_congruence (And e1 e2) sigma =
     (if is_empty (aval_congruence e1 sigma) ∨ is_empty (aval_congruence e2 sigma) then bot
      else of_bool_option congruence_of_int
             (and_opt (congruence_tobool (aval_congruence e1 sigma))
                (congruence_tobool (aval_congruence e2 sigma))))"
| "aval_congruence (Or e1 e2) sigma =
     (if is_empty (aval_congruence e1 sigma) ∨ is_empty (aval_congruence e2 sigma) then bot
      else of_bool_option congruence_of_int
             (or_opt (congruence_tobool (aval_congruence e1 sigma))
                (congruence_tobool (aval_congruence e2 sigma))))"

interpretation congruence_arith: mono_arith_ops
    aval_congruence congruence_of_int "(+)" "(-)" "(*)" congruence_div congruence_mod
    congruence_lt congruence_eqb congruence_tobool
  apply unfold_locales
  apply (simp_all add: congruence_plus_sound congruence_minus_sound congruence_times_sound
    congruence_div_sound congruence_mod_sound
```

```
                      congruence_plus_mono congruence_minus_mono congruence_times_mono
                        congruence_div_mono congruence_mod_mono
                      congruence_lt_sound congruence_eqb_sound
                      congruence_tobool_sound[unfolded truthy_def])
  apply (blast intro: congruence_lt_mono[unfolded is_empty_congruence]
                     congruence_eqb_mono[unfolded is_empty_congruence]
                     congruence_tobool_mono[unfolded is_empty_congruence])+
  done

end
```

14.3 Congruence_Backward

```
theory Congruence_Backward
  imports
    Congruence_Arithmetic
    "Voblint_Nonrelational.Exec_Backward"
    "HOL-Computational_Algebra.Euclidean_Algorithm"
begin
```

Multiplication preimages

For a known factor k, the preimage of c modulo m solves the linear congruence k*x = c modulo m. Dividing by gcd(k,m) and using the executable Bezout coefficient yields the complete solution class.

```
fun preimage_times_const_rep ::
    "congruence_rep => int => congruence_rep"
where
  "preimage_times_const_rep None k = None"
| "preimage_times_const_rep (Some (c, m)) k =
     normalize_congruence_rep
       (if m = 0 then
          if k = 0 then
            if c = 0 then Some (0, 1) else None
          else if k dvd c then Some (c div k, 0) else None
        else
          let g = gcd k m
          in if g dvd c then
               let s = fst (bezout_coefficients k m)
               in Some ((c div g) * s, m div g)
             else None)"

lemma normalized_preimage_times_const_rep [simp]:
  "normalized_congruence_rep (preimage_times_const_rep r k)"
  by (cases r)
     (auto simp: split_def split: if_splits prod.splits)

lift_definition preimage_times_const ::
    "congruence => int => congruence"
  is preimage_times_const_rep
  by simp

lemma bezout_linear_congruence:
  assumes g_nonzero: "gcd k m ~= 0"
      and compatible: "gcd k m dvd c"
  shows
    "m div gcd k m dvd
       x - (c div gcd k m) * fst (bezout_coefficients k m)
     ⟷
     m dvd k * x - c"
proof -
  let ?g = "gcd k m"
  let ?s = "fst (bezout_coefficients k m)"
  let ?t = "snd (bezout_coefficients k m)"
  let ?q = "c div ?g"
  let ?kp = "k div ?g"
  let ?mp = "m div ?g"
  have gk: "?g dvd k"
    by (rule gcd_dvd1)
  have gm: "?g dvd m"
    by (rule gcd_dvd2)
  have k_eq: "?g * ?kp = k"
    using dvd_mult_div_cancel[OF gk]
    by (simp add: mult.commute)
  have m_eq: "?g * ?mp = m"
    using dvd_mult_div_cancel[OF gm]
    by (simp add: mult.commute)
  have c_eq: "?g * ?q = c"
    using dvd_mult_div_cancel[OF compatible]
    by (simp add: mult.commute)
  have bezout:
    "?s * k + ?t * m = ?g"
    by (rule bezout_coefficients_fst_snd)
  have scaled:
    "?g * (?s * ?kp + ?t * ?mp) = ?g"
  proof -
    have
```

```
    "?g * (?s * ?kp + ?t * ?mp) =
     ?s * (?g * ?kp) + ?t * (?g * ?mp)"
    by (simp only: ring_distribs mult.assoc mult.commute
        mult.left_commute)
  also have "... = ?s * k + ?t * m"
    by (simp only: k_eq m_eq)
  also have "... = ?g"
    by (rule bezout)
  finally show ?thesis .
qed
have scaled':
  "?g * (?s * ?kp + ?t * ?mp) = ?g * 1"
  using scaled by simp
have unit: "?s * ?kp + ?t * ?mp = 1"
  using scaled'
  by (simp only: mult_left_cancel[OF g_nonzero])
show ?thesis
proof
  assume solution: "?mp dvd x - ?q * ?s"
  obtain u where solution_eq:
    "x - ?q * ?s = ?mp * u"
    using solution unfolding dvd_def by blast
  have q_unit:
    "?q * (?s * ?kp + ?t * ?mp) = ?q"
    using unit by simp
  have reduced:
    "?kp * x - ?q = ?mp * (?kp * u + ?q * (- ?t))"
    using solution_eq q_unit
    by (simp add: algebra_simps)
  show "m dvd k * x - c"
    unfolding dvd_def
  proof (rule exI[of _ "?kp * u + ?q * (- ?t)"])
    have scaled_reduced:
      "?g * (?kp * x - ?q) =
       ?g * (?mp * (?kp * u + ?q * (- ?t)))"
      by (rule arg_cong[OF reduced])
    show
      "k * x - c =
       m * (?kp * u + ?q * (- ?t))"
    proof -
      have "k * x - c = ?g * (?kp * x - ?q)"
        using k_eq c_eq by (simp add: algebra_simps)
      also have "... =
        ?g * (?mp * (?kp * u + ?q * (- ?t)))"
        by (rule scaled_reduced)
      also have "... =
        (?g * ?mp) * (?kp * u + ?q * (- ?t))"
        by (simp only: mult.assoc)
      also have "... = m * (?kp * u + ?q * (- ?t))"
        by (simp only: m_eq)
      finally show ?thesis .
    qed
  qed
next
  assume target: "m dvd k * x - c"
  obtain u where target_eq:
    "k * x - c = m * u"
    using target unfolding dvd_def by blast
  have scaled_reduced:
    "?g * (?kp * x - ?q) = ?g * (?mp * u)"
    using target_eq k_eq m_eq c_eq
    by (simp add: algebra_simps)
  have reduced: "?kp * x - ?q = ?mp * u"
    using scaled_reduced
    by (simp only: mult_left_cancel[OF g_nonzero])
  show "?mp dvd x - ?q * ?s"
    unfolding dvd_def
  proof (rule exI[of _ "?s * u + ?t * x"])
    have scaled_solution:
      "?s * (?kp * x - ?q) = ?s * (?mp * u)"
      using reduced by simp
    have unit_x:
      "x * (?s * ?kp + ?t * ?mp) = x"
      using unit by simp
    show
      "x - ?q * ?s = ?mp * (?s * u + ?t * x)"
```

```
        using scaled_solution unit_x
        by (simp add: algebra_simps)
    qed
  qed
qed

lemma gamma_preimage_times_const_rep:
  assumes "normalized_congruence_rep r"
  shows
    "gamma_congruence_rep (preimage_times_const_rep r k) =
     {x. k * x : gamma_congruence_rep r}"
proof (cases r)
  case None
  then show ?thesis by simp
next
  case (Some cm)
  note r_some = Some
  obtain c m where cm: "cm = (c, m)"
    by (cases cm)
  show ?thesis
  proof (cases "m = 0")
    case m_zero: True
    show ?thesis
    proof (cases "k = 0")
      case k_zero: True
      show ?thesis
      proof (cases "c = 0")
        case True
        with r_some cm m_zero k_zero show ?thesis by simp
      next
        case False
        with r_some cm m_zero k_zero show ?thesis by simp
      qed
    next
      case k_nonzero: False
      show ?thesis
      proof (cases "k dvd c")
        case compatible: True
        have quotient: "k * (c div k) = c"
          by (rule dvd_mult_div_cancel[OF compatible])
        have solution_iff:
          "k * x = c ⟷ x = c div k" for x
        proof -
          have
            "(k * x = c) =
             (k * x = k * (c div k))"
            using quotient by simp
          also have "... = (x = c div k)"
            by (rule mult_left_cancel[OF k_nonzero])
          finally show ?thesis .
        qed
        with r_some cm m_zero k_nonzero compatible
        show ?thesis
          by simp
      next
        case incompatible: False
        have no_solution: "k * x ~= c" for x
        proof
          assume "k * x = c"
          then have "k dvd c"
            unfolding dvd_def by blast
          with incompatible show False by contradiction
        qed
        with r_some cm m_zero k_nonzero incompatible
        show ?thesis
          by simp
      qed
    qed
  next
    case m_nonzero: False
    let ?g = "gcd k m"
    let ?q = "c div ?g"
    let ?s = "fst (bezout_coefficients k m)"
    have g_nonzero: "?g ~= 0"
      using m_nonzero by simp
    show ?thesis
```

```
    proof (cases "?g dvd c")
      case incompatible: False
      have no_solution: "~ m dvd k * x - c" for x
      proof
        assume target: "m dvd k * x - c"
        have g_product: "?g dvd k * x"
          by (rule dvd_mult2[OF gcd_dvd1])
        have g_target: "?g dvd k * x - c"
          using gcd_dvd2 target by (rule dvd_trans)
        have "?g dvd k * x - (k * x - c)"
          by (rule dvd_diff[OF g_product g_target])
        then have "?g dvd c"
          by simp
        with incompatible show False by contradiction
      qed
      with r_some cm m_nonzero incompatible
      show ?thesis by simp
    next
      case compatible: True
      have solution_iff:
        "m div ?g dvd x - ?q * ?s ⟷
         m dvd k * x - c" for x
        by (rule bezout_linear_congruence[OF g_nonzero compatible])
      have normalized_gamma:
        "gamma_congruence_rep
           (normalize_congruence_rep
             (Some (?q * ?s, m div ?g))) =
         {x. m div ?g dvd x - ?q * ?s}"
        by (simp only: gamma_normalize_congruence_rep
            gamma_congruence_rep.simps)
      from r_some cm m_nonzero compatible normalized_gamma
        solution_iff
      show ?thesis
        by (simp only: preimage_times_const_rep.simps if_False
            if_True Let_def gamma_congruence_rep.simps
            Set.set_eq_iff mem_Collect_eq;
            simp)
    qed
  qed
qed

lemma gamma_preimage_times_const [simp]:
  "gamma_congruence (preimage_times_const r k) =
   {x. k * x : gamma_congruence r}"
proof -
  have norm_r:
    "normalized_congruence_rep (Rep_congruence r)"
    by simp
  show ?thesis
    using gamma_preimage_times_const_rep[OF norm_r, of k]
    by (simp add: gamma_congruence_def preimage_times_const.rep_eq)
qed

lemma preimage_times_const_mono:
  assumes "r1 <= r2"
  shows "preimage_times_const r1 k <= preimage_times_const r2 k"
  using assms
  unfolding less_eq_congruence_iff_gamma
  by auto

fun inverse_times_candidate_rep ::
    "congruence_rep => congruence_rep => congruence_rep"
where
  "inverse_times_candidate_rep None factor = None"
| "inverse_times_candidate_rep r None = None"
| "inverse_times_candidate_rep
     (Some (c, m)) (Some (k, n)) =
     (if n = 0
      then preimage_times_const_rep (Some (c, m)) k
      else Some (0, 1))"

lemma normalized_inverse_times_candidate_rep [simp]:
  assumes "normalized_congruence_rep r"
      and "normalized_congruence_rep factor"
  shows
    "normalized_congruence_rep
```

```
      (inverse_times_candidate_rep r factor)"
  using assms
  by (cases r; cases factor)
     (auto split: prod.splits if_splits)

lift_definition inverse_times_candidate ::
    "congruence => congruence => congruence"
  is inverse_times_candidate_rep
  by simp

lemma inverse_times_candidate_sound:
  assumes "z : gamma_congruence r"
      and "y : gamma_congruence factor"
      and "x * y = z"
  shows "x : gamma_congruence (inverse_times_candidate r factor)"
proof (cases "Rep_congruence r")
  case None
  with assms(1) show ?thesis
    unfolding gamma_congruence_def by simp
next
  case (Some rp)
  note r_some = Some
  obtain c m where rp: "rp = (c, m)"
    by (cases rp)
  show ?thesis
  proof (cases "Rep_congruence factor")
    case None
    with assms(2) show ?thesis
      unfolding gamma_congruence_def by simp
  next
    case (Some fp)
    note factor_some = Some
    obtain k n where fp: "fp = (k, n)"
      by (cases fp)
    show ?thesis
    proof (cases "n = 0")
      case singleton: True
      have y_eq: "y = k"
        using assms(2) factor_some fp singleton
        unfolding gamma_congruence_def by simp
      have x_preimage:
        "x : gamma_congruence (preimage_times_const r k)"
        using assms(1,3) y_eq
        by (simp add: mult.commute)
      from x_preimage r_some factor_some rp fp singleton
      show ?thesis
        unfolding gamma_congruence_def
        by (simp add: inverse_times_candidate.rep_eq
            preimage_times_const.rep_eq)
    next
      case nonsingleton: False
      with r_some factor_some rp fp show ?thesis
        unfolding gamma_congruence_def
        by (simp add: inverse_times_candidate.rep_eq)
    qed
  qed
qed

lemma inverse_times_candidate_mono:
  assumes "r1 <= r2" and "factor1 <= factor2"
  shows
    "inverse_times_candidate r1 factor1 <=
     inverse_times_candidate r2 factor2"
proof -
  have r_subset:
    "gamma_congruence r1 ⊆ gamma_congruence r2"
    using assms(1)
    unfolding less_eq_congruence_iff_gamma .
  have factor_subset:
    "gamma_congruence factor1 ⊆ gamma_congruence factor2"
    using assms(2)
    unfolding less_eq_congruence_iff_gamma .
  show ?thesis
  proof (cases "Rep_congruence r1")
    case None
    then show ?thesis
```

```
    unfolding less_eq_congruence_iff_gamma
      gamma_congruence_def
    by (simp add: inverse_times_candidate.rep_eq)
next
  case (Some rp1)
  note r1_some = Some
  obtain c1 m1 where rp1: "rp1 = (c1, m1)"
    by (cases rp1)
  have c1_member: "c1 : gamma_congruence r1"
    using r1_some rp1
    by (simp add: gamma_congruence_def)
  show ?thesis
  proof (cases "Rep_congruence factor1")
    case None
    with r1_some rp1 show ?thesis
      unfolding less_eq_congruence_iff_gamma
        gamma_congruence_def
      by (simp add: inverse_times_candidate.rep_eq)
  next
    case (Some fp1)
    note factor1_some = Some
    obtain k1 n1 where fp1: "fp1 = (k1, n1)"
      by (cases fp1)
    have k1_member: "k1 : gamma_congruence factor1"
      using factor1_some fp1
      by (simp add: gamma_congruence_def)
    have shifted_member:
      "k1 + n1 : gamma_congruence factor1"
      using factor1_some fp1
      by (simp add: gamma_congruence_def)
    show ?thesis
    proof (cases "Rep_congruence r2")
      case None
      have "c1 : gamma_congruence r2"
        by (rule subsetD[OF r_subset c1_member])
      with None show ?thesis
        unfolding gamma_congruence_def by simp
    next
      case (Some rp2)
      note r2_some = Some
      obtain c2 m2 where rp2: "rp2 = (c2, m2)"
        by (cases rp2)
      show ?thesis
      proof (cases "Rep_congruence factor2")
        case None
        have "k1 : gamma_congruence factor2"
          by (rule subsetD[OF factor_subset k1_member])
        with None show ?thesis
          unfolding gamma_congruence_def by simp
      next
        case (Some fp2)
        note factor2_some = Some
        obtain k2 n2 where fp2: "fp2 = (k2, n2)"
          by (cases fp2)
        show ?thesis
        proof (cases "n2 = 0")
          case factor2_singleton: True
          have k1_in_factor2:
            "k1 : gamma_congruence factor2"
            by (rule subsetD[OF factor_subset k1_member])
          have shifted_in_factor2:
            "k1 + n1 : gamma_congruence factor2"
            by (rule subsetD[OF factor_subset shifted_member])
          have k1_eq: "k1 = k2"
            using k1_in_factor2 factor2_some fp2
              factor2_singleton
            unfolding gamma_congruence_def by simp
          have shifted_eq: "k1 + n1 = k2"
            using shifted_in_factor2 factor2_some fp2
              factor2_singleton
            unfolding gamma_congruence_def by simp
          have n1_zero: "n1 = 0"
            using k1_eq shifted_eq by simp
          have preimage_le:
            "preimage_times_const r1 k1 <=
             preimage_times_const r2 k2"
```

```
            using preimage_times_const_mono[OF assms(1), of k1]
              k1_eq
            by simp
          from preimage_le r1_some factor1_some rp1 fp1
            r2_some factor2_some rp2 fp2 n1_zero
            factor2_singleton
          show ?thesis
            unfolding less_eq_congruence_iff_gamma
              gamma_congruence_def
            by (simp add: inverse_times_candidate.rep_eq
                preimage_times_const.rep_eq)
        next
          case factor2_nonsingleton: False
          with r2_some factor2_some rp2 fp2
          show ?thesis
            unfolding less_eq_congruence_iff_gamma
              gamma_congruence_def
            by (simp add: inverse_times_candidate.rep_eq)
        qed
      qed
    qed
  qed
qed
```

Backward inverse operators

Congruences carry divisibility information but no order, so less-than cannot refine either operand. True equality is representable as intersection; false equality remains the identity because a congruence generally cannot represent a complement. Addition and subtraction use algebraic rearrangement, while multiplication uses the preimage construction proved above.

```
definition inv_less_congruence ::
    "bool => congruence => congruence => congruence * congruence"
where
  "inv_less_congruence result a b = (a, b)"

fun inv_eq_congruence ::
    "bool => congruence => congruence => congruence * congruence"
where
  "inv_eq_congruence True a b =
     (a ⊓ b, a ⊓ b)"
| "inv_eq_congruence False a b = (a, b)"

definition inv_plus_congruence ::
    "congruence => congruence => congruence =>
     congruence * congruence"
where
  "inv_plus_congruence r a b =
     (a ⊓ (r - b),
      b ⊓ (r - a))"

definition inv_minus_congruence ::
    "congruence => congruence => congruence =>
     congruence * congruence"
where
  "inv_minus_congruence r a b =
     (a ⊓ (r + b),
      b ⊓ (a - r))"

definition inv_times_congruence ::
    "congruence => congruence => congruence =>
     congruence * congruence"
where
  "inv_times_congruence r a b =
     (a ⊓ inverse_times_candidate r b,
      b ⊓ inverse_times_candidate r a)"

lemma inv_less_congruence_sound:
  assumes "x : gamma_congruence a"
      and "y : gamma_congruence b"
  shows
    "x : gamma_congruence
       (fst (inv_less_congruence result a b)) ∧
     y : gamma_congruence
       (snd (inv_less_congruence result a b))"
```

```
  using assms by (simp add: inv_less_congruence_def)

lemma inv_eq_congruence_sound:
  assumes "x : gamma_congruence a"
      and "y : gamma_congruence b"
      and "(x = y) = result"
  shows
    "x : gamma_congruence
       (fst (inv_eq_congruence result a b)) ∧
     y : gamma_congruence
       (snd (inv_eq_congruence result a b))"
  using assms
  by (cases result) auto

lemma inv_plus_congruence_sound:
  assumes "x : gamma_congruence a"
      and "y : gamma_congruence b"
      and "x + y : gamma_congruence r"
  shows
    "x : gamma_congruence
       (fst (inv_plus_congruence r a b)) ∧
     y : gamma_congruence
       (snd (inv_plus_congruence r a b))"
proof -
  have x_candidate: "x : gamma_congruence (r - b)"
    using congruence_minus_sound[OF assms(3) assms(2)]
    by simp
  have y_candidate: "y : gamma_congruence (r - a)"
    using congruence_minus_sound[OF assms(3) assms(1)]
    by simp
  show ?thesis
    using assms(1,2) x_candidate y_candidate
    by (simp add: inv_plus_congruence_def)
qed

lemma inv_minus_congruence_sound:
  assumes "x : gamma_congruence a"
      and "y : gamma_congruence b"
      and "x - y : gamma_congruence r"
  shows
    "x : gamma_congruence
       (fst (inv_minus_congruence r a b)) ∧
     y : gamma_congruence
       (snd (inv_minus_congruence r a b))"
proof -
  have x_candidate: "x : gamma_congruence (r + b)"
    using congruence_plus_sound[OF assms(3) assms(2)]
    by simp
  have y_candidate: "y : gamma_congruence (a - r)"
    using congruence_minus_sound[OF assms(1) assms(3)]
    by simp
  show ?thesis
    using assms(1,2) x_candidate y_candidate
    by (simp add: inv_minus_congruence_def)
qed

lemma inv_times_congruence_sound:
  assumes "x : gamma_congruence a"
      and "y : gamma_congruence b"
      and "x * y : gamma_congruence r"
  shows
    "x : gamma_congruence
       (fst (inv_times_congruence r a b)) ∧
     y : gamma_congruence
       (snd (inv_times_congruence r a b))"
proof -
  have x_candidate:
    "x : gamma_congruence (inverse_times_candidate r b)"
  proof (rule inverse_times_candidate_sound[
      where z = "x * y" and y = y])
    show "x * y : gamma_congruence r"
      by (rule assms(3))
    show "y : gamma_congruence b"
      by (rule assms(2))
    show "x * y = x * y" by simp
  qed
```

```
  have y_candidate:
    "y : gamma_congruence (inverse_times_candidate r a)"
  proof (rule inverse_times_candidate_sound[
      where z = "x * y" and y = x])
    show "x * y : gamma_congruence r"
      by (rule assms(3))
    show "x : gamma_congruence a"
      by (rule assms(1))
    show "y * x = x * y"
      by (simp add: mult.commute)
  qed
  show ?thesis
    using assms(1,2) x_candidate y_candidate
    by (simp add: inv_times_congruence_def)
qed

lemma inv_less_congruence_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows
    "le_pair
      (inv_less_congruence result a1 b1)
      (inv_less_congruence result a2 b2)"
  using assms by (simp add: inv_less_congruence_def)

lemma inv_eq_congruence_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows
    "le_pair
      (inv_eq_congruence result a1 b1)
      (inv_eq_congruence result a2 b2)"
  using assms inf_mono
  by (cases result) auto

lemma inv_plus_congruence_mono:
  assumes "r1 <= r2" and "a1 <= a2" and "b1 <= b2"
  shows
    "le_pair
      (inv_plus_congruence r1 a1 b1)
      (inv_plus_congruence r2 a2 b2)"
  using assms congruence_minus_mono
  by (auto simp: inv_plus_congruence_def intro: le_infI1 le_infI2)

lemma inv_minus_congruence_mono:
  assumes "r1 <= r2" and "a1 <= a2" and "b1 <= b2"
  shows
    "le_pair
      (inv_minus_congruence r1 a1 b1)
      (inv_minus_congruence r2 a2 b2)"
  using assms congruence_plus_mono congruence_minus_mono
  by (auto simp: inv_minus_congruence_def intro: le_infI1 le_infI2)

lemma inv_times_congruence_mono:
  assumes "r1 <= r2" and "a1 <= a2" and "b1 <= b2"
  shows
    "le_pair
      (inv_times_congruence r1 a1 b1)
      (inv_times_congruence r2 a2 b2)"
  using assms inverse_times_candidate_mono
  by (auto simp: inv_times_congruence_def intro: le_infI1 le_infI2)
```

The refinement operations and their certificate

Congruence's refinement choices: the lattice meet, and an inverse for every comparison and arithmetic operator. The guard filters and the branch are derived from this record and the evaluator in `Congruence_Transfer`.

```
definition congruence_refine_ops :: "congruence refine_ops" where
  "congruence_refine_ops =
     ⦇r_tobool = congruence_tobool, r_inv_less = inv_less_congruence,
      r_inv_eq = inv_eq_congruence, r_inv_plus = inv_plus_congruence,
      r_inv_minus = inv_minus_congruence, r_inv_times = inv_times_congruence,
      r_intersect = inf⦈"

lemma congruence_refine_ops_simps [simp]:
  "r_tobool congruence_refine_ops = congruence_tobool"
  "r_inv_less congruence_refine_ops = inv_less_congruence"
  "r_inv_eq congruence_refine_ops = inv_eq_congruence"
```

```
  "r_inv_plus congruence_refine_ops = inv_plus_congruence"
  "r_inv_minus congruence_refine_ops = inv_minus_congruence"
  "r_inv_times congruence_refine_ops = inv_times_congruence"
  "r_intersect congruence_refine_ops = inf"
  by (simp_all add: congruence_refine_ops_def)

lemma congruence_backward_domain:
  "mono_refinement inf aval_congruence congruence_tobool
     inv_less_congruence inv_eq_congruence
     inv_plus_congruence inv_minus_congruence inv_times_congruence"
proof unfold_locales
qed (simp_all add: inv_less_congruence_sound inv_eq_congruence_sound
       inv_plus_congruence_sound inv_minus_congruence_sound inv_times_congruence_sound
       congruence_tobool_sound inf_mono congruence_arith.aval_dom_mono
       inv_less_congruence_mono inv_eq_congruence_mono inv_plus_congruence_mono
       inv_minus_congruence_mono inv_times_congruence_mono
       le_infI1 le_infI2 congruence_tobool_mono)

end
```

14.4 Congruence_Special

```
theory Congruence_Special
  imports Congruence_Arithmetic "Voblint_Nonrelational.Special_Ops"
begin
```

What a Min or Max call tells you about a residue class

`Min a b` and `Max a b` each return one of their two operands, never a newly computed integer, so whatever describes both operands describes the result. Congruence answers with the join of the two argument classes: sound for either selection, and the sharpest answer available without knowing which one is taken. `Nondet_Int` constrains nothing and lands at `top`.

The reader needs `aval_congruence`, Congruence's expression evaluator, from `Voblint_Analysis_Congruence.Congruence_Arithme`

Two known-even operands stay even, exactly as Parity's `parity_min` keeps `PEven`; a mismatch of moduli falls back on their common coarsening rather than on `top`, which is where the join earns over a constant answer.

```
definition congruence_min :: "congruence => congruence => congruence" where
  "congruence_min a b = a ⊔ b"

definition congruence_max :: "congruence => congruence => congruence" where
  "congruence_max a b = a ⊔ b"

lemma gamma_congruence_sup_ub1: "gamma_congruence a ⊆ gamma_congruence (a ⊔ b)"
proof -
  have "γ (a::congruence) ⊆ γ (a ⊔ b)" by (rule gamma_mono[OF sup_ge1])
  then show ?thesis by simp
qed

lemma gamma_congruence_sup_ub2: "gamma_congruence b ⊆ gamma_congruence (a ⊔ b)"
proof -
  have "γ (b::congruence) ⊆ γ (a ⊔ b)" by (rule gamma_mono[OF sup_ge2])
  then show ?thesis by simp
qed

lemma congruence_min_sound:
  assumes "i ∈ gamma_congruence a" and "j ∈ gamma_congruence b"
  shows "min i j ∈ gamma_congruence (congruence_min a b)"
  using assms gamma_congruence_sup_ub1[of a b] gamma_congruence_sup_ub2[of b a]
  unfolding congruence_min_def min_def by auto

lemma congruence_max_sound:
  assumes "i ∈ gamma_congruence a" and "j ∈ gamma_congruence b"
  shows "max i j ∈ gamma_congruence (congruence_max a b)"
  using assms gamma_congruence_sup_ub1[of a b] gamma_congruence_sup_ub2[of b a]
  unfolding congruence_max_def max_def by auto

lemma congruence_min_mono:
  "a1 ≤ a2 ⟹ b1 ≤ b2 ⟹ congruence_min a1 b1 ≤ congruence_min a2 (b2::congruence)"
  unfolding congruence_min_def by (rule sup_mono)

lemma congruence_max_mono:
  "a1 ≤ a2 ⟹ b1 ≤ b2 ⟹ congruence_max a1 b1 ≤ congruence_max a2 (b2::congruence)"
  unfolding congruence_max_def by (rule sup_mono)
```

Special-call dispatch

`congruence_special_ops` pairs `congruence_min` and `congruence_max`, both the join of their operands. Interpreting `mono_minmax_ops` at this pair makes the generic special-call transfer sound and monotone for congruences.

```
definition congruence_special_ops :: "congruence special_ops" where
  "congruence_special_ops = (| special_min = congruence_min, special_max = congruence_max |)"

lemma congruence_special_ops_min [simp]: "special_min congruence_special_ops = congruence_min"
  by (simp add: congruence_special_ops_def)

lemma congruence_special_ops_max [simp]: "special_max congruence_special_ops = congruence_max"
  by (simp add: congruence_special_ops_def)

interpretation congruence_special: mono_minmax_ops congruence_special_ops aval_congruence
proof unfold_locales
  fix i j :: int and p q :: congruence
  assume "i ∈ γ p" and "j ∈ γ q"
  then show "min i j ∈ γ (special_min congruence_special_ops p q)"
```

```
    using congruence_min_sound by simp
next
  fix i j :: int and p q :: congruence
  assume "i ∈ γ p" and "j ∈ γ q"
  then show "max i j ∈ γ (special_max congruence_special_ops p q)"
    using congruence_max_sound by simp
next
  fix p1 p2 q1 q2 :: congruence
  assume "p1 ≤ p2" and "q1 ≤ q2"
  then show "special_min congruence_special_ops p1 q1
              ≤ special_min congruence_special_ops p2 q2"
    by (simp add: congruence_min_mono)
next
  fix p1 p2 q1 q2 :: congruence
  assume "p1 ≤ p2" and "q1 ≤ q2"
  then show "special_max congruence_special_ops p1 q1
              ≤ special_max congruence_special_ops p2 q2"
    by (simp add: congruence_max_mono)
qed

fun special_congruence ::
    "special_call => vname => (vname => congruence) => (vname => congruence)"
where
  "special_congruence Nondet_Int x d = d(x := top)"
| "special_congruence (Min a b) x d =
     d(x := congruence_min (aval_congruence a d) (aval_congruence b d))"
| "special_congruence (Max a b) x d =
     d(x := congruence_max (aval_congruence a d) (aval_congruence b d))"

lemma special_congruence_eq_transfer:
  "special_congruence sc x d = congruence_special.special_transfer sc x d"
  by (cases sc) simp_all

end
```

14.5 Congruence_Transfer

```
theory Congruence_Transfer
  imports
    Congruence_Backward
    Congruence_Special
    "Voblint_Nonrelational.Nonrelational_Transfer"
begin
```

What each kind of edge does to a map from variables to residue classes

One operation per edge the framework can hand a domain: an assignment writes the evaluated right-hand side, a guard runs the backward branch, a call binds the actuals into a fresh frame, a return publishes its expression into the return variable, and skip, body entry and check observation leave the map alone. None of that is about modular arithmetic, so none of it is stated here: it is `sound_nonrelational_ops`, proved sound and monotone once for every domain that gives one abstract value per variable.

What is Congruence's own are the primitive choices. `congruence_ops` collects the evaluator, the comparison tables, `congruence_refine_ops` — whose inverses make `x + 1 == 3` narrow `x` to a single integer — the special calls, and $\top$. The one interpretation below derives the guard filters, the branch, the check classifier and the transfer from that bundle, and rewrites the locale's special-call dispatch to `special_congruence`, Congruence's own `fun`, rather than introducing a second name for it.

```
lemma congruence_check_domain:
  "sound_check_query congruence_lt congruence_eqb gamma_state aval_congruence"
  by unfold_locales (rule congruence_arith.aval_abs_sound)

definition congruence_ops :: "congruence nonrelational_ops" where
  "congruence_ops =
     (| n_aval = aval_congruence, n_query = (|q_less = congruence_lt, q_eq = congruence_eqb|),
       n_refine = congruence_refine_ops, n_special = congruence_special_ops |)"

lemma congruence_ops_simps [simp]:
  "n_aval congruence_ops = aval_congruence"
  "q_less (n_query congruence_ops) = congruence_lt"
  "q_eq (n_query congruence_ops) = congruence_eqb"
  "n_refine congruence_ops = congruence_refine_ops"
  "n_special congruence_ops = congruence_special_ops"
  by (simp_all add: congruence_ops_def)

global_interpretation congruence_tf: mono_nonrelational_ops congruence_ops
  rewrites "sound_minmax_ops.special_transfer (n_special congruence_ops) (n_aval congruence_ops)
          = special_congruence"
  defines assign_congruence = congruence_tf.assign
    and skip_congruence = congruence_tf.skip
    and body_congruence = congruence_tf.body
    and event_congruence = congruence_tf.event
    and return_congruence = congruence_tf.ret
    and enter_frame_congruence_for = congruence_tf.enter_frame_for
    and enter_congruence_for = congruence_tf.enter_for
    and enter_congruence_ci_for = congruence_tf.enter_ci_for
    and congruence_tf_abs = congruence_tf.tf_abs
    and afilter_congruence = congruence_tf.backward.afilter
    and feasible_congruence = congruence_tf.backward.feasible
    and bfilter_congruence = congruence_tf.backward.bfilter
    and branch_congruence = congruence_tf.backward.branch
    and congruence_truthy_query = congruence_tf.check.truthy_query
    and congruence_check_query = congruence_tf.check.check_query
    and congruence_classify_check = congruence_tf.check.classify_check
    and congruence_eval_answer = congruence_tf.check.eval_answer
    and congruence_checks_proven = congruence_tf.check.abstract_checks_proven
proof -
  show "mono_nonrelational_ops congruence_ops"
  proof (rule mono_nonrelational_opsI,
         unfold congruence_ops_simps congruence_refine_ops_simps)
    show "mono_minmax_ops congruence_special_ops aval_congruence"
      by (rule congruence_special.mono_minmax_ops_axioms)
    show "mono_refinement inf aval_congruence congruence_tobool
            inv_less_congruence inv_eq_congruence
            inv_plus_congruence inv_minus_congruence inv_times_congruence"
      by (rule congruence_backward_domain)
    show "sound_check_query congruence_lt congruence_eqb gamma_state aval_congruence"
      by (rule congruence_check_domain)
  qed
qed (intro ext, simp add: special_congruence_eq_transfer)
```

No fact is renamed. The transfer functions get Congruence-prefixed names above because they are constants a caller applies, exported to OCaml and named in registration data; the theorems about them stay under `congruence_tf.`, which is where a reader looks to find out that they are the generic ones rather than Congruence's own. `skip_congruence`'s soundness is `congruence_tf.skip_sound`, the branch's is `congruence_tf.backward.branch_sound`, and the framework's transfer contract at Congruence is `congruence_tf.is_sound_nonrelational_transfer`.

end

14.6 Congruence_Exec

```
theory Congruence_Exec
  imports
    Congruence_Transfer
begin
```

Congruence on the executable carrier

The executable step and procedure entry are `Voblint_Nonrelational.Nonrelational_Ops`'s generic constructions at the bundle, `generic_tf_st_for congruence_ops` and `generic_enter_st_for congruence_ops`, and their agreement with the abstract transfer is `congruence_tf.tf_st_for_commute` and `congruence_tf.enter_st_for_commute`: nothing about them is the domain's to state.

The state a run starts in: a declared global holds the single integer `0`, every local is unconstrained.

```
abbreviation cinit_congruence_st :: "congruence default_st" where
  "cinit_congruence_st ≡ initial_default_st top (congruence_of_int 0)"

end
```

14.7 Congruence_Sound

```
theory Congruence_Sound
  imports
    Congruence_Exec
begin
```

What a run's initial abstract state describes

A run starts with every declared global at zero and every local unconstrained. The one lemma here says `cinit_congruence_st` describes exactly that: the singleton residue class `congruence_of_int 0` covers a declared global precisely, and $\top$ covers an unconstrained local trivially. It is the one place the zero-initialization of globals is turned into a residue class.

Congruence's specification, its concretization and the soundness of the one against the other are not restated here. Those are constants and theorems of `dg_domain_exec`, and an assembly reaches them by interpreting that locale from Congruence's own commute lemmas in `Voblint_Analysis_Congruence.Congruence_Exec` — so a per-domain copy would only rename what the locale already proves.

```
lemma congruence_cinit_gamma:
  "cinit_stores 𝒢 ⊆ default_st_gamma 𝒢 cinit_congruence_st"
  by (auto simp: cinit_stores_def gamma_state_def default_st_gamma_initial)

end
```

14.8 Congruence_Classify

theory Congruence_Classify
  imports
    Congruence_Exec
begin

Deciding a check from a map of residue classes

The check classifier is derived from Congruence's bundle: the Boolean recursion over `exp` (`Not`, `And`, `Or`), the three-way classification, and the node-indexed bridge to `checks_proven` come from the comparison tables `congruence_lt`/`congruence_e` and the evaluator, through the `congruence_tf` interpretation in `Voblint_Analysis_Congruence.Congruence_Transfer`. Their facts stay under `congruence_tf.check.`; `Congruence_Regression` exercises them.

end

14.9 Congruence_Analyses

```
theory Congruence_Analyses
  imports
    Congruence_Sound
    "Voblint_Nonrelational.Nonrel_Registration"
begin
```

Registering Congruence at every context and update rule

Congruence runs through the shared D/G pipeline at the unit context. The CLI runs it as a field of the combined state of `MCP_Analyses`, whose component and soundness the unit registration supplies. Each registration leaves the rule that merges a value side-effected into a global as a parameter `r`. The equation system, the solve, the result table and every soundness endpoint come from the interpreted locale; this theory only names the domain's own implementation and facts.

At the unit context

```
global_interpretation congruence_rule:
    nonrel_unit_rule congruence_ops cinit_congruence_st r
  for r
  by (intro nonrel_unit_ruleI nonrel_registrationI
      congruence_tf.sound_nonrelational_ops_axioms congruence_cinit_gamma)

end
```

Chapter 15

Voblint_Analysis_Relational

15.1 Rel_Order_Domain

```
theory Rel_Order_Domain
  imports "Voblint_Framework.DG_Spec_Sound" "Voblint_Framework.DG_Indexed_Generator"
    "Voblint_Framework.State_Restriction" "Voblint_Domain.Order_Lattice"
begin
```

A minimal relational analysis for `analysis_contract`

The analysis over `relc`, the order lattice of `Voblint_Domain.Order_Lattice`. The purpose of this file is not a useful analysis. It demonstrates that a non-`abs_state` carrier discharges `analysis_contract` with zero changes to the DG framework. Every transfer below is deliberately the most imprecise sound choice (forget on assign, havoc on call) except for a precise `assume_step`/`assume_not_step` pair, which is enough to make the carrier genuinely relational.

Local and global state together

```
definition gammaDG_relc :: "relc ⇒ relc ⇒ store set" where
  "gammaDG_relc d g = γ d ∩ γ g"

lemma gammaDG_relc_top [simp]: "gammaDG_relc ⊤ ⊤ = UNIV"
  unfolding gammaDG_relc_def by simp
```

Refining bare-variable comparisons

Only guards between two bare variables refine a `relc`. On the true branch `assume_step` adds `(x, y)` for a guard implying that `x` is at most `y`; every other guard leaves the state unchanged.

```
definition assume_step :: "exp ⇒ relc ⇒ relc" where
  "assume_step b d =
     (case d of
        RelBot ⇒ RelBot
      | RelC ps ⇒
          (case b of
             Less (V x) (V y) ⇒ RelC (insert (x, y) ps)
           | LessEq (V x) (V y) ⇒ RelC (insert (x, y) ps)
           | Greater (V x) (V y) ⇒ RelC (insert (y, x) ps)
           | GreaterEq (V x) (V y) ⇒ RelC (insert (y, x) ps)
           | Eq (V x) (V y) ⇒ RelC (insert (x, y) (insert (y, x) ps))
           | _ ⇒ RelC ps))"

lemma assume_step_sound[intro]:
  assumes "s ∈ γ d" "truthy (⟦b⟧e s)"
  shows "s ∈ γ (assume_step b d)"
  using assms
  by (cases d; cases b) (auto simp: assume_step_def split: exp.splits if_splits)
```

The false branch records the reversed order. Equality records both directions, on the true branch of `Eq` or the false branch of `NotEq`. Disequality alone cannot be represented by a conjunction of weak orders.

```
definition assume_not_step :: "exp ⇒ relc ⇒ relc" where
  "assume_not_step b d =
     (case d of
        RelBot ⇒ RelBot
      | RelC ps ⇒
          (case b of
             Less (V x) (V y) ⇒ RelC (insert (y, x) ps)
           | LessEq (V x) (V y) ⇒ RelC (insert (y, x) ps)
           | Greater (V x) (V y) ⇒ RelC (insert (x, y) ps)
           | GreaterEq (V x) (V y) ⇒ RelC (insert (x, y) ps)
           | NotEq (V x) (V y) ⇒ RelC (insert (x, y) (insert (y, x) ps))
           | _ ⇒ RelC ps))"

lemma assume_not_step_sound[intro]:
  assumes "s ∈ γ d" "¬ truthy (⟦b⟧e s)"
  shows "s ∈ γ (assume_not_step b d)"
  using assms
  by (cases d; cases b) (auto simp: assume_not_step_def split: exp.splits if_splits)
```

The transfer functions

Every step except the comparison refinements above is sound by forgetting or by leaving the carrier untouched – deliberately imprecise: no closure, no normalization, havoc-based calls.

```
definition relc_skip :: "relc ⇒ relc ⇒ relc × relc" where
  "relc_skip d g = (g, d)"

definition relc_assign :: "vname ⇒ exp ⇒ relc ⇒ relc ⇒ relc × relc" where
  "relc_assign x e d g = (forget_relc x g, forget_relc x d)"

definition relc_body :: "pname ⇒ relc ⇒ relc ⇒ relc × relc" where
  "relc_body p d g = (g, d)"
```

`return` reuses the same forget-based imprecision `relc_assign` already applies to every ordinary assignment: with an expression, forget `ret_var`; without one, behave like `skip`.

```
definition relc_return :: "exp option ⇒ pname ⇒ relc ⇒ relc ⇒ relc × relc" where
  "relc_return e p d g =
     (case e of None ⇒ relc_skip d g
      | Some a ⇒ relc_assign ret_var a d g)"

definition relc_special :: "special_call ⇒ vname ⇒ relc ⇒ relc ⇒ relc × relc" where
  "relc_special sc x d g = (forget_relc x g, forget_relc x d)"
```

A check observes its condition but never refines the state, matching `relc_skip`'s own imprecision.

```
definition relc_event :: "analysis_event ⇒ relc ⇒ relc ⇒ relc × relc" where
  "relc_event ev d g = (g, d)"
```

`assume_step`/`assume_not_step` stay separate, genuinely asymmetric operations (`x < y` vs. its mirror `y ≤ x` insert different pairs, not the same formula under a polarity flag); only the interface-level dispatch consolidates them into one polarity-parametrized branch operation.

```
definition relc_branch_step :: "exp ⇒ bool ⇒ relc ⇒ relc" where
  "relc_branch_step b pol d = (if pol then assume_step b d else assume_not_step b d)"
```

The relational counterpart of `bfilter_sound`, in the same shape: refining at a guard keeps every store at which the guard has the required truth value.

```
lemma relc_branch_step_sound [intro]:
  assumes "s ∈ γ d" "truthy (⟦b⟧e s) = pol"
  shows "s ∈ γ (relc_branch_step b pol d)"
  using assms by (cases pol) (auto simp: relc_branch_step_def)

definition relc_branch :: "exp ⇒ bool ⇒ relc ⇒ relc ⇒ relc × relc" where
  "relc_branch b pol d g = (g, relc_branch_step b pol d)"

definition relc_enter :: "call_info ⇒ relc ⇒ relc ⇒ relc × relc" where
  "relc_enter ci dc g = (⊤, ⊤)"
```

The caller continuation is the identity. This carrier discards every caller relation at its environment merge anyway, so there is no call-side invalidation for a continuation to express: filtering before a merge that already returns $\top$ would be indistinguishable from not filtering. Keeping it identity leaves this instance the least interesting sound one, which is its purpose.

```
definition relc_caller_cont :: "call_info ⇒ relc ⇒ relc ⇒ relc" where
  "relc_caller_cont ci dc g = dc"

definition relc_combine_env :: "call_info ⇒ relc ⇒ relc ⇒ relc ⇒ relc × relc" where
  "relc_combine_env ci dc de g = (⊤, ⊤)"

definition relc_combine_assign ::
  "call_info ⇒ relc ⇒ relc ⇒ relc × relc ⇒ relc × relc"
where
  "relc_combine_assign ci de g merged = merged"
```

Unlike the local-only domains, this one really uses the global channel: `relc_special` forgets the assigned name on both halves, and entry and the environment merge reset the shared relation to $\top$. Its transfers are therefore written as a read-compute-publish sequence.

`relc_transfer` is the adapter that turns one of this file's `d ⇒ g ⇒ (g, d)` operations into a manager transfer: query the shared relation, run the operation, publish its global half, answer with its local half. The manager interface does not require that pairing shape; here it is one domain's private convenience, which is the point.

```
definition relc_transfer ::
  "(relc ⇒ relc ⇒ relc × relc) ⇒ ('x,'k,unit,relc,relc) man_transfer"
where
  "relc_transfer f m =
     do {
       g ← man_global m ();
```

```
      let r = f (man_local m) g;
      _ ← man_sideg m () (fst r);
      sp_return (snd r)
    }"
```

Entry at this carrier: the same global read and publication, answering the one alternative whose continuation is the caller value unchanged.

```
definition relc_enter_transfer ::
  "(relc ⇒ relc ⇒ relc × relc) ⇒ ('x,'k,unit,relc,relc) man_enter_transfer"
where
  "relc_enter_transfer f m =
    do {
      g ← man_global m ();
      let r = f (man_local m) g;
      _ ← man_sideg m () (fst r);
      sp_return [(man_local m, snd r)]
    }"

definition relc_combine_transfer ::
  "(relc ⇒ relc ⇒ relc ⇒ relc × relc) ⇒ ('x,'k,unit,relc,relc) man_combine_transfer"
where
  "relc_combine_transfer f m de =
    do {
      g ← man_global m ();
      let r = f (man_local m) de g;
      _ ← man_sideg m () (fst r);
      sp_return (snd r)
    }"
```

The observations of a compiled `relc_transfer`: its answer is the operation's local half, and what it publishes at the routed key is the global half. These are what `analysis_contract` is stated against.

```
lemma traverse_relc_transfer [simp]:
  "dg_local (traverse_program (transfer_program (relc_transfer f) src (λ_. gk)) τ)
    = snd (f (dg_local (τ src)) (dg_global (τ (Inr gk))))"
  by (cases src)
    (simp_all add: transfer_program_def transfer_program_at_def relc_transfer_def
      mk_dg_man_def dg_read_at_def dg_read_global_def dg_sideg_def sp_bind_assoc
      Let_def)

lemma sides_relc_transfer [simp]:
  "dg_global (sides_of_program (transfer_program (relc_transfer f) src (λ_. gk)) τ (Inr gk))
    = fst (f (dg_local (τ src)) (dg_global (τ (Inr gk))))"
  by (cases src)
    (simp_all add: transfer_program_def transfer_program_at_def relc_transfer_def
      mk_dg_man_def dg_read_at_def dg_read_global_def dg_sideg_def sp_bind_assoc
      Let_def)
```

The order analysis never asks, so the query channel the generator installs leaves its transfers unchanged.

```
lemma relc_transfer_outer_man [simp]:
  "relc_transfer f (outer_man Q m) = relc_transfer f m"
  by (simp add: relc_transfer_def)

definition rel_order_spec :: "('x,'k,unit,relc,relc) dg_spec" where
  "rel_order_spec = local_dg_spec_template⦇
    dgs_skip := relc_transfer relc_skip,
    dgs_assign := (λx e. relc_transfer (relc_assign x e)),
    dgs_special := (λsc x. relc_transfer (relc_special sc x)),
    dgs_branch := (λb pol. relc_transfer (relc_branch b pol)),
    dgs_body := (λp. relc_transfer (relc_body p)),
    dgs_return := (λe p. relc_transfer (relc_return e p)),
    dgs_enter := (λci. relc_enter_transfer (relc_enter ci)),
    dgs_event := (λev. relc_transfer (relc_event ev)),
    dgs_combine_env := (λci. relc_combine_transfer (relc_combine_env ci))
   ⦈"
```

Expand the template and transfer in this code equation so the argument-free specification is a record of functions in generated ML. This avoids the value restriction on record updates without inlining the specification into its callers.

```
lemmas [code] =
  rel_order_spec_def[unfolded local_dg_spec_template_def relc_transfer_def, simplified]
```

Every field reads the shared slot, publishes once and answers, so the specification runs its continuation once wherever the generator runs it.

```
lemma dg_spec_wf_rel_order_spec [intro, simp]: "dg_spec_wf rel_order_spec"
proof (unfold dg_spec_wf_def, intro conjI allI impI)
  fix a :: edge_action and d :: relc and unknown_of :: "unit ⇒ 'b" and A
  show "sp_wf (dg_spec_step rel_order_spec a ((mk_dg_man d unknown_of)(|man_ask := A|)))"
    by (cases a) (auto simp: rel_order_spec_def relc_transfer_def Let_def mk_dg_man_def)
next
  fix d :: relc and unknown_of :: "unit ⇒ 'b" and A q
  show "sp_wf (dgs_query rel_order_spec ((mk_dg_man d unknown_of)(|man_ask := A|)) q)"
    by (simp add: rel_order_spec_def)
next
  fix ci and d :: relc and unknown_of :: "unit ⇒ 'b"
  show "sp_wf (enter♯ rel_order_spec ci (mk_dg_man d unknown_of))"
    by (auto simp: rel_order_spec_def relc_enter_transfer_def Let_def)
next
  fix ci and d :: relc and unknown_of :: "unit ⇒ 'b" and ex :: relc
  show "sp_wf (dg_spec_combine_transfer rel_order_spec ci (mk_dg_man d unknown_of) ex)"
    by (auto simp: rel_order_spec_def dg_spec_combine_transfer_def relc_combine_transfer_def
        local_combine_transfer_def Let_def)
qed

named_theorems rel_order_simps

declare
  relc_branch_def     [rel_order_simps]
  relc_branch_step_def    [rel_order_simps]
  relc_skip_def       [rel_order_simps]
  relc_body_def       [rel_order_simps]
  relc_return_def     [rel_order_simps]
  relc_event_def      [rel_order_simps]
  rel_order_spec_def     [rel_order_simps]
  gammaDG_relc_def        [rel_order_simps]
  relc_assign_def     [rel_order_simps]
  relc_special_def    [rel_order_simps]
```

Per-edge soundness

Each edge operation over-approximates the concrete edge semantics on `gammaDG_relc`, the stores both the local and the global component describe. Assignments and special calls forget every pair that mentions the target.

```
lemma relc_skip_sound[intro]:
  "edge_collect EA_Nop (gammaDG_relc d g) ⊆
     (case relc_skip d g of (g', d') ⇒ gammaDG_relc d' g')"
  unfolding relc_skip_def by simp

lemma relc_assign_sound[intro]:
  "edge_collect (EA_Assign x e) (gammaDG_relc d g) ⊆
     (case relc_assign x e d g of (g', d') ⇒ gammaDG_relc d' g')"
proof -
  have "edge_collect (EA_Assign x e) (gammaDG_relc d g)
      = {s(x := ⟦e⟧e s) | s. s ∈ gammaDG_relc d g}"
    by simp
  also have "... ⊆ γ (forget_relc x d) ∩ γ (forget_relc x g)"
    using forget_relc_sound unfolding gammaDG_relc_def by blast
  finally show ?thesis
    unfolding relc_assign_def gammaDG_relc_def by simp
qed

lemma relc_special_sound[intro]:
  "edge_collect (EA_Special sc x) (gammaDG_relc d g) ⊆
     (case relc_special sc x d g of (g', d') ⇒ gammaDG_relc d' g')"
proof -
  have "edge_collect (EA_Special sc x) (gammaDG_relc d g)
      ⊆ {s(x := v) | s v. s ∈ gammaDG_relc d g}"
    by (cases sc) auto
  also have "... ⊆ γ (forget_relc x d) ∩ γ (forget_relc x g)"
    using forget_relc_sound unfolding gammaDG_relc_def by blast
  finally show ?thesis
    unfolding relc_special_def gammaDG_relc_def by simp
qed

lemma relc_branch_sound_True[intro]:
  "edge_collect (EA_Assume b) (gammaDG_relc d g) ⊆
     (case relc_branch b True d g of (g', d') ⇒ gammaDG_relc d' g')"
proof -
  have "edge_collect (EA_Assume b) (gammaDG_relc d g)
      = {s. s ∈ gammaDG_relc d g ∧ truthy (⟦b⟧e s)}"
```

```
    by simp
  also have "... ⊆ γ (assume_step b d) ∩ γ g"
    using assume_step_sound unfolding gammaDG_relc_def by blast
  finally show ?thesis
    unfolding relc_branch_def relc_branch_step_def gammaDG_relc_def by simp
qed

lemma relc_branch_sound_False[intro]:
  "edge_collect (EA_AssumeNot b) (gammaDG_relc d g) ⊆
     (case relc_branch b False d g of (g', d') ⇒ gammaDG_relc d' g')"
proof -
  have "edge_collect (EA_AssumeNot b) (gammaDG_relc d g)
      = {s. s ∈ gammaDG_relc d g ∧ ¬ truthy (⟦b⟧e s)}"
    by simp
  also have "... ⊆ γ (assume_not_step b d) ∩ γ g"
    using assume_not_step_sound unfolding gammaDG_relc_def by blast
  finally show ?thesis
    unfolding relc_branch_def relc_branch_step_def gammaDG_relc_def by simp
qed

lemma relc_return_sound[intro]:
  "edge_collect (EA_Ret e p) (gammaDG_relc d g) ⊆
     (case relc_return e p d g of (g', d') ⇒ gammaDG_relc d' g')"
proof (cases e)
  case None
  then show ?thesis
    by (simp add: relc_return_def relc_skip_def)
next
  case (Some a)
  have "edge_collect (EA_Ret (Some a) p) (gammaDG_relc d g)
      = {s(ret_var := ⟦a⟧e s) | s. s ∈ gammaDG_relc d g}"
    by simp
  also have "... ⊆ γ (forget_relc ret_var d) ∩ γ (forget_relc ret_var g)"
    using forget_relc_sound unfolding gammaDG_relc_def by blast
  finally show ?thesis
    using Some
    by (simp add: relc_return_def relc_assign_def gammaDG_relc_def)
qed
```

The edge dispatch as one pure pairing operation, so the specification's compiled step reduces to `relc_transfer` of it.

```
fun relc_step_for :: "edge_action ⇒ relc ⇒ relc ⇒ relc × relc" where
  "relc_step_for EA_Nop = relc_skip"
| "relc_step_for (EA_Assign x e) = relc_assign x e"
| "relc_step_for (EA_Special sc x) = relc_special sc x"
| "relc_step_for (EA_Assume b) = relc_branch b True"
| "relc_step_for (EA_AssumeNot b) = relc_branch b False"
| "relc_step_for (EA_Body p) = relc_body p"
| "relc_step_for (EA_Ret e p) = relc_return e p"
| "relc_step_for (EA_Check l cnd) = relc_event (Check_Event l cnd)"

lemma dg_spec_step_rel_order_spec [simp]:
  "dg_spec_step rel_order_spec a = relc_transfer (relc_step_for a)"
  unfolding rel_order_spec_def by (cases a) simp_all

lemma dgs_enter_rel_order_spec [simp]:
  "enter♯ rel_order_spec ci = relc_enter_transfer (relc_enter ci)"
  unfolding rel_order_spec_def by simp

lemma relc_step_sound:
  "edge_collect a (gammaDG_relc d g) ⊆
     (case relc_step_for a d g of (g', d') ⇒ gammaDG_relc d' g')"
  by (cases a) (auto simp add: rel_order_simps split: option.splits)
```

Call-entry and combine soundness – havoc-based, both trivial via ⊤

The composed return pipeline: `combine_assign` keeps the template's identity default, so the whole combine is the environment merge, which resets both halves to ⊤.

```
lemma dg_spec_combine_transfer_rel_order_spec [simp]:
  "dg_spec_combine_transfer rel_order_spec ci = relc_combine_transfer (relc_combine_env ci)"
  unfolding dg_spec_combine_transfer_def rel_order_spec_def
  by (intro ext)
     (simp add: local_transfer_def local_combine_transfer_def relc_combine_transfer_def)
```

```
lemma traverse_relc_combine [simp]:
  "dg_local (traverse_program
             (combine_transfer_program (relc_combine_transfer f) src_cc src_ex (λ_. gk)) τ)
     = snd (f (dg_local (τ src_cc)) (dg_local (τ src_ex)) (dg_global (τ (Inr gk))))"
  by (cases src_cc; cases src_ex)
     (simp_all add: combine_transfer_program_def combine_program_at_def relc_combine_transfer_def
        mk_dg_man_def dg_read_at_def dg_read_global_def dg_sideg_def sp_bind_assoc Let_def)

lemma sides_relc_combine [simp]:
  "dg_global (sides_of_program
            (combine_transfer_program (relc_combine_transfer f) src_cc src_ex (λ_. gk))
            τ (Inr gk))
     = fst (f (dg_local (τ src_cc)) (dg_local (τ src_ex)) (dg_global (τ (Inr gk))))"
  by (cases src_cc; cases src_ex)
     (simp_all add: combine_transfer_program_def combine_program_at_def relc_combine_transfer_def
        mk_dg_man_def dg_read_at_def dg_read_global_def dg_sideg_def sp_bind_assoc Let_def)
```

The interpretation

Interpreting `analysis_contract` at `rel_order_spec` discharges the framework's obligations: a well-formed spec, monotone concretization, and sound edge and combine transfers, each from the per-edge lemmas above.

```
interpretation rel_order: analysis_contract rel_order_spec "λd e. gammaDG_relc d (e ())" 𝒢
proof (rule analysis_contract_unitI)
  show "dg_spec_wf rel_order_spec" by (rule dg_spec_wf_rel_order_spec)
next
  fix d d' :: relc and g g' :: relc
  assume "d ≤ d'" "g ≤ g'"
  then show "gammaDG_relc d g ⊆ gammaDG_relc d' g'"
    using gamma_relc_mono unfolding gammaDG_relc_def by blast
next
  fix a and τ :: "'a + 'b ⇒ (relc, relc) dg_state" and src gk
  show "edge_collect a (gammaDG_relc (dg_local (τ src)) (dg_global (τ (Inr gk))))
        ⊆ gammaDG_relc
            (edge_out rel_order_spec a src (λ_. gk) τ)
            (dg_global (sides_of_program (dg_spec_edge_program rel_order_spec a src (λ_. gk))
                       τ (Inr gk)))"
    using relc_step_sound[of a "dg_local (τ src)" "dg_global (τ (Inr gk))"]
    by (simp add: dg_spec_edge_program_def split: prod.splits)
next
  fix s t dc de and τ :: "'a + 'b ⇒ (relc, relc) dg_state" and gk ci
  show "⟦s ∈ gammaDG_relc dc (dg_global (τ (Inr gk)));
         t ∈ gammaDG_relc de (dg_global (τ (Inr gk)))⟧ ⟹
         combine_collect 𝒢 (ci_dst ci) s t
           ∈ gammaDG_relc
               (dg_local (traverse_rhs (sp_compile_with (λd. DG d bot)
                  (dg_spec_combine_transfer rel_order_spec ci
                     (mk_dg_man dc (λ_. gk)) de)) τ))
               (dg_global (sides_of_rhs (sp_compile_with (λd. DG d bot)
                  (dg_spec_combine_transfer rel_order_spec ci
                     (mk_dg_man dc (λ_. gk)) de)) τ (Inr gk)))"
    by (simp add: relc_combine_transfer_def mk_dg_man_def dg_read_global_def
        dg_sideg_def sp_bind_assoc Let_def relc_combine_env_def)
qed

end
```

15.2 Rel_Order_Local

```
theory Rel_Order_Local
  imports Rel_Order_Domain "Voblint_Framework.MCP_Spec"
begin
```

The order carrier as a component that asks

`rel_order_spec` reads and publishes the global channel, so it is not a local specification and cannot join the combination of cooperating analyses. This theory gives the same carrier a local-only form. Intraprocedurally it is the order analysis of `Voblint_Analysis_Relational.Rel_Order_Domain`, with one addition: after an assignment it asks the query channel how the assigned value compares with the other variables, and records every order the answer fixes. At calls it is deliberately coarse, entering and returning with no facts, so that the component studies cooperation and not relational call boundaries.

Answering queries from the order

A pair `(x, y)` means `s x` $\leq$ `s y`. It therefore decides `x <= y` and `y >= x` as true, `y < x` and `x > y` as false, and with the reverse pair also `x == y` as true and `x != y` as false. A comparison evaluates to `1` when true and `0` when false, so the answers are the exact integers `1` and `0`. Every other query is answered with $\top$, the claim that holds of every store.

`relc_le d a b` holds when `a` and `b` are variables whose order `d` records, or the same variable. The reflexive case lets a copy `x = y` learn both orders between `x` and `y` from the channel's own answer to `y <= y`.

```
fun var_of :: "exp ⇒ vname option" where
  "var_of (V x) = Some x"
| "var_of _ = None"

definition relc_le :: "relc ⇒ exp ⇒ exp ⇒ bool" where
  "relc_le d a b =
     (case (var_of a, var_of b) of (Some x, Some y) ⇒ x = y ∨ relc_has x y d | _ ⇒ False)"

definition relc_eval :: "relc ⇒ exp ⇒ answer" where
  "relc_eval d e =
     (case e of
        LessEq a b ⇒ if relc_le d a b then answer_of_int 1 else ⊤
      | GreaterEq a b ⇒ if relc_le d b a then answer_of_int 1 else ⊤
      | Less a b ⇒ if relc_le d b a then answer_of_int 0 else ⊤
      | Greater a b ⇒ if relc_le d a b then answer_of_int 0 else ⊤
      | exp.Eq a b ⇒ if relc_le d a b ∧ relc_le d b a then answer_of_int 1 else ⊤
      | NotEq a b ⇒ if relc_le d a b ∧ relc_le d b a then answer_of_int 0 else ⊤
      | _ ⇒ ⊤)"

fun relc_qry :: "relc ⇒ channel" where
  "relc_qry d (EvalInt e) = relc_eval d e"

lemma relc_has_sound: "s ∈ γ d ⟹ relc_has x y d ⟹ s x ≤ s y"
  by (cases d) auto

lemma var_of_SomeD: "var_of a = Some x ⟹ a = V x"
  by (cases a) simp_all

lemma relc_le_sound: "s ∈ γ d ⟹ relc_le d a b ⟹ ⟦a⟧ₑ s ≤ ⟦b⟧ₑ s"
  by (auto simp: relc_le_def split: option.splits dest!: var_of_SomeD
      dest: relc_has_sound)

lemma relc_eval_sound:
  "s ∈ γ d ⟹ ⟦e⟧ₑ s ∈ gamma_answer (relc_eval d e)"
  by (cases e) (auto simp: relc_eval_def
      dest: relc_le_sound intro: order_antisym)

lemma relc_qry_sound: "s ∈ γ d ⟹ eval_holds q (relc_qry d q) s"
  by (cases q) (simp add: relc_eval_sound)
```

Learning orders from the query channel

At `x = e` the question is asked about the state before the assignment, where `e` evaluates to the new value of `x` and every other variable keeps its value. The exact answer `1` to `e <= y` therefore makes `(x, y)` true afterwards, and `y <= e` makes `(y, x)` true. The candidate partners are the variables in `ys`, a parameter so that the construction stays executable.

```
definition relc_learn :: "channel ⇒ vname list ⇒ vname ⇒ exp ⇒ relc ⇒ relc" where
  "relc_learn ch ys x e d =
     (case d of
        RelBot ⇒ RelBot
      | RelC ps ⇒ RelC (ps
          ∪ set (map (λy. (x, y))
                (filter (λy. y ≠ x ∧ answer_const (ch (EvalInt (LessEq e (V y)))) = Some 1) ys))
          ∪ set (map (λy. (y, x))
                (filter (λy. y ≠ x ∧ answer_const (ch (EvalInt (LessEq (V y) e))) = Some 1) ys))))"

lemma relc_learn_sound:
  assumes "s(x := ⟦e⟧e s) ∈ γ d"
    and "eval_query.channel_holds ch s"
  shows "s(x := ⟦e⟧e s) ∈ γ (relc_learn ch ys x e d)"
proof (cases d)
  case RelBot
  with assms(1) show ?thesis by simp
next
  case (RelC ps)
  have up: "⟦e⟧e s ≤ s y" if "answer_const (ch (EvalInt (LessEq e (V y)))) = Some 1" for y
    using eval_holds_constD[OF eval_query.channel_holdsD[OF assms(2)] that]
    by (simp split: if_splits)
  have down: "s y ≤ ⟦e⟧e s" if "answer_const (ch (EvalInt (LessEq (V y) e))) = Some 1" for y
    using eval_holds_constD[OF eval_query.channel_holdsD[OF assms(2)] that]
    by (simp split: if_splits)
  show ?thesis
    using assms(1) RelC up down by (auto simp: relc_learn_def)
qed
```

The component

The order analysis as a component of the combined state, over the variables `ys` it relates. It starts from the conservative local specification: skip, body and events keep the relation, and the first return stage keeps the caller's. It supplies the operations without a default – assignments ask and learn, the callee starts from the empty relation, and the return keeps nothing, which is sound and imprecise by design – and overrides the handler and the branch, where it decides more than the default.

```
definition relc_ret :: "exp option ⇒ relc ⇒ relc" where
  "relc_ret eo d = (case eo of None ⇒ d | Some a ⇒ forget_relc ret_var d)"

definition order_spec :: "vname list ⇒ relc local_spec" where
  "order_spec ys = (conservative_local_spec
       (λch x e d. relc_learn ch ys x e (forget_relc x d))
       (λch sc x d. forget_relc x d)
       (λch eo p d. relc_ret eo d)
       (λch ci p. [(fst p, ⊤)])
       (λch' ci d de. ⊤))
     (|ls_query := (λch. relc_qry), ls_branch := (λch b pol d. relc_branch_step b pol d)|)"

theorem order_spec_sound: "sound_local_spec 𝒢 gamma_relc (order_spec ys)"
proof -
  have special: "sound_special gamma_relc (λch sc x d. forget_relc x d)"
    unfolding sound_special_def
  proof (intro allI impI)
    fix ch d s sc x t
    assume "s ∈ γ (d :: relc)" "t ∈ special_step sc x s"
    then show "t ∈ γ (forget_relc x d)" by (cases sc) auto
  qed
  have base: "sound_local_spec 𝒢 gamma_relc (conservative_local_spec
       (λch x e d. relc_learn ch ys x e (forget_relc x d)) (λch sc x d. forget_relc x d)
       (λch eo p d. relc_ret eo d) (λch ci p. [(fst p, ⊤)]) (λch' ci d de. ⊤))"
    by (rule sound_conservative_local_spec[OF gamma_relc_mono _ special])
       (auto simp: sound_assign_def sound_return_def sound_enter_def relc_ret_def
          sound_combine_env_identity split: option.splits intro!: relc_learn_sound)
  show ?thesis
    unfolding order_spec_def
    by (intro sound_local_spec_update_branch[OF sound_local_spec_update_query[OF base]])
       (auto simp: sound_query_def sound_branch_def relc_branch_step_def relc_qry_sound)
qed

lemma single_entry_order_spec: "single_entry (order_spec ys)"
  by (simp add: single_entry_def order_spec_def conservative_local_spec_def)

end
```

Chapter 16

Voblint_Analysis_Int

16.1 Int_Warrowing

```
theory Int_Warrowing
  imports
    "Voblint_Domain.Int_Lattice"
    "Voblint_Analysis_Sign.Sign_Warrowing"
    "Voblint_Analysis_Parity.Parity_Warrowing"
    "Voblint_Analysis_Interval.Interval_Warrowing"
    "Voblint_Analysis_Congruence.Congruence_Warrowing"
begin
```

Composite widening and narrowing

Purely componentwise, with no reduced-product refinement attached to either operation. Widening already matches Goblint's own `~norefine:true` choice for `join`/`widen`. Narrowing is a deliberate divergence from Goblint's `IntDomTuple`, forced by the vendored TD solver's interface:

`narrow_ge: b <= a ==> b <= a △ b narrow_le: b <= a ==> a △ b <= a`

Suppose componentwise narrowing gives `b <= narrow_raw a b <= a` and refinement is then applied on top, Goblint-style: `refine mode (narrow_raw a b)`. Refinement is reductive (`Int_Refinement_Control.refine_reductive`), so the result stays `<= narrow_raw a b <= a` and `narrow_le` survives – but there is no general reason for `b <= refine mode (narrow_raw a b)` to hold. Refinement may push the result strictly below `b`.

The tempting escape is a stability argument: if `b` were already refinement-stable (`refine mode b = b`) and refinement is monotone, then `b = refine mode b <= refine mode (narrow_raw a b)` would follow from `b <= narrow_raw a b`. But that needs every value reaching narrowing to already be refinement-stable, and this carrier has no such invariant: `Int_Backward`'s own composite examples exercise values such as `STop × [-1,0] × PEven × top`, where `refine mode b ~= b`. Join and widening deliberately do not refine either (mirroring Goblint's `~norefine:true` for both), so a value reaching narrowing by way of a widened solver state is not guaranteed stable. Building narrowing on an invariant the type does not enforce would be unsound at the class instance, not merely imprecise.

Composite widening and narrowing therefore run no refinement at all, matching every other component's own choice (`Sign_Warrowing`, `Interval_Warrowing`, `Parity_Warrowing`, `Congruence_Warrowing`): each component widens/narrows on its own terms, and the composite record update runs no cross-component step afterward.

The lattice instance in `Int_Lattice` sits on the record scheme `'a int_dom_scheme`, so `widen`/`narrow` follow the same route and need `warrowing` on the scheme's `more` field too. `unit`, the `more` type of the closed `int_dom`, gets the trivial instance.

```
instantiation unit :: warrowing
begin
definition widen_unit :: "unit => unit => unit" where "(a ▽ b) = ()"
definition narrow_unit :: "unit => unit => unit" where "(a △ b) = ()"
instance by intro_classes simp_all
end

instantiation int_dom_ext :: ("{bounded_lattice, warrowing}") warrowing
begin

definition widen_int_dom_ext :: "'a int_dom_scheme => 'a int_dom_scheme => 'a int_dom_scheme" where
  "((a :: 'a int_dom_scheme) ▽ b) =
     int_dom.extend
       (int_dom.truncate
         (a(|
           int_sign := (int_sign a ▽ int_sign b),
           int_ivl := (int_ivl a ▽ int_ivl b),
           int_parity := (int_parity a ▽ int_parity b),
           int_congruence := (int_congruence a ▽ int_congruence b)
         |)))
       (int_dom.more a ▽ int_dom.more b)"

definition narrow_int_dom_ext :: "'a int_dom_scheme => 'a int_dom_scheme => 'a int_dom_scheme" where
  "((a :: 'a int_dom_scheme) △ b) =
     int_dom.extend
       (int_dom.truncate
         (a(|
           int_sign := (int_sign a △ (int_sign b)),
           int_ivl := (int_ivl a △ (int_ivl b)),
           int_parity := (int_parity a △ (int_parity b)),
           int_congruence := (int_congruence a △ (int_congruence b))
         |)))
       (int_dom.more a △ (int_dom.more b))"
```

```
lemma int_dom_extend_truncate_select [simp]:
  "int_sign (int_dom.extend (int_dom.truncate r) m) = int_sign r"
  "int_ivl (int_dom.extend (int_dom.truncate r) m) = int_ivl r"
  "int_parity (int_dom.extend (int_dom.truncate r) m) = int_parity r"
  "int_congruence (int_dom.extend (int_dom.truncate r) m) = int_congruence r"
  "int_dom.more (int_dom.extend (int_dom.truncate r) m) = m"
  by (simp_all add: int_dom.defs)

instance proof intro_classes
  fix a b :: "'a int_dom_scheme"
  assume ba: "b <= a"
  have hb: "int_sign b <= int_sign a" "int_ivl b <= int_ivl a"
           "int_parity b <= int_parity a" "int_congruence b <= int_congruence a"
           "int_dom.more b <= int_dom.more a"
    using ba by (simp_all add: less_eq_int_dom_ext_def)
  have s: "int_sign b <= (int_sign a Δ (int_sign b))"
    by (rule narrow_ge[OF hb(1)])
  have i: "int_ivl b <= (int_ivl a Δ (int_ivl b))"
    by (rule narrow_ge[OF hb(2)])
  have p: "int_parity b <= (int_parity a Δ (int_parity b))"
    by (rule narrow_ge[OF hb(3)])
  have c: "int_congruence b <= (int_congruence a Δ (int_congruence b))"
    by (rule narrow_ge[OF hb(4)])
  have m: "int_dom.more b <= (int_dom.more a Δ (int_dom.more b))"
    by (rule narrow_ge[OF hb(5)])
  show "b <= (a Δ b)"
    unfolding narrow_int_dom_ext_def less_eq_int_dom_ext_def
    using s i p c m by simp
next
  fix a b :: "'a int_dom_scheme"
  assume ba: "b <= a"
  have hb: "int_sign b <= int_sign a" "int_ivl b <= int_ivl a"
           "int_parity b <= int_parity a" "int_congruence b <= int_congruence a"
           "int_dom.more b <= int_dom.more a"
    using ba by (simp_all add: less_eq_int_dom_ext_def)
  have s: "(int_sign a Δ (int_sign b)) <= int_sign a"
    by (rule narrow_le[OF hb(1)])
  have i: "(int_ivl a Δ (int_ivl b)) <= int_ivl a"
    by (rule narrow_le[OF hb(2)])
  have p: "(int_parity a Δ (int_parity b)) <= int_parity a"
    by (rule narrow_le[OF hb(3)])
  have c: "(int_congruence a Δ (int_congruence b)) <= int_congruence a"
    by (rule narrow_le[OF hb(4)])
  have m: "(int_dom.more a Δ (int_dom.more b)) <= int_dom.more a"
    by (rule narrow_le[OF hb(5)])
  show "(a Δ b) <= a"
    unfolding narrow_int_dom_ext_def less_eq_int_dom_ext_def
    using s i p c m by simp
next
  fix a b :: "'a int_dom_scheme"
  have s: "int_sign a <= (int_sign a ∇ int_sign b)"
    by (rule widen_ge1)
  have i: "int_ivl a <= (int_ivl a ∇ int_ivl b)"
    by (rule widen_ge1)
  have p: "int_parity a <= (int_parity a ∇ int_parity b)"
    by (rule widen_ge1)
  have c: "int_congruence a <= (int_congruence a ∇ int_congruence b)"
    by (rule widen_ge1)
  have m: "int_dom.more a <= (int_dom.more a ∇ int_dom.more b)"
    by (rule widen_ge1)
  show "a <= (a ∇ b)"
    unfolding widen_int_dom_ext_def less_eq_int_dom_ext_def
    using s i p c m by simp
next
  fix a b :: "'a int_dom_scheme"
  have s: "int_sign b <= (int_sign a ∇ int_sign b)"
    by (rule widen_ge2)
  have i: "int_ivl b <= (int_ivl a ∇ int_ivl b)"
    by (rule widen_ge2)
  have p: "int_parity b <= (int_parity a ∇ int_parity b)"
    by (rule widen_ge2)
  have c: "int_congruence b <= (int_congruence a ∇ int_congruence b)"
    by (rule widen_ge2)
  have m: "int_dom.more b <= (int_dom.more a ∇ int_dom.more b)"
    by (rule widen_ge2)
```

```
  show "b <= (a ▽ b)"
    unfolding widen_int_dom_ext_def less_eq_int_dom_ext_def
    using s i p c m by simp
qed

end
```

Numeric domain instance

`numeric_domain` extends `executable_domain`, which includes the solver's `warrowing`, so the composite instance asks the same of the `more` field as the widening and narrowing above.

```
instantiation int_dom_ext ::
  ("{bounded_lattice, warrowing}") numeric_domain
begin

definition gamma_abs_int_dom_ext [simp]:
  "numeric_domain_class.gamma (d :: 'a int_dom_scheme) = gamma_int_dom d"

definition is_empty_int_dom_ext [simp]:
  "is_empty (d :: 'a int_dom_scheme) = is_bottom_int_dom d"

definition to_string_int_dom_ext [simp]:
  "to_string (d :: 'a int_dom_scheme) = string_of_int_dom d"

instance
proof intro_classes
  show "numeric_domain_class.gamma (bot :: 'a int_dom_scheme) = {}"
    by (simp add: gamma_int_dom_def bot_int_dom_ext_def
          bot_sign_def bot_ivl_def bot_parity_def)
next
  show "numeric_domain_class.gamma (top :: 'a int_dom_scheme) = UNIV"
    by (simp add: gamma_int_dom_def top_int_dom_ext_def
          gamma_sign_top gamma_ivl_top top_ivl_def gamma_parity_top)
next
  fix a b :: "'a int_dom_scheme"
  show "a ≤ b ⟹ numeric_domain_class.gamma a ⊆ numeric_domain_class.gamma b"
    by (simp add: gamma_int_dom_mono)
next
  fix a :: "'a int_dom_scheme"
  show "is_empty a ⟷ numeric_domain_class.gamma a = {}"
    by (simp add: is_bottom_int_dom_correct)
qed

end

end
```

16.2 Int_Refinement

```
theory Int_Refinement
  imports
    Int_Warrowing
    "HOL-Library.While_Combinator"
begin
```

Closed-world integer-domain refinement

Refinement exchanges a fixed vocabulary of facts between the components of the concrete integer-domain record. Reduction-step laws isolate progressive composition from the record representation.

```
definition int_reduction_step :: "(int_dom => int_dom) => bool" where
  "int_reduction_step step ⟷
     (∀d. gamma_int_dom (step d) = gamma_int_dom d) ∧
     (∀d. step d ≤ d) ∧
     mono step"

lemma int_reduction_step_exact:
  assumes "int_reduction_step step"
  shows "gamma_int_dom (step d) = gamma_int_dom d"
  using assms unfolding int_reduction_step_def by blast

lemma int_reduction_step_reductive:
  assumes "int_reduction_step step"
  shows "step d ≤ d"
  using assms unfolding int_reduction_step_def by blast

lemma int_reduction_step_mono:
  assumes "int_reduction_step step"
  shows "mono step"
  using assms unfolding int_reduction_step_def by blast

lemma int_reduction_step_id [simp]:
  "int_reduction_step id"
  unfolding int_reduction_step_def mono_def by simp

lemma int_reduction_step_comp:
  assumes f: "int_reduction_step f"
      and g: "int_reduction_step g"
  shows "int_reduction_step (g o f)"
proof (unfold int_reduction_step_def, intro conjI allI)
  fix d
  show "gamma_int_dom ((g o f) d) = gamma_int_dom d"
    using int_reduction_step_exact[OF g, of "f d"]
          int_reduction_step_exact[OF f, of d]
    by simp
next
  fix d
  have "g (f d) ≤ f d"
    by (rule int_reduction_step_reductive[OF g])
  also have "f d ≤ d"
    by (rule int_reduction_step_reductive[OF f])
  finally show "(g o f) d ≤ d"
    by simp
next
  show "mono (g o f)"
    using int_reduction_step_mono[OF f]
          int_reduction_step_mono[OF g]
    unfolding mono_def by auto
qed

fun apply_reduction_steps ::
  "(int_dom => int_dom) list => int_dom => int_dom" where
  "apply_reduction_steps [] d = d"
| "apply_reduction_steps (step # steps) d =
     apply_reduction_steps steps (step d)"

lemma int_reduction_steps:
  assumes "∀step ∈ set steps. int_reduction_step step"
  shows "int_reduction_step (apply_reduction_steps steps)"
  using assms
proof (induction steps)
  case Nil
  then show ?case
```

```
    unfolding int_reduction_step_def mono_def by simp
next
  case (Cons step steps)
  have step: "int_reduction_step step"
    using Cons.prems by simp
  have rest: "int_reduction_step (apply_reduction_steps steps)"
    using Cons.IH Cons.prems by simp
  have valid: "int_reduction_step (apply_reduction_steps steps o step)"
    by (rule int_reduction_step_comp[OF step rest])
  from valid show ?case
    unfolding int_reduction_step_def mono_def by simp
qed
```

Refinement facts

Two kinds of fact travel between components: an interval and a congruence. `interval_sign_fact` and `interval_parity_fact` read off the sign and parity an interval implies; the lemmas below show each over-approximates the interval and is monotone.

```
type_synonym interval_fact = ivl
type_synonym congruence_fact = congruence

definition interval_fact_of_ivl :: "ivl => interval_fact" where
  "interval_fact_of_ivl i = i"

definition interval_sign_fact :: "interval_fact => sign" where
  "interval_sign_fact i =
     (if is_bottom_ivl i then SBot
      else case i of Ivl l u =>
        if u < Fin 0 then SNeg
        else if u = Fin 0 then
          (if l = Fin 0 then SZero else SNonPos)
        else if Fin 0 < l then SPos
        else if l = Fin 0 then SNonNeg
        else STop)"

definition interval_parity_fact :: "interval_fact => parity" where
  "interval_parity_fact i =
     (if is_bottom_ivl i then PBot
      else case i of
        Ivl (Fin l) (Fin u) =>
          if l = u then if even l then PEven else POdd else PTop
      | _ => PTop)"

lemma interval_sign_fact_sound:
  "gamma_ivl i ⊆ gamma_sign (interval_sign_fact i)"
proof
  fix n
  assume n: "n ∈ gamma_ivl i"
  obtain l u where i: "i = Ivl l u"
    by (rule ivl_exhaustE)
  from n have bounds: "l ≤ Fin n" "Fin n ≤ u"
    unfolding i by auto
  from n have nonempty: "¬ is_bottom_ivl i"
    using is_bottom_ivl_correct by auto
  show "n ∈ gamma_sign (interval_sign_fact i)"
    unfolding interval_sign_fact_def i
    using bounds nonempty
    by (cases l; cases u)
       (auto simp: is_bottom_ivl_def less_eint_def
          split: if_splits)
qed

lemma interval_parity_fact_sound:
  "gamma_ivl i ⊆ gamma_parity (interval_parity_fact i)"
proof
  fix n
  assume n: "n ∈ gamma_ivl i"
  obtain l u where i: "i = Ivl l u"
    by (rule ivl_exhaustE)
  from n have bounds: "l ≤ Fin n" "Fin n ≤ u"
    unfolding i by auto
  from n have nonempty: "¬ is_bottom_ivl i"
    using is_bottom_ivl_correct by auto
  show "n ∈ gamma_parity (interval_parity_fact i)"
    unfolding interval_parity_fact_def i
```

```
    using bounds nonempty
    by (cases l; cases u)
       (auto simp: is_bottom_ivl_def
          split: if_splits)

qed

lemma interval_fact_of_sign_mono:
  assumes "s ≤ t"
  shows "interval_fact_of_sign s ≤ interval_fact_of_sign t"
  using assms
  by (cases s; cases t)
     (simp_all add: less_eq_sign_def less_eq_ivl_def
        bot_ivl_def top_ivl_def ivl_top_def)

lemma interval_sign_fact_mono:
  assumes "i ≤ j"
  shows "interval_sign_fact i ≤ interval_sign_fact j"
proof -
  obtain l1 u1 where i: "i = Ivl l1 u1"
    by (rule ivl_exhaustE)
  obtain l2 u2 where j: "j = Ivl l2 u2"
    by (rule ivl_exhaustE)
  show ?thesis
    using assms
    unfolding i j interval_sign_fact_def
    by (cases l1; cases u1; cases l2; cases u2)
       (auto simp: less_eq_ivl_def is_bottom_ivl_def
          less_eint_def less_eq_sign_def
          split: if_splits)
qed

lemma interval_parity_fact_mono:
  assumes "i ≤ j"
  shows "interval_parity_fact i ≤ interval_parity_fact j"
proof -
  obtain l1 u1 where i: "i = Ivl l1 u1"
    by (rule ivl_exhaustE)
  obtain l2 u2 where j: "j = Ivl l2 u2"
    by (rule ivl_exhaustE)
  show ?thesis
    using assms
    unfolding i j interval_parity_fact_def
    by (cases l1; cases u1; cases l2; cases u2)
       (auto simp: less_eq_ivl_def is_bottom_ivl_def
          less_eq_parity_def
          split: if_splits)

qed
```

Interval-fact fan-out

`refine_interval` intersects the sign and interval components into one interval fact and meets every interval-readable component with it. The step preserves the concretization, only shrinks the record and is monotone, so it is an `int_reduction_step`.

```
definition refine_sign_with_interval ::
  "interval_fact => sign => sign" where
  "refine_sign_with_interval fct s =
     interval_sign_fact fct ⊓ s"

definition refine_ivl_with_interval ::
  "interval_fact => ivl => ivl" where
  "refine_ivl_with_interval fct i = intersect_ivl fct i"

definition refine_parity_with_interval ::
  "interval_fact => parity => parity" where
  "refine_parity_with_interval fct p =
     interval_parity_fact fct ⊓ p"

definition interval_fact_of_int_dom :: "int_dom => interval_fact" where
  "interval_fact_of_int_dom d =
     intersect_ivl
       (interval_fact_of_sign (int_sign d))
       (interval_fact_of_ivl (int_ivl d))"
```

```
definition refine_interval :: "int_dom => int_dom" where
  "refine_interval d =
     (let fct = interval_fact_of_int_dom d
      in d⦇
           int_sign := refine_sign_with_interval fct (int_sign d),
           int_ivl := refine_ivl_with_interval fct (int_ivl d),
           int_parity := refine_parity_with_interval fct (int_parity d)
         ⦈)"

lemma gamma_refine_sign_with_interval [simp]:
  "gamma_sign (refine_sign_with_interval fct s) =
   gamma_sign (interval_sign_fact fct) ∩ gamma_sign s"
  by (simp add: refine_sign_with_interval_def)

lemma gamma_refine_ivl_with_interval:
  "gamma_ivl (refine_ivl_with_interval fct i) =
   gamma_ivl fct ∩ gamma_ivl i"
  by (simp only: refine_ivl_with_interval_def gamma_intersect_ivl_exact)

lemma gamma_refine_parity_with_interval [simp]:
  "gamma_parity (refine_parity_with_interval fct p) =
   gamma_parity (interval_parity_fact fct) ∩ gamma_parity p"
  by (simp add: refine_parity_with_interval_def)

lemma refine_sign_with_interval_le:
  "refine_sign_with_interval fct s ≤ s"
  unfolding refine_sign_with_interval_def
  by (rule inf_le2)

lemma refine_ivl_with_interval_le:
  "refine_ivl_with_interval fct i ≤ i"
  unfolding refine_ivl_with_interval_def
  by (rule intersect_ivl_le2)

lemma refine_parity_with_interval_le:
  "refine_parity_with_interval fct p ≤ p"
  unfolding refine_parity_with_interval_def
  by (rule inf_le2)

lemma refine_sign_with_interval_mono:
  assumes "fct1 ≤ fct2" and "s1 ≤ s2"
  shows
    "refine_sign_with_interval fct1 s1 ≤
     refine_sign_with_interval fct2 s2"
  unfolding refine_sign_with_interval_def
  by (rule inf_mono;
      use assms interval_sign_fact_mono in blast)

lemma refine_ivl_with_interval_mono:
  assumes "fct1 ≤ fct2" and "i1 ≤ i2"
  shows
    "refine_ivl_with_interval fct1 i1 ≤
     refine_ivl_with_interval fct2 i2"
  unfolding refine_ivl_with_interval_def
  by (rule intersect_ivl_mono; fact)

lemma refine_parity_with_interval_mono:
  assumes "fct1 ≤ fct2" and "p1 ≤ p2"
  shows
    "refine_parity_with_interval fct1 p1 ≤
     refine_parity_with_interval fct2 p2"
  unfolding refine_parity_with_interval_def
  by (rule inf_mono;
      use assms interval_parity_fact_mono in blast)

lemma gamma_interval_fact_of_int_dom:
  "gamma_ivl (interval_fact_of_int_dom d) =
   gamma_sign (int_sign d) ∩ gamma_ivl (int_ivl d)"
  unfolding interval_fact_of_int_dom_def interval_fact_of_ivl_def
  by (simp only: gamma_intersect_ivl_exact gamma_interval_fact_of_sign)

lemma interval_fact_of_int_dom_mono:
  assumes "d ≤ e"
  shows "interval_fact_of_int_dom d ≤ interval_fact_of_int_dom e"
proof -
  from assms have components:
```

```
    "int_sign d ≤ int_sign e"
    "int_ivl d ≤ int_ivl e"
    by (simp_all add: less_eq_int_dom_ext_def)
  show ?thesis
    unfolding interval_fact_of_int_dom_def interval_fact_of_ivl_def
    by (rule intersect_ivl_mono)
       (use components interval_fact_of_sign_mono in auto)
qed

lemma refine_interval_exact:
  "gamma_int_dom (refine_interval d) = gamma_int_dom d"
proof -
  let ?fct = "interval_fact_of_int_dom d"
  have fct:
    "gamma_ivl ?fct =
     gamma_sign (int_sign d) ∩ gamma_ivl (int_ivl d)"
    by (rule gamma_interval_fact_of_int_dom)
  have sign_cover:
    "gamma_ivl ?fct ⊆
     gamma_sign (interval_sign_fact ?fct)"
    by (rule interval_sign_fact_sound)
  have parity_cover:
    "gamma_ivl ?fct ⊆
     gamma_parity (interval_parity_fact ?fct)"
    by (rule interval_parity_fact_sound)
  show ?thesis
    unfolding refine_interval_def gamma_int_dom_def Let_def
    using fct sign_cover parity_cover
    by (auto simp: gamma_refine_ivl_with_interval)
qed

lemma refine_interval_reductive:
  "refine_interval d ≤ d"
  unfolding refine_interval_def Let_def
  by (simp add: less_eq_int_dom_ext_def
        refine_sign_with_interval_le
        refine_ivl_with_interval_le
        refine_parity_with_interval_le)

lemma refine_interval_mono:
  "mono refine_interval"
proof (rule monoI)
  fix d e :: int_dom
  assume de: "d ≤ e"
  from de have components:
    "int_sign d ≤ int_sign e"
    "int_ivl d ≤ int_ivl e"
    "int_parity d ≤ int_parity e"
    "int_congruence d ≤ int_congruence e"
    by (simp_all add: less_eq_int_dom_ext_def)
  have fct_le:
    "interval_fact_of_int_dom d ≤ interval_fact_of_int_dom e"
    by (rule interval_fact_of_int_dom_mono[OF de])
  show "refine_interval d ≤ refine_interval e"
    unfolding refine_interval_def Let_def
    by (simp add: less_eq_int_dom_ext_def
          refine_sign_with_interval_mono[OF fct_le components(1)]
          refine_ivl_with_interval_mono[OF fct_le components(2)]
          refine_parity_with_interval_mono[OF fct_le components(3)]
          components(4))
qed

lemma int_reduction_step_refine_interval:
  "int_reduction_step refine_interval"
  unfolding int_reduction_step_def
  using refine_interval_exact refine_interval_reductive refine_interval_mono
  by blast
```

Congruence-fact fan-out

`refine_congruence` first sharpens the congruence by the parity, then uses the result to round the interval bounds to the nearest class members (`congruence_lower_bound`, `congruence_upper_bound`) and to sharpen the parity. Like the interval fan-out it is an `int_reduction_step`.

```
definition congruence_fact_of_congruence ::
  "congruence => congruence_fact" where
```

```
  "congruence_fact_of_congruence c = c"

definition congruence_fact_of_int_dom ::
  "int_dom => congruence_fact" where
  "congruence_fact_of_int_dom d =
     restrict_congruence_by_parity
       (int_parity d)
       (congruence_fact_of_congruence (int_congruence d))"

fun congruence_lower_bound :: "int => int => eint => eint" where
  "congruence_lower_bound c m MinInf = MinInf"
| "congruence_lower_bound c m (Fin l) =
     Fin (l + ((c - l) mod m))"
| "congruence_lower_bound c m PlusInf = PlusInf"

fun congruence_upper_bound :: "int => int => eint => eint" where
  "congruence_upper_bound c m MinInf = MinInf"
| "congruence_upper_bound c m (Fin u) =
     Fin (u - ((u - c) mod m))"
| "congruence_upper_bound c m PlusInf = PlusInf"

fun refine_ivl_with_congruence_rep ::
  "congruence_rep => ivl => ivl" where
  "refine_ivl_with_congruence_rep None i = bot"
| "refine_ivl_with_congruence_rep (Some (c, m)) (Ivl l u) =
     (if m = 0 then
        intersect_ivl (Ivl (Fin c) (Fin c)) (Ivl l u)
      else
        intersect_ivl
          (mk_ivl
            (congruence_lower_bound c m l)
            (congruence_upper_bound c m u))
          (Ivl l u))"

definition refine_ivl_with_congruence ::
  "congruence_fact => ivl => ivl" where
  "refine_ivl_with_congruence fct i =
     refine_ivl_with_congruence_rep (Rep_congruence fct) i"

fun parity_fact_of_congruence_rep ::
  "congruence_rep => parity" where
  "parity_fact_of_congruence_rep None = PBot"
| "parity_fact_of_congruence_rep (Some (c, m)) =
     (if m = 0 ∨ even m then
        if even c then PEven else POdd
      else PTop)"

definition parity_fact_of_congruence ::
  "congruence_fact => parity" where
  "parity_fact_of_congruence fct =
     parity_fact_of_congruence_rep (Rep_congruence fct)"

definition refine_parity_with_congruence ::
  "congruence_fact => parity => parity" where
  "refine_parity_with_congruence fct p =
     parity_fact_of_congruence fct ⊓ p"

definition refine_congruence_with_congruence ::
  "congruence_fact => congruence => congruence" where
  "refine_congruence_with_congruence fct c = fct"

lemma gamma_congruence_fact_of_int_dom [simp]:
  "gamma_congruence (congruence_fact_of_int_dom d) =
   gamma_parity (int_parity d) ∩
   gamma_congruence (int_congruence d)"
  unfolding congruence_fact_of_int_dom_def
    congruence_fact_of_congruence_def
  by simp

lemma congruence_fact_of_int_dom_mono:
  assumes de: "d <= e"
  shows
    "congruence_fact_of_int_dom d <=
     congruence_fact_of_int_dom e"
proof -
  from de have parity:
```

```
    "int_parity d <= int_parity e"
    by (simp add: less_eq_int_dom_ext_def)
  from de have congruence:
    "int_congruence d <= int_congruence e"
    by (simp add: less_eq_int_dom_ext_def)
  have parity_gamma:
    "gamma_parity (int_parity d) ⊆
     gamma_parity (int_parity e)"
    using parity
    unfolding less_eq_parity_def
    by (rule gamma_parity_mono)
  have congruence_gamma:
    "gamma_congruence (int_congruence d) ⊆
     gamma_congruence (int_congruence e)"
    using congruence
    unfolding less_eq_congruence_iff_gamma .
  show ?thesis
    unfolding less_eq_congruence_iff_gamma
    using parity_gamma congruence_gamma by auto
qed

lemma congruence_fact_of_int_dom_le_congruence:
  "congruence_fact_of_int_dom d <= int_congruence d"
  unfolding less_eq_congruence_iff_gamma
  by auto

lemma congruence_lower_bound_le:
  fixes c m l n :: int
  assumes modulus: "0 < m"
      and lower: "l <= n"
      and congruent: "m dvd n - c"
  shows "l + ((c - l) mod m) <= n"
proof -
  have residues:
    "(c - l) mod m = (n - l) mod m"
    using congruent by (simp add: mod_eq_dvd_iff dvd_diff_commute)
  have nonnegative: "0 <= n - l"
    using lower by simp
  have remainder: "(n - l) mod m <= n - l"
    by (rule zmod_le_nonneg_dividend[OF nonnegative])
  from residues remainder show ?thesis by linarith
qed

lemma le_congruence_upper_bound:
  fixes c m n u :: int
  assumes modulus: "0 < m"
      and upper: "n <= u"
      and congruent: "m dvd n - c"
  shows "n <= u - ((u - c) mod m)"
proof -
  have residues:
    "(u - c) mod m = (u - n) mod m"
    using congruent by (simp add: mod_eq_dvd_iff)
  have nonnegative: "0 <= u - n"
    using upper by simp
  have remainder: "(u - n) mod m <= u - n"
    by (rule zmod_le_nonneg_dividend[OF nonnegative])
  from residues remainder show ?thesis by linarith
qed

lemma refine_ivl_with_congruence_cover:
  "gamma_ivl i ∩ gamma_congruence fct ⊆
   gamma_ivl (refine_ivl_with_congruence fct i)"
proof
  fix n
  assume member:
    "n ∈ gamma_ivl i ∩ gamma_congruence fct"
  obtain l u where i: "i = Ivl l u"
    by (rule ivl_exhaustE)
  have normalized:
    "normalized_congruence_rep (Rep_congruence fct)"
    by simp
  show
    "n ∈ gamma_ivl (refine_ivl_with_congruence fct i)"
  proof (cases "Rep_congruence fct")
    case None
```

```
    with member show ?thesis
      unfolding gamma_congruence_def by simp
  next
    case (Some cm)
    obtain c m where cm: "cm = (c, m)"
      by (cases cm)
    have rep: "Rep_congruence fct = Some (c, m)"
      using Some cm by simp
    have normalized':
      "normalized_congruence_rep (Some (c, m))"
      using normalized unfolding rep .
    from member have in_i: "n ∈ gamma_ivl i"
      by simp
    from member have congruent: "m dvd n - c"
      unfolding gamma_congruence_def rep by simp
    show ?thesis
    proof (cases "m = 0")
      case True
      from congruent True have n: "n = c" by simp
      have in_point:
        "n ∈ gamma_ivl (Ivl (Fin c) (Fin c))"
        using n by simp
      have in_i': "n ∈ gamma_ivl (Ivl l u)"
        using in_i unfolding i .
      have refined:
        "refine_ivl_with_congruence fct i =
         intersect_ivl (Ivl (Fin c) (Fin c)) (Ivl l u)"
        unfolding refine_ivl_with_congruence_def rep i
        using True by simp
      show ?thesis
        unfolding refined
        by (rule intersect_ivl_gamma[OF in_point in_i'])
    next
      case False
      have modulus: "0 < m"
        by (rule normalized_congruence_rep_nonzero[
              OF normalized' False])
      have lower:
        "congruence_lower_bound c m l <= Fin n"
        using in_i congruent modulus
        unfolding i
        by (cases l)
           (auto intro: congruence_lower_bound_le)
      have upper:
        "Fin n <= congruence_upper_bound c m u"
        using in_i congruent modulus
        unfolding i
        by (cases u)
           (auto intro: le_congruence_upper_bound)
      have in_hull:
        "n ∈ gamma_ivl
          (mk_ivl
            (congruence_lower_bound c m l)
            (congruence_upper_bound c m u))"
        using lower upper
        by (simp only: gamma_mk_ivl gamma_ivl.simps
            mem_Collect_eq)
      have in_i': "n ∈ gamma_ivl (Ivl l u)"
        using in_i unfolding i .
      have refined:
        "refine_ivl_with_congruence fct i =
         intersect_ivl
           (mk_ivl
             (congruence_lower_bound c m l)
             (congruence_upper_bound c m u))
           (Ivl l u)"
        unfolding refine_ivl_with_congruence_def rep i
        using False by simp
      show ?thesis
        unfolding refined
        by (rule intersect_ivl_gamma[OF in_hull in_i'])
    qed
  qed
qed

lemma refine_ivl_with_congruence_le:
```

```
  "refine_ivl_with_congruence fct i <= i"
proof (cases "Rep_congruence fct")
  case None
  then show ?thesis
    unfolding refine_ivl_with_congruence_def by simp
next
  case (Some cm)
  obtain c m where cm: "cm = (c, m)"
    by (cases cm)
  have rep: "Rep_congruence fct = Some (c, m)"
    using Some cm by simp
  obtain l u where i: "i = Ivl l u"
    by (rule ivl_exhaustE)
  show ?thesis
  proof (cases "m = 0")
    case True
    have refined:
      "refine_ivl_with_congruence fct i =
       intersect_ivl (Ivl (Fin c) (Fin c)) i"
      unfolding refine_ivl_with_congruence_def rep i
      using True by simp
    show ?thesis
      unfolding refined by (rule intersect_ivl_le2)
  next
    case False
    have refined:
      "refine_ivl_with_congruence fct i =
       intersect_ivl
         (mk_ivl
           (congruence_lower_bound c m l)
           (congruence_upper_bound c m u))
         i"
      unfolding refine_ivl_with_congruence_def rep i
      using False by simp
    show ?thesis
      unfolding refined by (rule intersect_ivl_le2)
  qed
qed

lemma normalize_refine_ivl_with_congruence [simp]:
  "normalize_ivl (refine_ivl_with_congruence fct i) =
   refine_ivl_with_congruence fct i"
  unfolding refine_ivl_with_congruence_def
  by (cases "Rep_congruence fct"; cases i)
     (auto simp only:
        refine_ivl_with_congruence_rep.simps
        normalize_ivl_bot
        normalize_ivl_intersect_ivl
        split: prod.splits if_splits)

lemma congruence_lower_bound_congruent:
  fixes c m l :: int
  shows
    "m dvd
     (l + ((c - l) mod m)) - c"
proof -
  have base:
    "m dvd (c - l) - ((c - l) mod m)"
    by (rule dvd_minus_mod)
  have negated:
    "m dvd - ((c - l) - ((c - l) mod m))"
    by (subst dvd_minus_iff) (rule base)
  have equal:
    "- ((c - l) - ((c - l) mod m)) =
     (l + ((c - l) mod m)) - c"
    by linarith
  from negated show ?thesis
    unfolding equal .
qed

lemma congruence_upper_bound_congruent:
  fixes c m u :: int
  shows
    "m dvd
     (u - ((u - c) mod m)) - c"
proof -
```

```
  have base:
    "m dvd (u - c) - ((u - c) mod m)"
    by (rule dvd_minus_mod)
  have equal:
    "(u - c) - ((u - c) mod m) =
     (u - ((u - c) mod m)) - c"
    by linarith
  from base show ?thesis
    unfolding equal .
qed

lemma congruence_lower_bound_mono:
  fixes c1 m1 c2 m2 :: int
    and l1 l2 :: eint
  assumes m1: "0 < m1"
      and m2: "0 < m2"
      and moduli: "m2 dvd m1"
      and residues: "m2 dvd c1 - c2"
      and bounds: "l2 <= l1"
  shows
    "congruence_lower_bound c2 m2 l2 <=
     congruence_lower_bound c1 m1 l1"
proof (cases l1)
  case MinInf
  then show ?thesis
    using bounds by (cases l2) simp_all
next
  case (Fin a)
  have l1: "l1 = Fin a"
    by (rule Fin)
  show ?thesis
  proof (cases l2)
    case MinInf
    then show ?thesis by simp
  next
    case (Fin b)
    have ba: "b <= a"
      using bounds l1 Fin by simp
    let ?n = "a + ((c1 - a) mod m1)"
    have a_le_n: "a <= ?n"
      using pos_mod_sign[OF m1, of "c1 - a"]
      by linarith
    have congruent1: "m1 dvd ?n - c1"
      by (rule congruence_lower_bound_congruent)
    have congruent2a: "m2 dvd ?n - c1"
      by (rule dvd_trans[OF moduli congruent1])
    have congruent2: "m2 dvd ?n - c2"
    proof -
      have combined:
        "m2 dvd (?n - c1) + (c1 - c2)"
        by (rule dvd_add[OF congruent2a residues])
      have equal:
        "(?n - c1) + (c1 - c2) = ?n - c2"
        by linarith
      from combined show ?thesis
        unfolding equal .
    qed
    have b_le_n: "b <= ?n"
      using ba a_le_n by linarith
    have
      "b + ((c2 - b) mod m2) <= ?n"
      by (rule congruence_lower_bound_le[
            OF m2 b_le_n congruent2])
    with l1 Fin show ?thesis by simp
  next
    case PlusInf
    with bounds l1 show ?thesis by simp
  qed
next
  case PlusInf
  then show ?thesis by simp
qed

lemma congruence_upper_bound_mono:
  fixes c1 m1 c2 m2 :: int
    and u1 u2 :: eint
```

```
  assumes m1: "0 < m1"
      and m2: "0 < m2"
      and moduli: "m2 dvd m1"
      and residues: "m2 dvd c1 - c2"
      and bounds: "u1 <= u2"
  shows
    "congruence_upper_bound c1 m1 u1 <=
     congruence_upper_bound c2 m2 u2"
proof (cases u1)
  case MinInf
  then show ?thesis by simp
next
  case (Fin a)
  have u1: "u1 = Fin a"
    by (rule Fin)
  show ?thesis
  proof (cases u2)
    case MinInf
    with bounds u1 show ?thesis by simp
  next
    case (Fin b)
    have ab: "a <= b"
      using bounds u1 Fin by simp
    let ?n = "a - ((a - c1) mod m1)"
    have n_le_a: "?n <= a"
      using pos_mod_sign[OF m1, of "a - c1"]
      by linarith
    have congruent1: "m1 dvd ?n - c1"
      by (rule congruence_upper_bound_congruent)
    have congruent2a: "m2 dvd ?n - c1"
      by (rule dvd_trans[OF moduli congruent1])
    have congruent2: "m2 dvd ?n - c2"
    proof -
      have combined:
        "m2 dvd (?n - c1) + (c1 - c2)"
        by (rule dvd_add[OF congruent2a residues])
      have equal:
        "(?n - c1) + (c1 - c2) = ?n - c2"
        by linarith
      from combined show ?thesis
        unfolding equal .
    qed
    have n_le_b: "?n <= b"
      using n_le_a ab by linarith
    have
      "?n <= b - ((b - c2) mod m2)"
      by (rule le_congruence_upper_bound[
            OF m2 n_le_b congruent2])
    with u1 Fin show ?thesis by simp
  next
    case PlusInf
    then show ?thesis by simp
  qed
next
  case PlusInf
  then show ?thesis
    using bounds by (cases u2) simp_all
qed

lemma intersect_ivl_eq_bot_if_disjoint:
  assumes
    "gamma_ivl a ∩ gamma_ivl b = {}"
  shows "intersect_ivl a b = bot"
proof -
  have empty:
    "gamma_ivl (intersect_ivl a b) = {}"
    using assms gamma_intersect_ivl_exact by simp
  have bottom:
    "is_bottom_ivl (intersect_ivl a b)"
    using empty is_bottom_ivl_correct by blast
  have normalized:
    "normalize_ivl (intersect_ivl a b) =
     intersect_ivl a b"
    by simp
  show ?thesis
    using normalize_ivl_bot_case[OF bottom] normalized
```

```
    by simp
qed

lemma refine_ivl_with_congruence_mono:
  assumes "fct1 <= fct2" "i1 <= i2"
  shows
    "refine_ivl_with_congruence fct1 i1 <=
     refine_ivl_with_congruence fct2 i2"
proof -
  have rep_le:
    "congruence_le_rep
       (Rep_congruence fct1)
       (Rep_congruence fct2)"
    using assms(1)
    unfolding less_eq_congruence_def congruence_le_def .
  have normalized1:
    "normalized_congruence_rep (Rep_congruence fct1)"
    by simp
  have normalized2:
    "normalized_congruence_rep (Rep_congruence fct2)"
    by simp
  obtain l1 u1 where i1: "i1 = Ivl l1 u1"
    by (rule ivl_exhaustE)
  obtain l2 u2 where i2: "i2 = Ivl l2 u2"
    by (rule ivl_exhaustE)
  have bounds: "l2 <= l1" "u1 <= u2"
    using assms(2)
    unfolding i1 i2 less_eq_ivl_def
    by simp_all
  have singleton_le_iff:
    "Ivl (Fin n) (Fin n) <= i ⟷
     n ∈ gamma_ivl i"
    for n i
    by (cases i) (simp add: less_eq_ivl_def)
  show ?thesis
  proof (cases "Rep_congruence fct1")
    case None
    then show ?thesis
      unfolding refine_ivl_with_congruence_def by simp
  next
    case (Some cm1)
    obtain c1 m1 where cm1: "cm1 = (c1, m1)"
      by (cases cm1)
    have rep1: "Rep_congruence fct1 = Some (c1, m1)"
      using Some cm1 by simp
    show ?thesis
    proof (cases "Rep_congruence fct2")
      case None
      with rep_le rep1 show ?thesis by simp
    next
      case (Some cm2)
      obtain c2 m2 where cm2: "cm2 = (c2, m2)"
        by (cases cm2)
      have rep2: "Rep_congruence fct2 = Some (c2, m2)"
        using Some cm2 by simp
      have relation:
        "m2 dvd m1" "m2 dvd c1 - c2"
        using rep_le unfolding rep1 rep2
        by simp_all
      have normalized1':
        "normalized_congruence_rep (Some (c1, m1))"
        using normalized1 unfolding rep1 .
      have normalized2':
        "normalized_congruence_rep (Some (c2, m2))"
        using normalized2 unfolding rep2 .
      show ?thesis
      proof (cases "m1 = 0")
        case True
        show ?thesis
        proof (cases "m2 = 0")
          case True2: True
          have residues: "c1 = c2"
            using relation(2) True2 by simp
          have point_le:
            "Ivl (Fin c1) (Fin c1) <=
             Ivl (Fin c2) (Fin c2)"
```

```
      using residues by simp
    have refined1:
      "refine_ivl_with_congruence fct1 i1 =
       intersect_ivl (Ivl (Fin c1) (Fin c1)) i1"
      unfolding refine_ivl_with_congruence_def rep1 i1
      using True by simp
    have refined2:
      "refine_ivl_with_congruence fct2 i2 =
       intersect_ivl (Ivl (Fin c2) (Fin c2)) i2"
      unfolding refine_ivl_with_congruence_def rep2 i2
      using True2 by simp
    show ?thesis
      unfolding refined1 refined2
      by (rule intersect_ivl_mono[OF point_le assms(2)])
  next
    case False
    show ?thesis
    proof (cases "c1 ∈ gamma_ivl i1")
      case False
      have empty:
        "gamma_ivl (Ivl (Fin c1) (Fin c1)) ∩
         gamma_ivl i1 = {}"
        using False by auto
      have source_refined:
        "refine_ivl_with_congruence fct1 i1 =
         intersect_ivl
           (Ivl (Fin c1) (Fin c1)) i1"
        unfolding refine_ivl_with_congruence_def rep1 i1
        using True by simp
      have source_bottom:
        "refine_ivl_with_congruence fct1 i1 = bot"
        unfolding source_refined
        by (rule intersect_ivl_eq_bot_if_disjoint[OF empty])
      then show ?thesis by simp
    next
      case TrueMember: True
      have c1_i2: "c1 ∈ gamma_ivl i2"
        by (rule subsetD[OF gamma_ivl_mono[OF assms(2)]
              TrueMember])
      have c1_fct2: "c1 ∈ gamma_congruence fct2"
        unfolding gamma_congruence_def rep2
        using relation(2) by simp
      have c1_target:
        "c1 ∈ gamma_ivl
          (refine_ivl_with_congruence fct2 i2)"
        by (rule subsetD[
              OF refine_ivl_with_congruence_cover])
           (use c1_i2 c1_fct2 in auto)
      have point_le:
        "Ivl (Fin c1) (Fin c1) <=
         refine_ivl_with_congruence fct2 i2"
        using c1_target singleton_le_iff by blast
      have source_refined:
        "refine_ivl_with_congruence fct1 i1 =
         intersect_ivl
           (Ivl (Fin c1) (Fin c1)) i1"
        unfolding refine_ivl_with_congruence_def rep1 i1
        using True by simp
      have source_le:
        "refine_ivl_with_congruence fct1 i1 <=
         Ivl (Fin c1) (Fin c1)"
        unfolding source_refined
        by (rule intersect_ivl_le1)
      show ?thesis
        by (rule order_trans[OF source_le point_le])
    qed
  qed
next
  case False
  have modulus1: "0 < m1"
    by (rule normalized_congruence_rep_nonzero(3)
          [OF normalized1' False])
  have m2_nonzero: "m2 ≠ 0"
    using relation(1) modulus1 by auto
  have modulus2: "0 < m2"
    by (rule normalized_congruence_rep_nonzero(3)
```

```
            [OF normalized2' m2_nonzero])
        have lower:
          "congruence_lower_bound c2 m2 l2 <=
           congruence_lower_bound c1 m1 l1"
          by (rule congruence_lower_bound_mono[
                OF modulus1 modulus2 relation(1)
                  relation(2) bounds(1)])
        have upper:
          "congruence_upper_bound c1 m1 u1 <=
           congruence_upper_bound c2 m2 u2"
          by (rule congruence_upper_bound_mono[
                OF modulus1 modulus2 relation(1)
                  relation(2) bounds(2)])
        have raw_hull:
          "Ivl
             (congruence_lower_bound c1 m1 l1)
             (congruence_upper_bound c1 m1 u1) <=
           Ivl
             (congruence_lower_bound c2 m2 l2)
             (congruence_upper_bound c2 m2 u2)"
          using lower upper
          by (simp add: less_eq_ivl_def)
        have hull:
          "mk_ivl
             (congruence_lower_bound c1 m1 l1)
             (congruence_upper_bound c1 m1 u1) <=
           mk_ivl
             (congruence_lower_bound c2 m2 l2)
             (congruence_upper_bound c2 m2 u2)"
          by (rule mk_ivl_mono[OF raw_hull])
        have refined1:
          "refine_ivl_with_congruence fct1 i1 =
           intersect_ivl
             (mk_ivl
               (congruence_lower_bound c1 m1 l1)
               (congruence_upper_bound c1 m1 u1))
             i1"
          unfolding refine_ivl_with_congruence_def rep1 i1
          using False by simp
        have refined2:
          "refine_ivl_with_congruence fct2 i2 =
           intersect_ivl
             (mk_ivl
               (congruence_lower_bound c2 m2 l2)
               (congruence_upper_bound c2 m2 u2))
             i2"
          unfolding refine_ivl_with_congruence_def rep2 i2
          using m2_nonzero by simp
        show ?thesis
          unfolding refined1 refined2
          by (rule intersect_ivl_mono[OF hull assms(2)])
      qed
    qed
  qed
qed

lemma parity_fact_of_congruence_sound:
  "gamma_congruence fct ⊆
   gamma_parity (parity_fact_of_congruence fct)"
proof
  fix n
  assume member: "n ∈ gamma_congruence fct"
  show
    "n ∈ gamma_parity (parity_fact_of_congruence fct)"
  proof (cases "Rep_congruence fct")
    case None
    with member show ?thesis
      unfolding gamma_congruence_def by simp
  next
    case (Some cm)
    obtain c m where cm: "cm = (c, m)"
      by (cases cm)
    have rep: "Rep_congruence fct = Some (c, m)"
      using Some cm by simp
    have congruent: "m dvd n - c"
      using member
```

```
      unfolding gamma_congruence_def rep by simp
    show ?thesis
    proof (cases "m = 0 ∨ even m")
      case True
      have same_parity: "even n = even c"
      proof (cases "m = 0")
        case True
        with congruent have "n = c" by simp
        then show ?thesis by simp
      next
        case False
        with True have even_m: "even m" by simp
        have even_difference: "even (n - c)"
          using even_m congruent
          by (meson dvd_trans)
        then show ?thesis
          by (simp add: even_diff_iff)
      qed
      show ?thesis
        unfolding parity_fact_of_congruence_def rep
        using True same_parity
        by auto
    next
      case False
      then show ?thesis
        unfolding parity_fact_of_congruence_def rep
        by simp
    qed
  qed
qed

lemma parity_fact_of_congruence_mono:
  assumes "fct1 <= fct2"
  shows
    "parity_fact_of_congruence fct1 <=
     parity_fact_of_congruence fct2"
proof -
  have rep_le:
    "congruence_le_rep
       (Rep_congruence fct1)
       (Rep_congruence fct2)"
    using assms
    unfolding less_eq_congruence_def congruence_le_def .
  show ?thesis
  proof (cases "Rep_congruence fct1")
    case None
    then show ?thesis
      unfolding parity_fact_of_congruence_def
        less_eq_parity_def
      by simp
  next
    case (Some cm1)
    obtain c1 m1 where cm1: "cm1 = (c1, m1)"
      by (cases cm1)
    have rep1: "Rep_congruence fct1 = Some (c1, m1)"
      using Some cm1 by simp
    show ?thesis
    proof (cases "Rep_congruence fct2")
      case None
      with rep_le rep1 show ?thesis by simp
    next
      case (Some cm2)
      obtain c2 m2 where cm2: "cm2 = (c2, m2)"
        by (cases cm2)
      have rep2: "Rep_congruence fct2 = Some (c2, m2)"
        using Some cm2 by simp
      have relation:
        "m2 dvd m1" "m2 dvd c1 - c2"
        using rep_le unfolding rep1 rep2
        by simp_all
      show ?thesis
      proof (cases "m2 = 0 ∨ even m2")
        case False
        have target_top:
          "parity_fact_of_congruence fct2 = (top :: parity)"
          unfolding parity_fact_of_congruence_def rep2
```

```
          top_parity_def
        using False by simp
      show ?thesis
        unfolding target_top by simp
    next
      case True
      have source_precise: "m1 = 0 ∨ even m1"
      proof (cases "m2 = 0")
        case True
        with relation(1) have "m1 = 0" by simp
        then show ?thesis by simp
      next
        case False
        with True have even_m2: "even m2" by simp
        have "even m1"
          using even_m2 relation(1)
          by (meson dvd_trans)
        then show ?thesis by simp
      qed
      have same_parity: "even c1 = even c2"
      proof (cases "m2 = 0")
        case True
        with relation(2) have "c1 = c2" by simp
        then show ?thesis by simp
      next
        case False
        with True have even_m2: "even m2" by simp
        have even_difference: "even (c1 - c2)"
          using even_m2 relation(2)
          by (meson dvd_trans)
        then show ?thesis
          by (simp add: even_diff_iff)
      qed
      have equal_facts:
        "parity_fact_of_congruence fct1 =
         parity_fact_of_congruence fct2"
        unfolding parity_fact_of_congruence_def rep1 rep2
        using True source_precise same_parity
        by auto
      then show ?thesis by simp
    qed
  qed
qed

lemma gamma_refine_parity_with_congruence [simp]:
  "gamma_parity (refine_parity_with_congruence fct p) =
   gamma_parity (parity_fact_of_congruence fct) ∩
   gamma_parity p"
  by (simp add: refine_parity_with_congruence_def)

lemma refine_parity_with_congruence_le:
  "refine_parity_with_congruence fct p <= p"
  unfolding refine_parity_with_congruence_def
  by (rule inf_le2)

lemma refine_parity_with_congruence_mono:
  assumes "fct1 <= fct2" "p1 <= p2"
  shows
    "refine_parity_with_congruence fct1 p1 <=
     refine_parity_with_congruence fct2 p2"
  unfolding refine_parity_with_congruence_def
  by (rule inf_mono)
     (use assms parity_fact_of_congruence_mono in auto)

lemma refine_congruence_with_congruence_le:
  assumes "fct <= c"
  shows "refine_congruence_with_congruence fct c <= c"
  using assms
  unfolding refine_congruence_with_congruence_def .

lemma refine_congruence_with_congruence_mono:
  assumes "fct1 <= fct2"
  shows
    "refine_congruence_with_congruence fct1 c1 <=
     refine_congruence_with_congruence fct2 c2"
```

```
  using assms
  unfolding refine_congruence_with_congruence_def .

definition refine_congruence :: "int_dom => int_dom" where
  "refine_congruence d =
     (let fct = congruence_fact_of_int_dom d
      in d⦇
           int_ivl :=
             refine_ivl_with_congruence fct (int_ivl d),
           int_parity :=
             refine_parity_with_congruence fct (int_parity d),
           int_congruence :=
             refine_congruence_with_congruence
               fct (int_congruence d)
         ⦈)"

lemma refine_congruence_exact:
  "gamma_int_dom (refine_congruence d) = gamma_int_dom d"
proof -
  let ?fct = "congruence_fact_of_int_dom d"
  have ivl_cover:
    "gamma_ivl (int_ivl d) ∩
     gamma_congruence ?fct ⊆
     gamma_ivl
       (refine_ivl_with_congruence ?fct (int_ivl d))"
    by (rule refine_ivl_with_congruence_cover)
  have ivl_subset:
    "gamma_ivl
       (refine_ivl_with_congruence ?fct (int_ivl d))
     ⊆ gamma_ivl (int_ivl d)"
    by (rule gamma_ivl_mono)
       (rule refine_ivl_with_congruence_le)
  have parity_cover:
    "gamma_congruence ?fct ⊆
     gamma_parity
       (refine_parity_with_congruence
         ?fct (int_parity d))"
  proof -
    have fact_sound:
      "gamma_congruence ?fct ⊆
       gamma_parity (parity_fact_of_congruence ?fct)"
      by (rule parity_fact_of_congruence_sound)
    show ?thesis
      using fact_sound
      by auto
  qed
  show ?thesis
    unfolding refine_congruence_def gamma_int_dom_def
      Let_def refine_congruence_with_congruence_def
    using ivl_cover ivl_subset parity_cover
    by auto
qed

lemma refine_congruence_reductive:
  "refine_congruence d <= d"
  unfolding refine_congruence_def Let_def
  by (simp add: less_eq_int_dom_ext_def
        refine_ivl_with_congruence_le
        refine_parity_with_congruence_le
        refine_congruence_with_congruence_le
        congruence_fact_of_int_dom_le_congruence)

lemma refine_congruence_mono:
  "mono refine_congruence"
proof (rule monoI)
  fix d e :: int_dom
  assume de: "d <= e"
  from de have components:
    "int_sign d <= int_sign e"
    "int_ivl d <= int_ivl e"
    "int_parity d <= int_parity e"
    "int_congruence d <= int_congruence e"
    by (simp_all add: less_eq_int_dom_ext_def)
  have fct_le:
    "congruence_fact_of_int_dom d <=
     congruence_fact_of_int_dom e"
```

```
    by (rule congruence_fact_of_int_dom_mono[OF de])
  show "refine_congruence d <= refine_congruence e"
    unfolding refine_congruence_def Let_def
    by (simp add: less_eq_int_dom_ext_def
          components(1)
          refine_ivl_with_congruence_mono[
            OF fct_le components(2)]
          refine_parity_with_congruence_mono[
            OF fct_le components(3)]
          refine_congruence_with_congruence_mono[
            OF fct_le])
qed

lemma refine_congruence_reduction_step:
  "int_reduction_step refine_congruence"
  unfolding int_reduction_step_def
  using refine_congruence_exact
    refine_congruence_reductive
    refine_congruence_mono
  by blast

definition refinement_steps :: "(int_dom => int_dom) list" where
  "refinement_steps = [refine_interval, refine_congruence]"

definition refine_round :: "int_dom => int_dom" where
  "refine_round = apply_reduction_steps refinement_steps"

lemma refine_round_reduction_step:
  "int_reduction_step refine_round"
  unfolding refine_round_def
  by (rule int_reduction_steps)
     (simp add: refinement_steps_def
        int_reduction_step_refine_interval
        refine_congruence_reduction_step)

end
```

16.3 Int_Refinement_Control

```
theory Int_Refinement_Control
  imports Int_Refinement
begin
```

How often to run the exchange, and what a run costs

The laws in `Voblint_Analysis_Int.Int_Refinement` say what one component may tell another; they say nothing about when to ask. This theory settles that. `refine_round` is one full pass over the exchange; a `refine_mode` picks how many passes an analysis run takes — `Refine_Never` none, `Refine_Once` a single pass, `Refine_Fixpoint` passes until the value stops changing.

The fixpoint mode is the awkward one. Iteration is a `while_option` loop, so it returns `None` when it does not stabilize; `refine_fix` keeps the input in that branch to stay a total function, while the generated code keeps running the structural-equality loop. Every mode is proved reductive and concretization-preserving, but only `Refine_Never` and `Refine_Once` are monotone — `refine_nonfixpoint_mono` — which is why callers that need monotonicity carry the `mode` $\neq$ `Refine_Fixpoint` side condition.

```
datatype refine_mode =
    Refine_Never
  | Refine_Once
  | Refine_Fixpoint

definition canonical_refine_step :: "int_dom => int_dom" where
  "canonical_refine_step d =
     (let d' = refine_round d
      in if is_empty d' then bot else d')"

definition refine_fix_option :: "int_dom => int_dom option" where
  "refine_fix_option d =
     while_option
       (λx. canonical_refine_step x ≠ x)
       canonical_refine_step
       d"
```

The option result is `None` exactly when structural iteration does not stabilize. The total logical wrapper returns the input in that branch; generated execution remains in the structural-equality loop.

```
definition refine_fix :: "int_dom => int_dom" where
  "refine_fix d =
     (case refine_fix_option d of
        Some r => r
      | None => d)"

definition refine :: "refine_mode => int_dom => int_dom" where
  "refine mode d =
     (case mode of
        Refine_Never => d
      | Refine_Once => refine_round d
      | Refine_Fixpoint => refine_fix d)"


lemma canonical_refine_step_exact:
  "gamma_int_dom (canonical_refine_step d) = gamma_int_dom d"
proof (cases "is_empty (refine_round d)")
  case True
  have round_empty:
    "gamma_int_dom (refine_round d) = {}"
    using True
    by (simp add: is_bottom_int_dom_correct)
  have input_empty: "gamma_int_dom d = {}"
    using round_empty
      int_reduction_step_exact[OF refine_round_reduction_step]
    by simp
  have bottom_empty:
    "gamma_int_dom (bot :: int_dom) = {}"
  proof -
    have "γ (bot :: int_dom) = {}"
      by (rule gamma_bot)
    then show ?thesis by simp
  qed
  show ?thesis
    unfolding canonical_refine_step_def Let_def
    using True input_empty bottom_empty by simp
```

```
next
  case False
  show ?thesis
    unfolding canonical_refine_step_def Let_def
    using False
      int_reduction_step_exact[OF refine_round_reduction_step]
    by simp
qed


lemma refine_fix_option_exact:
  assumes result: "refine_fix_option d = Some r"
  shows "gamma_int_dom r = gamma_int_dom d"
proof -
  have loop:
    "while_option
       (λx. canonical_refine_step x ≠ x)
       canonical_refine_step d = Some r"
    using result unfolding refine_fix_option_def .
  show ?thesis
  proof (rule while_option_rule[
      where P="λx.
        gamma_int_dom x = gamma_int_dom d"
        and b="λx.
          canonical_refine_step x ≠ x"
        and c=canonical_refine_step
        and s=d
        and t=r])
    fix x
    assume "gamma_int_dom x = gamma_int_dom d"
      "canonical_refine_step x ≠ x"
    then show
      "gamma_int_dom (canonical_refine_step x) =
       gamma_int_dom d"
      using canonical_refine_step_exact by simp
  next
    show
      "while_option
        (λx. canonical_refine_step x ≠ x)
        canonical_refine_step d = Some r"
      by (rule loop)
  next
    show "gamma_int_dom d = gamma_int_dom d"
      by simp
  qed
qed

lemma refine_fix_option_stable:
  assumes result: "refine_fix_option d = Some r"
  shows "canonical_refine_step r = r"
proof -
  have loop:
    "while_option
       (λx. canonical_refine_step x ≠ x)
       canonical_refine_step d = Some r"
    using result unfolding refine_fix_option_def .
  have "¬ canonical_refine_step r ≠ r"
    by (rule while_option_stop[OF loop])
  then show ?thesis by simp
qed

lemma refine_round_bot [simp]:
  "refine_round (bot :: int_dom) = bot"
proof (rule antisym)
  show "refine_round (bot :: int_dom) <= bot"
    by (rule int_reduction_step_reductive[
          OF refine_round_reduction_step])
  show "bot <= refine_round (bot :: int_dom)"
    by simp
qed

lemma refine_fix_option_round_stable:
  assumes result: "refine_fix_option d = Some r"
  shows "refine_round r = r"
proof -
  have stable: "canonical_refine_step r = r"
```

```
    by (rule refine_fix_option_stable[OF result])
  show ?thesis
  proof (cases "is_empty (refine_round r)")
    case True
    have "r = bot"
      using stable True
      unfolding canonical_refine_step_def Let_def
      by simp
    then show ?thesis by simp
  next
    case False
    with stable show ?thesis
      unfolding canonical_refine_step_def Let_def
      by simp
  qed
qed


lemma refine_fix_exact [simp]:
  "gamma_int_dom (refine_fix d) = gamma_int_dom d"
proof (cases "refine_fix_option d")
  case None
  then show ?thesis
    unfolding refine_fix_def by simp
next
  case (Some r)
  have "gamma_int_dom r = gamma_int_dom d"
    by (rule refine_fix_option_exact[OF Some])
  with Some show ?thesis
    unfolding refine_fix_def by simp
qed

lemma refine_never [simp]:
  "refine Refine_Never d = d"
  by (simp add: refine_def)

lemma refine_once [simp]:
  "refine Refine_Once d = refine_round d"
  by (simp add: refine_def)

lemma refine_fixpoint [simp]:
  "refine Refine_Fixpoint d = refine_fix d"
  by (simp add: refine_def)

lemma refine_once_reduction_step:
  "int_reduction_step (refine Refine_Once)"
proof -
  have "refine Refine_Once = refine_round"
    by (rule ext) simp
  then show ?thesis
    by (simp add: refine_round_reduction_step)
qed

lemma refine_once_reductive:
  "refine Refine_Once d <= d"
  by (rule int_reduction_step_reductive[
        OF refine_once_reduction_step])

lemma refine_once_mono:
  "mono (refine Refine_Once)"
  by (rule int_reduction_step_mono[
        OF refine_once_reduction_step])

lemma refine_exact:
  "gamma_int_dom (refine mode d) = gamma_int_dom d"
proof (cases mode)
  case Refine_Never
  then show ?thesis by simp
next
  case Refine_Once
  then show ?thesis
    using int_reduction_step_exact[OF refine_round_reduction_step]
    by simp
next
  case Refine_Fixpoint
  then show ?thesis by simp
```

```
qed

lemma canonical_refine_step_reductive:
  "canonical_refine_step d <= d"
proof (cases "is_empty (refine_round d)")
  case True
  then show ?thesis
    unfolding canonical_refine_step_def Let_def
    by simp
next
  case False
  then show ?thesis
    unfolding canonical_refine_step_def Let_def
    using int_reduction_step_reductive[OF refine_round_reduction_step]
    by simp
qed

lemma refine_fix_option_reductive:
  assumes result: "refine_fix_option d = Some r"
  shows "r <= d"
proof -
  have loop:
    "while_option
       (λx. canonical_refine_step x ≠ x)
       canonical_refine_step d = Some r"
    using result unfolding refine_fix_option_def .
  show ?thesis
  proof (rule while_option_rule[
      where P="λx. x <= d"
        and b="λx. canonical_refine_step x ≠ x"
        and c=canonical_refine_step
        and s=d
        and t=r])
    fix x
    assume "x <= d" "canonical_refine_step x ≠ x"
    then show "canonical_refine_step x <= d"
      using canonical_refine_step_reductive[of x] by simp
  next
    show
      "while_option
        (λx. canonical_refine_step x ≠ x)
        canonical_refine_step d = Some r"
      by (rule loop)
  next
    show "d <= d" by simp
  qed
qed

lemma refine_fix_reductive [simp]:
  "refine_fix d <= d"
proof (cases "refine_fix_option d")
  case None
  then show ?thesis
    unfolding refine_fix_def by simp
next
  case (Some r)
  have "r <= d"
    by (rule refine_fix_option_reductive[OF Some])
  with Some show ?thesis
    unfolding refine_fix_def by simp
qed

lemma refine_reductive:
  "refine mode d <= d"
proof (cases mode)
  case Refine_Never
  then show ?thesis by simp
next
  case Refine_Once
  then show ?thesis
    using refine_once_reductive by simp
next
  case Refine_Fixpoint
  then show ?thesis by simp
qed
```

Witnesses

Concrete values for the claims above. `progressive_refinement_input` is the raw sum of an odd value in `[-1,-1]` and one in `[0,1]`: its components together denote only `0`, but one round still leaves Sign at `SNonPos`, so a further round makes progress and `Refine_Once` differs from `Refine_Fixpoint`.

```
definition progressive_refinement_input :: int_dom where
  "progressive_refinement_input =
     int_dom_sipc STop (Ivl (Fin (-1)) (Fin 0)) PEven (top :: congruence)"

lemma refinement_round_is_progressive:
  "refine_round progressive_refinement_input =
     int_dom_sipc SNonPos (Ivl (Fin 0) (Fin 0)) PEven (mk_congruence 0 2)"
  by eval

lemma refinement_once_not_fixpoint:
  "refine Refine_Once progressive_refinement_input
     ≠ refine Refine_Fixpoint progressive_refinement_input"
  by eval
```

Why `Voblint_Analysis_Int.Int_Warrowing` runs no refinement after `narrow`. `a` is a widened state (top); `b` a newer, more precise result whose components together denote only `0` while Sign has not caught up. Componentwise narrowing satisfies `b ≤ narrow a b ≤ a`; refining that result lets Sign derive `SNonPos` from the exact Interval bound, strictly below `b`'s `STop`, which breaks the solver's `narrow_ge` bracket.

```
lemma post_narrow_refinement_would_violate_narrow_ge:
  "let a = (top :: int_dom);
       b = int_dom_sipc STop (Ivl (Fin (-1)) (Fin 0)) PEven (top :: congruence)
   in b ≤ a ∧ ¬ (b ≤ refine Refine_Once (a ∆ b))"
  by eval

end
```

16.4 Int_Arithmetic

```
theory Int_Arithmetic
  imports
    Int_Refinement_Control
    "Voblint_Analysis_Sign.Sign_Arithmetic"
    "Voblint_Analysis_Interval.Interval_Backward"
    "Voblint_Analysis_Parity.Parity_Domain"
    "Voblint_Analysis_Congruence.Congruence_Arithmetic"
begin
```

Composite integer arithmetic

Raw operations preserve the independent component results. Refinement is an explicit policy at the composite operation boundary, so the same arithmetic evaluator can expose the unreduced product or apply one or repeated progressive reduction rounds.

```
definition plus_int_dom_raw :: "int_dom => int_dom => int_dom" where
  "plus_int_dom_raw a b =
     (top :: int_dom)⦇
       int_sign := int_sign a + int_sign b,
       int_ivl := int_ivl a + int_ivl b,
       int_parity := int_parity a + int_parity b,
       int_congruence := int_congruence a + int_congruence b
     ⦈"

definition minus_int_dom_raw :: "int_dom => int_dom => int_dom" where
  "minus_int_dom_raw a b =
     (top :: int_dom)⦇
       int_sign := int_sign a - int_sign b,
       int_ivl := int_ivl a - int_ivl b,
       int_parity := int_parity a - int_parity b,
       int_congruence := int_congruence a - int_congruence b
     ⦈"

definition times_int_dom_raw :: "int_dom => int_dom => int_dom" where
  "times_int_dom_raw a b =
     (top :: int_dom)⦇
       int_sign := int_sign a * int_sign b,
       int_ivl := int_ivl a * int_ivl b,
       int_parity := int_parity a * int_parity b,
       int_congruence := int_congruence a * int_congruence b
     ⦈"

definition div_int_dom_raw :: "int_dom => int_dom => int_dom" where
  "div_int_dom_raw a b =
     (top :: int_dom)⦇
       int_sign := sign_div (int_sign a) (int_sign b),
       int_ivl := ivl_div (int_ivl a) (int_ivl b),
       int_parity := parity_div (int_parity a) (int_parity b),
       int_congruence := congruence_div (int_congruence a) (int_congruence b)
     ⦈"

definition mod_int_dom_raw :: "int_dom => int_dom => int_dom" where
  "mod_int_dom_raw a b =
     (top :: int_dom)⦇
       int_sign := sign_mod (int_sign a) (int_sign b),
       int_ivl := ivl_mod (int_ivl a) (int_ivl b),
       int_parity := parity_mod (int_parity a) (int_parity b),
       int_congruence := congruence_mod (int_congruence a) (int_congruence b)
     ⦈"

definition plus_int_dom ::
    "refine_mode => int_dom => int_dom => int_dom"
where
  "plus_int_dom mode a b =
     refine mode (plus_int_dom_raw a b)"

definition minus_int_dom ::
    "refine_mode => int_dom => int_dom => int_dom"
where
  "minus_int_dom mode a b =
     refine mode (minus_int_dom_raw a b)"

definition times_int_dom ::
    "refine_mode => int_dom => int_dom => int_dom"
```

```
where
  "times_int_dom mode a b =
     refine mode (times_int_dom_raw a b)"


definition div_int_dom ::
    "refine_mode => int_dom => int_dom => int_dom"
where
  "div_int_dom mode a b =
     refine mode (div_int_dom_raw a b)"


definition mod_int_dom ::
    "refine_mode => int_dom => int_dom => int_dom"
where
  "mod_int_dom mode a b =
     refine mode (mod_int_dom_raw a b)"
```

Raw operation laws

Each raw operation applies the matching sign, interval, parity and congruence operation to its own field, so its soundness and monotonicity follow from the four component laws with no interaction between fields.

```
lemma plus_int_dom_raw_sound:
  assumes "x : gamma_int_dom a"
      and "y : gamma_int_dom b"
  shows "x + y : gamma_int_dom (plus_int_dom_raw a b)"
  using assms
  by (auto simp: gamma_int_dom_def plus_int_dom_raw_def
        intro: sign_plus_sound ivl_plus_sound
          parity_plus_sound congruence_plus_sound)

lemma minus_int_dom_raw_sound:
  assumes "x : gamma_int_dom a"
      and "y : gamma_int_dom b"
  shows "x - y : gamma_int_dom (minus_int_dom_raw a b)"
  using assms
  by (auto simp: gamma_int_dom_def minus_int_dom_raw_def
        intro: sign_minus_sound ivl_minus_sound
          parity_minus_sound congruence_minus_sound)

lemma times_int_dom_raw_sound:
  assumes "x : gamma_int_dom a"
      and "y : gamma_int_dom b"
  shows "x * y : gamma_int_dom (times_int_dom_raw a b)"
  using assms
  by (auto simp: gamma_int_dom_def times_int_dom_raw_def
        intro: sign_times_sound ivl_times_sound
          parity_times_sound congruence_times_sound)

lemma div_int_dom_raw_sound:
  assumes "x : gamma_int_dom a"
      and "y : gamma_int_dom b"
  shows "c_div x y : gamma_int_dom (div_int_dom_raw a b)"
  using assms
  by (auto simp: gamma_int_dom_def div_int_dom_raw_def
        intro: sign_div_sound ivl_div_sound
          parity_div_sound congruence_div_sound)

lemma mod_int_dom_raw_sound:
  assumes "x : gamma_int_dom a"
      and "y : gamma_int_dom b"
  shows "c_mod x y : gamma_int_dom (mod_int_dom_raw a b)"
  using assms
  by (auto simp: gamma_int_dom_def mod_int_dom_raw_def
        intro: sign_mod_sound ivl_mod_sound
          parity_mod_sound congruence_mod_sound)

lemma plus_int_dom_raw_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows "plus_int_dom_raw a1 b1 <= plus_int_dom_raw a2 b2"
  using assms
  by (auto simp: plus_int_dom_raw_def less_eq_int_dom_ext_def
        intro: sign_plus_combine_mono ivl_plus_mono
          parity_plus_combine_mono congruence_plus_mono)
```

```
lemma minus_int_dom_raw_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows "minus_int_dom_raw a1 b1 <= minus_int_dom_raw a2 b2"
  using assms
  by (auto simp: minus_int_dom_raw_def less_eq_int_dom_ext_def
        intro: sign_minus_combine_mono ivl_minus_mono
          parity_minus_combine_mono congruence_minus_mono)

lemma times_int_dom_raw_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows "times_int_dom_raw a1 b1 <= times_int_dom_raw a2 b2"
  using assms
  by (auto simp: times_int_dom_raw_def less_eq_int_dom_ext_def
        intro: sign_times_combine_mono ivl_times_mono
          parity_times_combine_mono congruence_times_mono)


lemma div_int_dom_raw_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows "div_int_dom_raw a1 b1 <= div_int_dom_raw a2 b2"
  using assms
  by (auto simp: div_int_dom_raw_def less_eq_int_dom_ext_def
        intro: sign_div_mono ivl_div_mono
          parity_div_mono congruence_div_mono)


lemma mod_int_dom_raw_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows "mod_int_dom_raw a1 b1 <= mod_int_dom_raw a2 b2"
  using assms
  by (auto simp: mod_int_dom_raw_def less_eq_int_dom_ext_def
        intro: sign_mod_mono ivl_mod_mono
          parity_mod_mono congruence_mod_mono)
```

Mode-aware operation laws

The mode-aware operations apply `refine mode` to the raw result. Soundness holds in every mode because refinement preserves the concretization (`refine_exact`); monotonicity needs a mode other than `Refine_Fixpoint` (`refine_nonfixpoint_mono`).

```
lemma plus_int_dom_sound:
  assumes "x : gamma_int_dom a"
      and "y : gamma_int_dom b"
  shows "x + y : gamma_int_dom (plus_int_dom mode a b)"
proof -
  have "x + y : gamma_int_dom (plus_int_dom_raw a b)"
    by (rule plus_int_dom_raw_sound[OF assms])
  then show ?thesis
    unfolding plus_int_dom_def
    using refine_exact by simp
qed

lemma minus_int_dom_sound:
  assumes "x : gamma_int_dom a"
      and "y : gamma_int_dom b"
  shows "x - y : gamma_int_dom (minus_int_dom mode a b)"
proof -
  have "x - y : gamma_int_dom (minus_int_dom_raw a b)"
    by (rule minus_int_dom_raw_sound[OF assms])
  then show ?thesis
    unfolding minus_int_dom_def
    using refine_exact by simp
qed

lemma times_int_dom_sound:
  assumes "x : gamma_int_dom a"
      and "y : gamma_int_dom b"
  shows "x * y : gamma_int_dom (times_int_dom mode a b)"
proof -
  have "x * y : gamma_int_dom (times_int_dom_raw a b)"
    by (rule times_int_dom_raw_sound[OF assms])
  then show ?thesis
    unfolding times_int_dom_def
    using refine_exact by simp
qed
```

```
lemma div_int_dom_sound:
  assumes "x : gamma_int_dom a"
      and "y : gamma_int_dom b"
  shows "c_div x y : gamma_int_dom (div_int_dom mode a b)"
proof -
  have "c_div x y : gamma_int_dom (div_int_dom_raw a b)"
    by (rule div_int_dom_raw_sound[OF assms])
  then show ?thesis
    unfolding div_int_dom_def
    using refine_exact by simp
qed

lemma mod_int_dom_sound:
  assumes "x : gamma_int_dom a"
      and "y : gamma_int_dom b"
  shows "c_mod x y : gamma_int_dom (mod_int_dom mode a b)"
proof -
  have "c_mod x y : gamma_int_dom (mod_int_dom_raw a b)"
    by (rule mod_int_dom_raw_sound[OF assms])
  then show ?thesis
    unfolding mod_int_dom_def
    using refine_exact by simp
qed
```

The total logical Fixpoint wrapper returns its input when the structural loop has no result. Without a termination theorem this fallback does not support unconditional order monotonicity. Never and Once inherit monotonicity directly from the identity and the verified reduction round.

```
lemma refine_nonfixpoint_mono:
  assumes "mode ~= Refine_Fixpoint"
  shows "mono (refine mode)"
proof (cases mode)
  case Refine_Never
  then show ?thesis by (simp add: mono_def)
next
  case Refine_Once
  then show ?thesis using refine_once_mono by simp
next
  case Refine_Fixpoint
  with assms show ?thesis by simp
qed

lemma plus_int_dom_mono:
  assumes "mode ~= Refine_Fixpoint"
      and "a1 <= a2" and "b1 <= b2"
  shows "plus_int_dom mode a1 b1 <= plus_int_dom mode a2 b2"
proof -
  have raw:
    "plus_int_dom_raw a1 b1 <= plus_int_dom_raw a2 b2"
    by (rule plus_int_dom_raw_mono[OF assms(2,3)])
  show ?thesis
    unfolding plus_int_dom_def
    by (rule monoD[OF refine_nonfixpoint_mono[OF assms(1)] raw])
qed

lemma minus_int_dom_mono:
  assumes "mode ~= Refine_Fixpoint"
      and "a1 <= a2" and "b1 <= b2"
  shows "minus_int_dom mode a1 b1 <= minus_int_dom mode a2 b2"
proof -
  have raw:
    "minus_int_dom_raw a1 b1 <= minus_int_dom_raw a2 b2"
    by (rule minus_int_dom_raw_mono[OF assms(2,3)])
  show ?thesis
    unfolding minus_int_dom_def
    by (rule monoD[OF refine_nonfixpoint_mono[OF assms(1)] raw])
qed

lemma times_int_dom_mono:
  assumes "mode ~= Refine_Fixpoint"
      and "a1 <= a2" and "b1 <= b2"
  shows "times_int_dom mode a1 b1 <= times_int_dom mode a2 b2"
proof -
  have raw:
    "times_int_dom_raw a1 b1 <= times_int_dom_raw a2 b2"
    by (rule times_int_dom_raw_mono[OF assms(2,3)])
  show ?thesis
```

```
    unfolding times_int_dom_def
    by (rule monoD[OF refine_nonfixpoint_mono[OF assms(1)] raw])
qed


lemma div_int_dom_mono:
  assumes "mode ~= Refine_Fixpoint"
      and "a1 <= a2" and "b1 <= b2"
  shows "div_int_dom mode a1 b1 <= div_int_dom mode a2 b2"
proof -
  have raw:
    "div_int_dom_raw a1 b1 <= div_int_dom_raw a2 b2"
    by (rule div_int_dom_raw_mono[OF assms(2,3)])
  show ?thesis
    unfolding div_int_dom_def
    by (rule monoD[OF refine_nonfixpoint_mono[OF assms(1)] raw])
qed


lemma mod_int_dom_mono:
  assumes "mode ~= Refine_Fixpoint"
      and "a1 <= a2" and "b1 <= b2"
  shows "mod_int_dom mode a1 b1 <= mod_int_dom mode a2 b2"
proof -
  have raw:
    "mod_int_dom_raw a1 b1 <= mod_int_dom_raw a2 b2"
    by (rule mod_int_dom_raw_mono[OF assms(2,3)])
  show ?thesis
    unfolding mod_int_dom_def
    by (rule monoD[OF refine_nonfixpoint_mono[OF assms(1)] raw])
qed
```

Comparison and truthiness queries

`int_dom` has no fixed `plus`/`minus`/`times` class instance – its arithmetic is parameterized by `refine_mode` (`plus_int_dom Refine_Never` and `plus_int_dom Refine_Fixpoint` are genuinely different operations) – so `int_dom` cannot literally interpret `Voblint_Nonrelational.Abstract_Arithmetic`'s `sound_arith_ops` locale, which fixes those as type-class operations. `int_dom_lt`/`int_dom_eqb`/ `int_dom_tobool` below still follow that locale's query shape exactly (`bool option`, `Some True`/`Some False`/`None`), and `aval_int_dom`'s `Less`/`Eq`/`Not`/`And`/`Or` cases below reuse that same shape by hand; only the generic locale interpretation itself does not transfer.

Each query consults the composite's own four components in turn – Interval and Sign first, since they carry ordering information Parity and Congruence lack – taking the first that decides. This treats `int_dom` as one reduced value with one query per comparison, not as four independent per-component evaluations mechanically joined.

`first_deciding`/`first_deciding2` are the shared "ask components in priority order, stop at the first that decides" policy, factored out once so `int_dom_lt`/`int_dom_eqb`/`int_dom_tobool` below are plain registries rather than repeating the same nested-`case` pattern three times. The query LIST holds functions, not pre-evaluated `bool option` results: generated ML is strict, so passing already-evaluated results would force all four component queries every time, discarding the short-circuit that a hand-written nested `case` gets for free.

```
fun first_deciding :: "('a => 'b option) list => 'a => 'b option" where
  "first_deciding [] x = None"
| "first_deciding (q # qs) x = (case q x of Some y => Some y | None => first_deciding qs x)"

fun first_deciding2 :: "('a => 'a => 'b option) list => 'a => 'a => 'b option" where
  "first_deciding2 [] x y = None"
| "first_deciding2 (q # qs) x y = (case q x y of Some z => Some z | None => first_deciding2 qs x y)"

lemma first_deciding_SomeE [elim]:
  assumes "first_deciding qs x = Some y"
  obtains q where "q ∈ set qs" and "q x = Some y"
  using assms by (induction qs) (auto split: option.splits)

lemma first_deciding_not_none_of_component:
  assumes "q ∈ set qs" and "q x = Some y"
  shows "first_deciding qs x ≠ None"
  using assms by (induction qs) (auto split: option.splits)

lemma first_deciding2_SomeE [elim]:
  assumes "first_deciding2 qs x y = Some z"
  obtains q where "q ∈ set qs" and "q x y = Some z"
  using assms by (induction qs) (auto split: option.splits)
```

```
lemma first_deciding2_not_none_of_component:
  assumes "q ∈ set qs" and "q x y = Some z"
  shows "first_deciding2 qs x y ≠ None"
  using assms by (induction qs) (auto split: option.splits)
```

A component decision found at the wider operands survives at narrower ones, so the whole query still decides there.

```
lemma first_deciding_mono:
  assumes "first_deciding qs x2 = Some c"
    and "⋀q. q ∈ set qs ⟹ q x2 = Some c ⟹ q x1 = Some c"
  shows "first_deciding qs x1 ≠ None"
proof -
  obtain q where q: "q ∈ set qs" "q x2 = Some c"
    using assms(1) by (rule first_deciding_SomeE)
  show ?thesis using first_deciding_not_none_of_component[OF q(1) assms(2)[OF q]] .
qed

lemma first_deciding2_mono:
  assumes "first_deciding2 qs x2 y2 = Some c"
    and "⋀q. q ∈ set qs ⟹ q x2 y2 = Some c ⟹ q x1 y1 = Some c"
  shows "first_deciding2 qs x1 y1 ≠ None"
proof -
  obtain q where q: "q ∈ set qs" "q x2 y2 = Some c"
    using assms(1) by (rule first_deciding2_SomeE)
  show ?thesis using first_deciding2_not_none_of_component[OF q(1) assms(2)[OF q]] .
qed

definition int_dom_lt :: "int_dom => int_dom => bool option" where
  "int_dom_lt d1 d2 =
     first_deciding2
       [λa b. interval_less (int_ivl a) (int_ivl b),
        λa b. sign_lt (int_sign a) (int_sign b),
        λa b. parity_lt (int_parity a) (int_parity b),
        λa b. congruence_lt (int_congruence a) (int_congruence b)]
       d1 d2"

definition int_dom_eqb :: "int_dom => int_dom => bool option" where
  "int_dom_eqb d1 d2 =
     first_deciding2
       [λa b. interval_eq (int_ivl a) (int_ivl b),
        λa b. sign_eqb (int_sign a) (int_sign b),
        λa b. parity_eqb (int_parity a) (int_parity b),
        λa b. congruence_eqb (int_congruence a) (int_congruence b)]
       d1 d2"

definition int_dom_tobool :: "int_dom => bool option" where
  "int_dom_tobool d =
     first_deciding
       [λa. interval_tobool (int_ivl a),
        λa. sign_tobool (int_sign a),
        λa. parity_tobool (int_parity a),
        λa. congruence_tobool (int_congruence a)]
       d"

lemma int_dom_not_bot_componentsE [elim]:
  assumes "¬ is_empty (d::int_dom)"
  obtains n where "n ∈ gamma_sign (int_sign d)" "n ∈ gamma_ivl (int_ivl d)"
    "n ∈ gamma_parity (int_parity d)" "n ∈ gamma_congruence (int_congruence d)"
proof -
  have "gamma_int_dom d ≠ {}"
    using assms by (simp add: is_bottom_int_dom_correct)
  then obtain n where "n ∈ gamma_int_dom d" by blast
  then have "n ∈ gamma_sign (int_sign d) ∧ n ∈ gamma_ivl (int_ivl d) ∧
             n ∈ gamma_parity (int_parity d) ∧ n ∈ gamma_congruence (int_congruence d)"
    by (simp add: gamma_int_dom_def)
  then show ?thesis using that by blast
qed

lemma int_dom_lt_sound:
  assumes "int_dom_lt d1 d2 = Some c" and "x ∈ gamma_int_dom d1" and "y ∈ gamma_int_dom d2"
  shows "(x < y) = c"
proof -
  have hx: "x ∈ gamma_sign (int_sign d1)" "x ∈ gamma_ivl (int_ivl d1)"
           "x ∈ gamma_parity (int_parity d1)" "x ∈ gamma_congruence (int_congruence d1)"
    using assms(2) by (simp_all add: gamma_int_dom_def)
```

```
  have hy: "y ∈ gamma_sign (int_sign d2)" "y ∈ gamma_ivl (int_ivl d2)"
           "y ∈ gamma_parity (int_parity d2)" "y ∈ gamma_congruence (int_congruence d2)"
    using assms(3) by (simp_all add: gamma_int_dom_def)
  from assms(1) show ?thesis
    unfolding int_dom_lt_def
    by (rule first_deciding2_SomeE)
       (auto simp only: list.set insert_iff empty_iff
          dest: interval_less_ivl_sound[OF _ hx(2) hy(2)] sign_lt_sound[OF _ hx(1) hy(1)]
          parity_lt_sound[OF _ hx(3) hy(3)] congruence_lt_sound[OF _ hx(4) hy(4)])
qed

lemma int_dom_eqb_sound:
  assumes "int_dom_eqb d1 d2 = Some c" and "x ∈ gamma_int_dom d1" and "y ∈ gamma_int_dom d2"
  shows "(x = y) = c"
proof -
  have hx: "x ∈ gamma_sign (int_sign d1)" "x ∈ gamma_ivl (int_ivl d1)"
           "x ∈ gamma_parity (int_parity d1)" "x ∈ gamma_congruence (int_congruence d1)"
    using assms(2) by (simp_all add: gamma_int_dom_def)
  have hy: "y ∈ gamma_sign (int_sign d2)" "y ∈ gamma_ivl (int_ivl d2)"
           "y ∈ gamma_parity (int_parity d2)" "y ∈ gamma_congruence (int_congruence d2)"
    using assms(3) by (simp_all add: gamma_int_dom_def)
  from assms(1) show ?thesis
    unfolding int_dom_eqb_def
    by (rule first_deciding2_SomeE)
       (auto simp only: list.set insert_iff empty_iff
          dest: interval_eq_ivl_sound[OF _ hx(2) hy(2)] sign_eqb_sound[OF _ hx(1) hy(1)]
          parity_eqb_sound[OF _ hx(3) hy(3)] congruence_eqb_sound[OF _ hx(4) hy(4)])
qed

lemma int_dom_tobool_sound:
  assumes "int_dom_tobool d = Some c" and "x ∈ gamma_int_dom d"
  shows "(x ≠ 0) = c"
proof -
  have hx: "x ∈ gamma_sign (int_sign d)" "x ∈ gamma_ivl (int_ivl d)"
           "x ∈ gamma_parity (int_parity d)" "x ∈ gamma_congruence (int_congruence d)"
    using assms(2) by (simp_all add: gamma_int_dom_def)
  from assms(1) show ?thesis
    unfolding int_dom_tobool_def
    by (rule first_deciding_SomeE)
       (auto simp only: list.set insert_iff empty_iff
          dest: interval_tobool_sound[OF _ hx(2), unfolded truthy_def]
            sign_tobool_sound[OF _ hx(1)]
          parity_tobool_sound[OF _ hx(3)] congruence_tobool_sound[OF _ hx(4)])
qed

lemma int_dom_lt_mono:
  assumes h1: "¬ is_empty (d1::int_dom)" and h2: "¬ is_empty e1"
      and hd: "d1 ≤ d2" and he: "e1 ≤ e2"
      and hwide: "int_dom_lt d2 e2 = Some c"
  shows "int_dom_lt d1 e1 = Some c"
proof -
  obtain n where hn: "n ∈ gamma_sign (int_sign d1)" "n ∈ gamma_ivl (int_ivl d1)"
      "n ∈ gamma_parity (int_parity d1)" "n ∈ gamma_congruence (int_congruence d1)"
    using int_dom_not_bot_componentsE[OF h1] .
  obtain m where hm: "m ∈ gamma_sign (int_sign e1)" "m ∈ gamma_ivl (int_ivl e1)"
      "m ∈ gamma_parity (int_parity e1)" "m ∈ gamma_congruence (int_congruence e1)"
    using int_dom_not_bot_componentsE[OF h2] .
  have le1: "int_sign d1 ≤ int_sign d2" "int_ivl d1 ≤ int_ivl d2"
            "int_parity d1 ≤ int_parity d2" "int_congruence d1 ≤ int_congruence d2"
    using hd by (simp_all add: less_eq_int_dom_ext_def)
  have le2: "int_sign e1 ≤ int_sign e2" "int_ivl e1 ≤ int_ivl e2"
            "int_parity e1 ≤ int_parity e2" "int_congruence e1 ≤ int_congruence e2"
    using he by (simp_all add: less_eq_int_dom_ext_def)
  have nb1: "¬ is_empty (int_sign d1)" "¬ is_empty (int_ivl d1)"
            "¬ is_empty (int_parity d1)" "¬ is_empty (int_congruence d1)"
    using hn by (auto simp: is_bottom_sign_correct is_bottom_ivl_correct
                  is_bottom_parity_correct is_bottom_congruence_correct)
  have nb2: "¬ is_empty (int_sign e1)" "¬ is_empty (int_ivl e1)"
            "¬ is_empty (int_parity e1)" "¬ is_empty (int_congruence e1)"
    using hm by (auto simp: is_bottom_sign_correct is_bottom_ivl_correct
                  is_bottom_parity_correct is_bottom_congruence_correct)
  have "int_dom_lt d1 e1 ≠ None"
    using hwide unfolding int_dom_lt_def
    by (rule first_deciding2_mono)
       (auto simp only: list.set insert_iff empty_iff
          intro: interval_less_mono[OF nb1(2) nb2(2) le1(2) le2(2)]
```

```
        sign_lt_mono[OF nb1(1) nb2(1) le1(1) le2(1)]
        parity_lt_mono[OF nb1(3) nb2(3) le1(3) le2(3)]
        congruence_lt_mono[OF nb1(4) nb2(4) le1(4) le2(4)])
  then obtain x where hx: "int_dom_lt d1 e1 = Some x" by auto
  have n_d1: "n ∈ gamma_int_dom d1" and m_e1: "m ∈ gamma_int_dom e1"
    using hn hm by (simp_all add: gamma_int_dom_def)
  have "(n < m) = c"
    using int_dom_lt_sound[OF hwide] n_d1 m_e1 gamma_mono[OF hd] gamma_mono[OF he] by auto
  moreover have "(n < m) = x" using int_dom_lt_sound[OF hx n_d1 m_e1] .
  ultimately show ?thesis using hx by simp
qed

lemma int_dom_eqb_mono:
  assumes h1: "¬ is_empty (d1::int_dom)" and h2: "¬ is_empty e1"
      and hd: "d1 ≤ d2" and he: "e1 ≤ e2"
      and hwide: "int_dom_eqb d2 e2 = Some c"
  shows "int_dom_eqb d1 e1 = Some c"
proof -
  obtain n where hn: "n ∈ gamma_sign (int_sign d1)" "n ∈ gamma_ivl (int_ivl d1)"
      "n ∈ gamma_parity (int_parity d1)" "n ∈ gamma_congruence (int_congruence d1)"
    using int_dom_not_bot_componentsE[OF h1] .
  obtain m where hm: "m ∈ gamma_sign (int_sign e1)" "m ∈ gamma_ivl (int_ivl e1)"
      "m ∈ gamma_parity (int_parity e1)" "m ∈ gamma_congruence (int_congruence e1)"
    using int_dom_not_bot_componentsE[OF h2] .
  have le1: "int_sign d1 ≤ int_sign d2" "int_ivl d1 ≤ int_ivl d2"
          "int_parity d1 ≤ int_parity d2" "int_congruence d1 ≤ int_congruence d2"
    using hd by (simp_all add: less_eq_int_dom_ext_def)
  have le2: "int_sign e1 ≤ int_sign e2" "int_ivl e1 ≤ int_ivl e2"
          "int_parity e1 ≤ int_parity e2" "int_congruence e1 ≤ int_congruence e2"
    using he by (simp_all add: less_eq_int_dom_ext_def)
  have nb1: "¬ is_empty (int_sign d1)" "¬ is_empty (int_ivl d1)"
          "¬ is_empty (int_parity d1)" "¬ is_empty (int_congruence d1)"
    using hn by (auto simp: is_bottom_sign_correct is_bottom_ivl_correct
                  is_bottom_parity_correct is_bottom_congruence_correct)
  have nb2: "¬ is_empty (int_sign e1)" "¬ is_empty (int_ivl e1)"
          "¬ is_empty (int_parity e1)" "¬ is_empty (int_congruence e1)"
    using hm by (auto simp: is_bottom_sign_correct is_bottom_ivl_correct
                  is_bottom_parity_correct is_bottom_congruence_correct)
  have "int_dom_eqb d1 e1 ≠ None"
    using hwide unfolding int_dom_eqb_def
    by (rule first_deciding2_mono)
       (auto simp only: list.set insert_iff empty_iff
          intro: interval_eq_mono[OF nb1(2) nb2(2) le1(2) le2(2)]
          sign_eqb_mono[OF nb1(1) nb2(1) le1(1) le2(1)]
          parity_eqb_mono[OF nb1(3) nb2(3) le1(3) le2(3)]
          congruence_eqb_mono[OF nb1(4) nb2(4) le1(4) le2(4)])
  then obtain x where hx: "int_dom_eqb d1 e1 = Some x" by auto
  have n_d1: "n ∈ gamma_int_dom d1" and m_e1: "m ∈ gamma_int_dom e1"
    using hn hm by (simp_all add: gamma_int_dom_def)
  have "(n = m) = c"
    using int_dom_eqb_sound[OF hwide] n_d1 m_e1 gamma_mono[OF hd] gamma_mono[OF he] by auto
  moreover have "(n = m) = x" using int_dom_eqb_sound[OF hx n_d1 m_e1] .
  ultimately show ?thesis using hx by simp
qed

lemma int_dom_tobool_mono:
  assumes h1: "¬ is_empty (d1::int_dom)" and hd: "d1 ≤ d2"
      and hwide: "int_dom_tobool d2 = Some c"
  shows "int_dom_tobool d1 = Some c"
proof -
  obtain n where hn: "n ∈ gamma_sign (int_sign d1)" "n ∈ gamma_ivl (int_ivl d1)"
      "n ∈ gamma_parity (int_parity d1)" "n ∈ gamma_congruence (int_congruence d1)"
    using int_dom_not_bot_componentsE[OF h1] .
  have le1: "int_sign d1 ≤ int_sign d2" "int_ivl d1 ≤ int_ivl d2"
          "int_parity d1 ≤ int_parity d2" "int_congruence d1 ≤ int_congruence d2"
    using hd by (simp_all add: less_eq_int_dom_ext_def)
  have nb1: "¬ is_empty (int_sign d1)" "¬ is_empty (int_ivl d1)"
          "¬ is_empty (int_parity d1)" "¬ is_empty (int_congruence d1)"
    using hn by (auto simp: is_bottom_sign_correct is_bottom_ivl_correct
                  is_bottom_parity_correct is_bottom_congruence_correct)
  have "int_dom_tobool d1 ≠ None"
    using hwide unfolding int_dom_tobool_def
    by (rule first_deciding_mono)
       (auto simp only: list.set insert_iff empty_iff
          intro: interval_tobool_mono[OF nb1(2) le1(2)] sign_tobool_mono[OF nb1(1) le1(1)]
          parity_tobool_mono[OF nb1(3) le1(3)] congruence_tobool_mono[OF nb1(4) le1(4)])
```

```
  then obtain x where hx: "int_dom_tobool d1 = Some x" by auto
  have n_d1: "n ∈ gamma_int_dom d1" using hn by (simp add: gamma_int_dom_def)
  have "(n ≠ 0) = c"
    using int_dom_tobool_sound[OF hwide] n_d1 gamma_mono[OF hd] by auto
  moreover have "(n ≠ 0) = x" using int_dom_tobool_sound[OF hx n_d1] .
  ultimately show ?thesis using hx by simp
qed
```

Arithmetic-expression evaluation

`Less`/`Eq`/`Not`/`And`/`Or` mirror the shape `Voblint_Nonrelational.Abstract_Arithmetic.sound_arith_ops` would assume were `int_dom` able to interpret it (see the comment above `int_dom_lt`): each operand is recursively evaluated to a whole `int_dom` value first, and only the composite-level `int_dom_lt`/ `int_dom_eqb`/`int_dom_tobool` query decides the result – never a separate per-component recursive walk.

```
fun aval_int_dom ::
    "refine_mode => exp => (vname => int_dom) => int_dom"
where
  "aval_int_dom mode (N n) sigma = int_dom_of_int n"
| "aval_int_dom mode (V x) sigma = sigma x"
| "aval_int_dom mode (Plus e1 e2) sigma =
     plus_int_dom mode
       (aval_int_dom mode e1 sigma)
       (aval_int_dom mode e2 sigma)"
| "aval_int_dom mode (Minus e1 e2) sigma =
     minus_int_dom mode
       (aval_int_dom mode e1 sigma)
       (aval_int_dom mode e2 sigma)"
| "aval_int_dom mode (Times e1 e2) sigma =
     times_int_dom mode
       (aval_int_dom mode e1 sigma)
       (aval_int_dom mode e2 sigma)"
| "aval_int_dom mode (Div e1 e2) sigma =
     div_int_dom mode
       (aval_int_dom mode e1 sigma)
       (aval_int_dom mode e2 sigma)"
| "aval_int_dom mode (Mod e1 e2) sigma =
     mod_int_dom mode
       (aval_int_dom mode e1 sigma)
       (aval_int_dom mode e2 sigma)"
| "aval_int_dom mode (Less e1 e2) sigma =
     (let a = aval_int_dom mode e1 sigma; b = aval_int_dom mode e2 sigma
      in if is_empty a ∨ is_empty b then bot
         else of_bool_option int_dom_of_int (int_dom_lt a b))"
| "aval_int_dom mode (LessEq e1 e2) sigma =
     (let a = aval_int_dom mode e1 sigma; b = aval_int_dom mode e2 sigma
      in if is_empty a ∨ is_empty b then bot
         else of_bool_option int_dom_of_int (map_option HOL.Not (int_dom_lt b a)))"
| "aval_int_dom mode (Greater e1 e2) sigma =
     (let a = aval_int_dom mode e1 sigma; b = aval_int_dom mode e2 sigma
      in if is_empty a ∨ is_empty b then bot
         else of_bool_option int_dom_of_int (int_dom_lt b a))"
| "aval_int_dom mode (GreaterEq e1 e2) sigma =
     (let a = aval_int_dom mode e1 sigma; b = aval_int_dom mode e2 sigma
      in if is_empty a ∨ is_empty b then bot
         else of_bool_option int_dom_of_int (map_option HOL.Not (int_dom_lt a b)))"
| "aval_int_dom mode (NotEq e1 e2) sigma =
     (let a = aval_int_dom mode e1 sigma; b = aval_int_dom mode e2 sigma
      in if is_empty a ∨ is_empty b then bot
         else of_bool_option int_dom_of_int (map_option HOL.Not (int_dom_eqb a b)))"
| "aval_int_dom mode (exp.Eq e1 e2) sigma =
     (let a = aval_int_dom mode e1 sigma; b = aval_int_dom mode e2 sigma
      in if is_empty a ∨ is_empty b then bot
         else of_bool_option int_dom_of_int (int_dom_eqb a b))"
| "aval_int_dom mode (exp.Not e) sigma =
     (let a = aval_int_dom mode e sigma
      in if is_empty a then bot
         else of_bool_option int_dom_of_int (map_option HOL.Not (int_dom_tobool a)))"
| "aval_int_dom mode (And e1 e2) sigma =
     (let a = aval_int_dom mode e1 sigma; b = aval_int_dom mode e2 sigma
      in if is_empty a ∨ is_empty b then bot
         else of_bool_option int_dom_of_int (and_opt (int_dom_tobool a) (int_dom_tobool b)))"
| "aval_int_dom mode (Or e1 e2) sigma =
     (let a = aval_int_dom mode e1 sigma; b = aval_int_dom mode e2 sigma
      in if is_empty a ∨ is_empty b then bot
         else of_bool_option int_dom_of_int (or_opt (int_dom_tobool a) (int_dom_tobool b)))"
```

`aval_int_dom` has exactly the shape `sound_arith_ops` assumes, once that locale's `pls`/`mns`/`tms` parameters are instantiated at this domain's mode-indexed operations instead of the type-class `+`/`-`/`*` the other four domains pass. So the shared induction over `exp` is reused here rather than repeated: what stays local is only the per-operation and per-query facts it consumes.

Soundness holds at every `mode`. Monotonicity holds only away from `Refine_Fixpoint`, so it is a second interpretation, inside a context that carries that side condition — which is what puts the condition on the exported `int_dom_mono_evaluator` and keeps it off `int_dom_sound_evaluator`.

```
lemma int_dom_truth_test: "mono_truth_test int_dom_tobool"
  by unfold_locales
     (auto dest: int_dom_tobool_sound intro: int_dom_tobool_mono)

context
  fixes mode :: refine_mode
begin

interpretation int_arith: sound_arith_ops
    "aval_int_dom mode" int_dom_of_int
    "plus_int_dom mode" "minus_int_dom mode" "times_int_dom mode" "div_int_dom mode"
      "mod_int_dom mode"
    int_dom_lt int_dom_eqb int_dom_tobool
  by unfold_locales
     (simp_all add: Let_def plus_int_dom_sound minus_int_dom_sound times_int_dom_sound
       div_int_dom_sound mod_int_dom_sound
                    int_dom_lt_sound int_dom_eqb_sound int_dom_tobool_sound)

lemma int_dom_sound_evaluator: "sound_evaluator gamma_state (aval_int_dom mode)"
  by unfold_locales (rule int_arith.aval_abs_sound)

end

context
  fixes mode :: refine_mode
  assumes mode_nonfixpoint: "mode ≠ Refine_Fixpoint"
begin

interpretation int_arith_mono: mono_arith_ops
    "aval_int_dom mode" int_dom_of_int
    "plus_int_dom mode" "minus_int_dom mode" "times_int_dom mode" "div_int_dom mode"
      "mod_int_dom mode"
    int_dom_lt int_dom_eqb int_dom_tobool
  by unfold_locales
     (simp_all add: Let_def plus_int_dom_sound minus_int_dom_sound times_int_dom_sound
       div_int_dom_sound mod_int_dom_sound
                    int_dom_lt_sound int_dom_eqb_sound int_dom_tobool_sound
                    plus_int_dom_mono[OF mode_nonfixpoint]
                    minus_int_dom_mono[OF mode_nonfixpoint]
                    times_int_dom_mono[OF mode_nonfixpoint]
                    div_int_dom_mono[OF mode_nonfixpoint]
                    mod_int_dom_mono[OF mode_nonfixpoint]
                    int_dom_lt_mono int_dom_eqb_mono int_dom_tobool_mono)


lemma int_dom_mono_evaluator: "mono_evaluator gamma_state (aval_int_dom mode)"
  by (intro mono_evaluator.intro int_dom_sound_evaluator mono_evaluator_axioms.intro)
     (rule int_arith_mono.aval_abs_mono)

end

end
```

16.5 Int_Backward

```
theory Int_Backward
  imports
    Int_Arithmetic
    "Voblint_Analysis_Sign.Sign_Numeric_Queries"
    "Voblint_Analysis_Parity.Parity_Backward"
    "Voblint_Analysis_Parity.Parity_Numeric_Queries"
    "Voblint_Analysis_Congruence.Congruence_Backward"
begin
```

Composite integer-domain backward filtering

Composite backward inversion follows the same shape as composite forward arithmetic (`Int_Arithmetic`): a raw componentwise operator, reusing each component's own existing inverse where one exists, then a mode-aware wrapper that refines the returned candidates with `refine mode`.

No component here invents a new inverse operator. Sign and Interval both already fall back to the shared identity `inv_conservative` for `+`/`-`/`*` in their own theories (`Sign_Backward`, `Interval_Backward`); this composite reuses that same choice rather than inventing per-component arithmetic inversion the domains themselves do not have. Parity and Congruence invert arithmetic in their own theories, and the composite reuses both: `inv_plus_parity`, `inv_minus_parity`, `inv_times_parity` (`Parity_Backward`) and `inv_plus_congruence`, `inv_minus_congruence`, `inv_times_congruenc` (`Congruence_Backward`). A comparison leaves the Parity component unchanged, as Parity's own `less` inverse does; an equality narrows it through the composite `intersect_int_dom` on the true branch below.

Precision Sign and Interval cannot recover directly at inversion time is not lost: the mode-aware wrapper's `refine mode` step re-derives their bounds from the (possibly Congruence-tightened) returned operand, exactly as `refine_interval`/`refine_congruence` already do for forward arithmetic.

Composite semantic intersection

```
definition intersect_int_dom :: "int_dom => int_dom => int_dom" where
  "intersect_int_dom d1 d2 =
     d1⦇
       int_sign := int_sign d1 ⊓ int_sign d2,
       int_ivl := intersect_ivl (int_ivl d1) (int_ivl d2),
       int_parity := int_parity d1 ⊓ int_parity d2,
       int_congruence := int_congruence d1 ⊓ int_congruence d2
     ⦈"
```

`intersect_ivl`'s defining equation is globally tagged `[simp]` (`Interval_Lattice`), so plain `simp`/`auto` unfolds it to `normalize_ivl (a ⊓ b)` before `gamma_intersect_ivl_exact` or `intersect_ivl_le1`/`intersect_ivl_le2`/`intersect_ivl_mono` – all stated in terms of the abstract `intersect_ivl` – get a chance to match. `del: intersect_ivl_def` below keeps `intersect_ivl` opaque for exactly those calls, matching `is_bottom_int_dom_correct`'s own `simp only: gamma_intersect_ivl_exact ...` workaround in `Int_Lattice`.

```
lemma intersect_int_dom_sound:
  assumes "n ∈ gamma_int_dom a" and "n ∈ gamma_int_dom b"
  shows "n ∈ gamma_int_dom (intersect_int_dom a b)"
proof -
  have hn: "n ∈ gamma_sign (int_sign a)" "n ∈ gamma_ivl (int_ivl a)"
           "n ∈ gamma_parity (int_parity a)" "n ∈ gamma_congruence (int_congruence a)"
    using assms(1) by (simp_all add: gamma_int_dom_def)
  have hm: "n ∈ gamma_sign (int_sign b)" "n ∈ gamma_ivl (int_ivl b)"
           "n ∈ gamma_parity (int_parity b)" "n ∈ gamma_congruence (int_congruence b)"
    using assms(2) by (simp_all add: gamma_int_dom_def)
  have ivl_fact: "n ∈ gamma_ivl (intersect_ivl (int_ivl a) (int_ivl b))"
    unfolding intersect_ivl_def
    using meet_ivl_gamma[OF hn(2) hm(2)] normalize_ivl_gamma by simp
  have sign_fact: "n ∈ gamma_sign (int_sign a ⊓ int_sign b)"
    using hn(1) hm(1) by simp
  have parity_fact: "n ∈ gamma_parity (int_parity a ⊓ int_parity b)"
    using hn(3) hm(3) by simp
  have congruence_fact:
    "n ∈ gamma_congruence (int_congruence a ⊓ int_congruence b)"
    using hn(4) hm(4) by simp
  show ?thesis
    unfolding gamma_int_dom_def intersect_int_dom_def
    using sign_fact ivl_fact parity_fact congruence_fact
    by (simp del: intersect_ivl_def)
qed

lemma intersect_int_dom_le1: "intersect_int_dom a b ≤ a"
```

```
  unfolding intersect_int_dom_def less_eq_int_dom_ext_def
  by (simp add: inf_le1 intersect_ivl_le1
      del: intersect_ivl_def)

lemma intersect_int_dom_le2: "intersect_int_dom a b ≤ b"
  unfolding intersect_int_dom_def less_eq_int_dom_ext_def
  by (simp add: inf_le2 intersect_ivl_le2
      del: intersect_ivl_def)

lemma intersect_int_dom_mono:
  assumes "a1 ≤ a2" and "b1 ≤ b2"
  shows "intersect_int_dom a1 b1 ≤ intersect_int_dom a2 b2"
proof -
  have s: "int_sign a1 ≤ int_sign a2" "int_ivl a1 ≤ int_ivl a2"
          "int_parity a1 ≤ int_parity a2" "int_congruence a1 ≤ int_congruence a2"
    using assms(1) by (simp_all add: less_eq_int_dom_ext_def)
  have t: "int_sign b1 ≤ int_sign b2" "int_ivl b1 ≤ int_ivl b2"
          "int_parity b1 ≤ int_parity b2" "int_congruence b1 ≤ int_congruence b2"
    using assms(2) by (simp_all add: less_eq_int_dom_ext_def)
  show ?thesis
    unfolding intersect_int_dom_def less_eq_int_dom_ext_def
    using s t
    by (simp add: intersect_ivl_mono[OF s(2) t(2)] le_infI1 le_infI2
        del: intersect_ivl_def)
qed

definition intersect_int_dom_mode ::
    "refine_mode => int_dom => int_dom => int_dom"
where
  "intersect_int_dom_mode mode a b = refine mode (intersect_int_dom a b)"

lemma intersect_int_dom_mode_sound:
  assumes "n ∈ gamma_int_dom a" and "n ∈ gamma_int_dom b"
  shows "n ∈ gamma_int_dom (intersect_int_dom_mode mode a b)"
  unfolding intersect_int_dom_mode_def
  using intersect_int_dom_sound[OF assms] refine_exact by simp
```

`refine mode` is reductive/exact for every mode (`Int_Refinement`) but only monotone off `Refine_Fixpoint` (`Int_Arithmetic`'s `refine_nonfixpoint_mono`); these two helpers compose that fact with a reductive/monotone step once, for `intersect_int_dom_mode` and the mode-aware inverse operators below.

```
lemma refine_mode_reductive_trans:
  assumes "a ≤ d"
  shows "refine mode a ≤ d"
  using refine_reductive[of mode a] assms by (rule order_trans)

lemma refine_mode_mono_trans:
  assumes "mode ≠ Refine_Fixpoint" and "a ≤ b"
  shows "refine mode a ≤ refine mode b"
  using monoD[OF refine_nonfixpoint_mono[OF assms(1)] assms(2)] .

lemma intersect_int_dom_mode_reductive1:
  "intersect_int_dom_mode mode a b ≤ a"
  unfolding intersect_int_dom_mode_def
  using refine_mode_reductive_trans[OF intersect_int_dom_le1] .

lemma intersect_int_dom_mode_reductive2:
  "intersect_int_dom_mode mode a b ≤ b"
  unfolding intersect_int_dom_mode_def
  using refine_mode_reductive_trans[OF intersect_int_dom_le2] .

lemma intersect_int_dom_mode_mono:
  assumes "mode ≠ Refine_Fixpoint" and "a1 ≤ a2" and "b1 ≤ b2"
  shows "intersect_int_dom_mode mode a1 b1 ≤ intersect_int_dom_mode mode a2 b2"
  unfolding intersect_int_dom_mode_def
  using refine_mode_mono_trans[OF assms(1) intersect_int_dom_mono[OF assms(2,3)]] .
```

Raw componentwise inverse operators

The raw inverses run each component's backward operator on its own field and reassemble the record. A component with no useful inverse goes through `inv_conservative`, and a true equality intersects both operands with `intersect_int_dom`.

```
definition inv_less_int_dom_raw ::
    "bool => int_dom => int_dom => int_dom * int_dom"
where
```

```
  "inv_less_int_dom_raw res d1 d2 =
     (let (s1, s2) = inv_less_sign res (int_sign d1) (int_sign d2);
          (i1, i2) = inv_less_ivl res (int_ivl d1) (int_ivl d2);
          (c1, c2) = inv_less_congruence res (int_congruence d1) (int_congruence d2)
      in
        (d1(|int_sign := s1, int_ivl := i1, int_congruence := c1|),
         d2(|int_sign := s2, int_ivl := i2, int_congruence := c2|)))"

definition inv_eq_int_dom_raw ::
    "bool => int_dom => int_dom => int_dom * int_dom"
where
  "inv_eq_int_dom_raw res d1 d2 =
     (if res then (intersect_int_dom d1 d2, intersect_int_dom d1 d2)
      else
        (let (s1, s2) = inv_eq_sign False (int_sign d1) (int_sign d2);
             (i1, i2) = inv_eq_ivl False (int_ivl d1) (int_ivl d2);
             (c1, c2) = inv_eq_congruence False (int_congruence d1) (int_congruence d2)
         in
           (d1(|int_sign := s1, int_ivl := i1, int_congruence := c1|),
            d2(|int_sign := s2, int_ivl := i2, int_congruence := c2|))))"

definition inv_plus_int_dom_raw ::
    "int_dom => int_dom => int_dom => int_dom * int_dom"
where
  "inv_plus_int_dom_raw r d1 d2 =
     (let (s1, s2) = inv_conservative (int_sign r) (int_sign d1) (int_sign d2);
          (i1, i2) = inv_conservative (int_ivl r) (int_ivl d1) (int_ivl d2);
          (p1, p2) = inv_plus_parity (int_parity r) (int_parity d1) (int_parity d2);
          (c1, c2) =
            inv_plus_congruence (int_congruence r) (int_congruence d1) (int_congruence d2)
      in
        (d1(|int_sign := s1, int_ivl := i1, int_parity := p1, int_congruence := c1|),
         d2(|int_sign := s2, int_ivl := i2, int_parity := p2, int_congruence := c2|)))"

definition inv_minus_int_dom_raw ::
    "int_dom => int_dom => int_dom => int_dom * int_dom"
where
  "inv_minus_int_dom_raw r d1 d2 =
     (let (s1, s2) = inv_conservative (int_sign r) (int_sign d1) (int_sign d2);
          (i1, i2) = inv_conservative (int_ivl r) (int_ivl d1) (int_ivl d2);
          (p1, p2) = inv_minus_parity (int_parity r) (int_parity d1) (int_parity d2);
          (c1, c2) =
            inv_minus_congruence (int_congruence r) (int_congruence d1) (int_congruence d2)
      in
        (d1(|int_sign := s1, int_ivl := i1, int_parity := p1, int_congruence := c1|),
         d2(|int_sign := s2, int_ivl := i2, int_parity := p2, int_congruence := c2|)))"

definition inv_times_int_dom_raw ::
    "int_dom => int_dom => int_dom => int_dom * int_dom"
where
  "inv_times_int_dom_raw r d1 d2 =
     (let (s1, s2) = inv_conservative (int_sign r) (int_sign d1) (int_sign d2);
          (i1, i2) = inv_conservative (int_ivl r) (int_ivl d1) (int_ivl d2);
          (p1, p2) = inv_times_parity (int_parity r) (int_parity d1) (int_parity d2);
          (c1, c2) =
            inv_times_congruence (int_congruence r) (int_congruence d1) (int_congruence d2)
      in
        (d1(|int_sign := s1, int_ivl := i1, int_parity := p1, int_congruence := c1|),
         d2(|int_sign := s2, int_ivl := i2, int_parity := p2, int_congruence := c2|)))"
```

Raw soundness

A raw inverse keeps every concrete operand pair consistent with the observed outcome: if `x` and `y` lie in the inputs and produce the comparison result or the value in `r`, they lie in the refined pair. Each proof cites the component soundness lemmas.

```
lemma inv_less_int_dom_raw_sound:
  assumes "x ∈ gamma_int_dom d1" and "y ∈ gamma_int_dom d2" and "(x < y) = res"
  shows
    "x ∈ gamma_int_dom (fst (inv_less_int_dom_raw res d1 d2)) ∧
     y ∈ gamma_int_dom (snd (inv_less_int_dom_raw res d1 d2))"
  using assms(1,2) inv_less_sign_sound[OF _ _ assms(3)] inv_less_ivl_sound[OF _ _ assms(3)]
  by (simp add: inv_less_int_dom_raw_def Let_def case_prod_beta gamma_int_dom_def
      inv_less_congruence_def)

lemma inv_eq_int_dom_raw_sound:
```

```
  assumes "x ∈ gamma_int_dom d1" and "y ∈ gamma_int_dom d2" and "(x = y) = res"
  shows
    "x ∈ gamma_int_dom (fst (inv_eq_int_dom_raw res d1 d2)) ∧
     y ∈ gamma_int_dom (snd (inv_eq_int_dom_raw res d1 d2))"
proof (cases res)
  case True
  then show ?thesis
    using assms intersect_int_dom_sound[OF assms(1), of d2] by (simp add: inv_eq_int_dom_raw_def)
next
  case False
  then show ?thesis
    using assms(1,2) inv_eq_sign_sound[OF _ _ assms(3)] inv_eq_ivl_sound[OF _ _ assms(3)]
      inv_eq_congruence_sound[OF _ _ assms(3)]
    by (simp add: inv_eq_int_dom_raw_def Let_def case_prod_beta gamma_int_dom_def
        del: inv_eq_sign.simps)
qed

lemma inv_plus_int_dom_raw_sound:
  assumes "x ∈ gamma_int_dom d1" and "y ∈ gamma_int_dom d2" and "x + y ∈ gamma_int_dom r"
  shows
    "x ∈ gamma_int_dom (fst (inv_plus_int_dom_raw r d1 d2)) ∧
     y ∈ gamma_int_dom (snd (inv_plus_int_dom_raw r d1 d2))"
  using assms inv_plus_congruence_sound[of x _ y _ "int_congruence r"]
    inv_plus_parity_sound[of x _ y _ "int_parity r"]
  by (simp add: inv_plus_int_dom_raw_def Let_def case_prod_beta gamma_int_dom_def
      inv_conservative_def)

lemma inv_minus_int_dom_raw_sound:
  assumes "x ∈ gamma_int_dom d1" and "y ∈ gamma_int_dom d2" and "x - y ∈ gamma_int_dom r"
  shows
    "x ∈ gamma_int_dom (fst (inv_minus_int_dom_raw r d1 d2)) ∧
     y ∈ gamma_int_dom (snd (inv_minus_int_dom_raw r d1 d2))"
  using assms inv_minus_congruence_sound[of x _ y _ "int_congruence r"]
    inv_minus_parity_sound[of x _ y _ "int_parity r"]
  by (simp add: inv_minus_int_dom_raw_def Let_def case_prod_beta gamma_int_dom_def
      inv_conservative_def)

lemma inv_times_int_dom_raw_sound:
  assumes "x ∈ gamma_int_dom d1" and "y ∈ gamma_int_dom d2" and "x * y ∈ gamma_int_dom r"
  shows
    "x ∈ gamma_int_dom (fst (inv_times_int_dom_raw r d1 d2)) ∧
     y ∈ gamma_int_dom (snd (inv_times_int_dom_raw r d1 d2))"
  using assms inv_times_congruence_sound[of x _ y _ "int_congruence r"]
    inv_times_parity_sound[of x _ y _ "int_parity r"]
  by (simp add: inv_times_int_dom_raw_def Let_def case_prod_beta gamma_int_dom_def
      inv_conservative_def)
```

Raw monotonicity

The raw inverses are monotone in every argument, stated as `le_pair` on the refined pair and proved from the componentwise monotonicity lemmas.

```
lemma inv_less_int_dom_raw_mono:
  assumes "d1 ≤ d2" and "e1 ≤ e2"
  shows "le_pair (inv_less_int_dom_raw res d1 e1) (inv_less_int_dom_raw res d2 e2)"
  using assms
  by (simp add: inv_less_int_dom_raw_def Let_def case_prod_beta less_eq_int_dom_ext_def
      inv_less_sign_mono inv_less_ivl_mono inv_less_congruence_def)

lemma inv_eq_int_dom_raw_mono:
  assumes "d1 ≤ d2" and "e1 ≤ e2"
  shows "le_pair (inv_eq_int_dom_raw res d1 e1) (inv_eq_int_dom_raw res d2 e2)"
proof (cases res)
  case True
  then show ?thesis
    using intersect_int_dom_mono[OF assms] by (simp add: inv_eq_int_dom_raw_def)
next
  case False
  then show ?thesis
    using assms
    by (simp add: inv_eq_int_dom_raw_def Let_def case_prod_beta less_eq_int_dom_ext_def
        inv_eq_sign_mono inv_eq_ivl_mono inv_eq_congruence_mono del: inv_eq_sign.simps)
qed

lemma inv_plus_int_dom_raw_mono:
  assumes "r1 ≤ r2" and "d1 ≤ d2" and "e1 ≤ e2"
```

```
  shows "le_pair (inv_plus_int_dom_raw r1 d1 e1) (inv_plus_int_dom_raw r2 d2 e2)"
  using assms
  by (simp add: inv_plus_int_dom_raw_def Let_def case_prod_beta less_eq_int_dom_ext_def
      inv_conservative_def inv_plus_congruence_mono inv_plus_parity_mono)

lemma inv_minus_int_dom_raw_mono:
  assumes "r1 ≤ r2" and "d1 ≤ d2" and "e1 ≤ e2"
  shows "le_pair (inv_minus_int_dom_raw r1 d1 e1) (inv_minus_int_dom_raw r2 d2 e2)"
  using assms
  by (simp add: inv_minus_int_dom_raw_def Let_def case_prod_beta less_eq_int_dom_ext_def
      inv_conservative_def inv_minus_congruence_mono inv_minus_parity_mono)

lemma inv_times_int_dom_raw_mono:
  assumes "r1 ≤ r2" and "d1 ≤ d2" and "e1 ≤ e2"
  shows "le_pair (inv_times_int_dom_raw r1 d1 e1) (inv_times_int_dom_raw r2 d2 e2)"
  using assms
  by (simp add: inv_times_int_dom_raw_def Let_def case_prod_beta less_eq_int_dom_ext_def
      inv_conservative_def inv_times_congruence_mono inv_times_parity_mono)
```

Mode-aware wrappers

`inv_less_int_dom` and its siblings apply `refine mode` to both halves of the raw result, so backward filtering follows the same refinement policy as the forward arithmetic.

```
definition inv_less_int_dom ::
    "refine_mode => bool => int_dom => int_dom => int_dom * int_dom"
where
  "inv_less_int_dom mode res d1 d2 =
     (let (r1, r2) = inv_less_int_dom_raw res d1 d2
      in (refine mode r1, refine mode r2))"

definition inv_eq_int_dom ::
    "refine_mode => bool => int_dom => int_dom => int_dom * int_dom"
where
  "inv_eq_int_dom mode res d1 d2 =
     (let (r1, r2) = inv_eq_int_dom_raw res d1 d2
      in (refine mode r1, refine mode r2))"

definition inv_plus_int_dom ::
    "refine_mode => int_dom => int_dom => int_dom => int_dom * int_dom"
where
  "inv_plus_int_dom mode r d1 d2 =
     (let (r1, r2) = inv_plus_int_dom_raw r d1 d2
      in (refine mode r1, refine mode r2))"

definition inv_minus_int_dom ::
    "refine_mode => int_dom => int_dom => int_dom => int_dom * int_dom"
where
  "inv_minus_int_dom mode r d1 d2 =
     (let (r1, r2) = inv_minus_int_dom_raw r d1 d2
      in (refine mode r1, refine mode r2))"

definition inv_times_int_dom ::
    "refine_mode => int_dom => int_dom => int_dom => int_dom * int_dom"
where
  "inv_times_int_dom mode r d1 d2 =
     (let (r1, r2) = inv_times_int_dom_raw r d1 d2
      in (refine mode r1, refine mode r2))"
```

Each wrapper refines both raw candidates, so its soundness and monotonicity reduce to the raw operator's through `refine_exact` and `refine_nonfixpoint_mono`.

```
lemma inv_int_dom_map_prod:
  "inv_less_int_dom mode res d1 d2 =
     map_prod (refine mode) (refine mode) (inv_less_int_dom_raw res d1 d2)"
  "inv_eq_int_dom mode res d1 d2 =
     map_prod (refine mode) (refine mode) (inv_eq_int_dom_raw res d1 d2)"
  "inv_plus_int_dom mode r d1 d2 =
     map_prod (refine mode) (refine mode) (inv_plus_int_dom_raw r d1 d2)"
  "inv_minus_int_dom mode r d1 d2 =
     map_prod (refine mode) (refine mode) (inv_minus_int_dom_raw r d1 d2)"
  "inv_times_int_dom mode r d1 d2 =
     map_prod (refine mode) (refine mode) (inv_times_int_dom_raw r d1 d2)"
  by (simp_all add: inv_less_int_dom_def inv_eq_int_dom_def inv_plus_int_dom_def
      inv_minus_int_dom_def inv_times_int_dom_def map_prod_def case_prod_beta)
```

Backward-domain interpretation

Soundness and the reductiveness of `intersect` hold at every mode, so every mode interprets `sound_refinement` and shares the precise, dead-arm-eliminating `branch`/`branch_st`. Monotonicity needs `mode` $\neq$ `Refine_Fixpoint`: `refine_fix`'s total wrapper has no monotonicity theorem (`Int_Refinement`), a faithful transliteration of Goblint's `fixpoint` loop. So `Refine_Never` and `Refine_Once` interpret the full `mono_refinement`, and `Refine_Fixpoint` stays out of reach of `branch_mono` and the rest of the monotonicity layer.

```
lemma int_backward_domain:
  "sound_refinement (intersect_int_dom_mode mode) (aval_int_dom mode) int_dom_tobool
     (inv_less_int_dom mode) (inv_eq_int_dom mode)
     (inv_plus_int_dom mode) (inv_minus_int_dom mode) (inv_times_int_dom mode)"
proof (intro sound_refinement.intro sound_intersection.intro
    int_dom_sound_evaluator mono_truth_test.axioms(1)[OF int_dom_truth_test]
    sound_inverse_ops.intro sound_inverse_ops_axioms.intro)
qed (simp_all add: inv_int_dom_map_prod refine_exact intersect_int_dom_mode_sound
       inv_less_int_dom_raw_sound inv_eq_int_dom_raw_sound inv_plus_int_dom_raw_sound
       inv_minus_int_dom_raw_sound inv_times_int_dom_raw_sound
       intersect_int_dom_mode_reductive1 intersect_int_dom_mode_reductive2)

lemma int_dom_backward_domain_mono:
  assumes "mode ≠ Refine_Fixpoint"
  shows
    "mono_refinement (intersect_int_dom_mode mode) (aval_int_dom mode) int_dom_tobool
       (inv_less_int_dom mode) (inv_eq_int_dom mode)
       (inv_plus_int_dom mode) (inv_minus_int_dom mode) (inv_times_int_dom mode)"
proof (intro mono_refinement.intro int_backward_domain
    int_dom_mono_evaluator[OF assms] int_dom_truth_test mono_refinement_axioms.intro
    mono_intersection.intro mono_intersection_axioms.intro
    sound_intersection.intro)
qed (auto simp: assms inv_int_dom_map_prod refine_mode_mono_trans intersect_int_dom_mode_mono
       refine_exact intersect_int_dom_mode_sound
       intersect_int_dom_mode_reductive1 intersect_int_dom_mode_reductive2
       inv_less_int_dom_raw_mono inv_eq_int_dom_raw_mono
       inv_plus_int_dom_raw_mono inv_minus_int_dom_raw_mono inv_times_int_dom_raw_mono)
```

The refinement operations, per mode

The three refinement modes differ only in how often the four components are reduced against each other, so one record parametric in the mode carries all three. The guard filters and the branch are derived from it in `Int_Transfer`.

```
definition int_refine_ops :: "refine_mode ⇒ int_dom refine_ops" where
  "int_refine_ops mode =
     (|r_tobool = int_dom_tobool, r_inv_less = inv_less_int_dom mode,
      r_inv_eq = inv_eq_int_dom mode, r_inv_plus = inv_plus_int_dom mode,
      r_inv_minus = inv_minus_int_dom mode, r_inv_times = inv_times_int_dom mode,
      r_intersect = intersect_int_dom_mode mode|)"

lemma int_refine_ops_simps [simp]:
  "r_tobool (int_refine_ops mode) = int_dom_tobool"
  "r_inv_less (int_refine_ops mode) = inv_less_int_dom mode"
  "r_inv_eq (int_refine_ops mode) = inv_eq_int_dom mode"
  "r_inv_plus (int_refine_ops mode) = inv_plus_int_dom mode"
  "r_inv_minus (int_refine_ops mode) = inv_minus_int_dom mode"
  "r_inv_times (int_refine_ops mode) = inv_times_int_dom mode"
  "r_intersect (int_refine_ops mode) = intersect_int_dom_mode mode"
  by (simp_all add: int_refine_ops_def)
```

Comparison queries

A comparison is answered component by component, as Goblint's `IntDomTuple` answers `lt` and `eq`: it is decided when some component decides it on its own value, and no step here combines the components. The refinement mode therefore reaches a comparison only through its operands, which `aval_int_dom` reduces as the mode says; under `Refine_Never` a fact that only the reduced product knows is not available to a check.

```
definition int_less_true :: "int_dom ⇒ int_dom ⇒ bool" where
  "int_less_true a b ⟷
     sign_less_true (int_sign a) (int_sign b)
     ∨ interval_less_true (int_ivl a) (int_ivl b)
     ∨ parity_less_true (int_parity a) (int_parity b)
     ∨ congruence_lt (int_congruence a) (int_congruence b) = Some True"

definition int_less_false :: "int_dom ⇒ int_dom ⇒ bool" where
  "int_less_false a b ⟷
```

```
    sign_less_false (int_sign a) (int_sign b)
    ∨ interval_less_false (int_ivl a) (int_ivl b)
    ∨ parity_less_false (int_parity a) (int_parity b)
    ∨ congruence_lt (int_congruence a) (int_congruence b) = Some False"

definition int_eq_true :: "int_dom ⇒ int_dom ⇒ bool" where
  "int_eq_true a b ⟷
    sign_eq_true (int_sign a) (int_sign b)
    ∨ interval_eq_true (int_ivl a) (int_ivl b)
    ∨ parity_eq_true (int_parity a) (int_parity b)
    ∨ congruence_eqb (int_congruence a) (int_congruence b) = Some True"

definition int_eq_false :: "int_dom ⇒ int_dom ⇒ bool" where
  "int_eq_false a b ⟷
    sign_eq_false (int_sign a) (int_sign b)
    ∨ interval_eq_false (int_ivl a) (int_ivl b)
    ∨ parity_eq_false (int_parity a) (int_parity b)
    ∨ congruence_eqb (int_congruence a) (int_congruence b) = Some False"

lemma gamma_int_dom_components:
  assumes "i ∈ γ (a :: int_dom)"
  shows "i ∈ gamma_sign (int_sign a)" "i ∈ gamma_ivl (int_ivl a)"
    "i ∈ gamma_parity (int_parity a)" "i ∈ gamma_congruence (int_congruence a)"
  using assms by (simp_all add: gamma_int_dom_def)

lemma int_less_true_sound:
  assumes "int_less_true a b" and "i ∈ γ a" and "j ∈ γ b"
  shows "i < j"
  using assms(1) gamma_int_dom_components[OF assms(2)] gamma_int_dom_components[OF assms(3)]
  unfolding int_less_true_def
  by (auto dest: sign_less_true_sound interval_less_true_sound parity_less_true_sound
      congruence_lt_sound)

lemma int_less_false_sound:
  assumes "int_less_false a b" and "i ∈ γ a" and "j ∈ γ b"
  shows "¬ i < j"
  using assms(1) gamma_int_dom_components[OF assms(2)] gamma_int_dom_components[OF assms(3)]
  unfolding int_less_false_def
  by (auto dest: sign_less_false_sound interval_less_false_sound parity_less_false_sound
      congruence_lt_sound)

lemma int_eq_true_sound:
  assumes "int_eq_true a b" and "i ∈ γ a" and "j ∈ γ b"
  shows "i = j"
  using assms(1) gamma_int_dom_components[OF assms(2)] gamma_int_dom_components[OF assms(3)]
  unfolding int_eq_true_def
  by (auto dest: sign_eq_true_sound interval_eq_true_sound parity_eq_true_sound
      congruence_eqb_sound)

lemma int_eq_false_sound:
  assumes "int_eq_false a b" and "i ∈ γ a" and "j ∈ γ b"
  shows "i ≠ j"
  using assms(1) gamma_int_dom_components[OF assms(2)] gamma_int_dom_components[OF assms(3)]
  unfolding int_eq_false_def
  by (auto dest: sign_eq_false_sound interval_eq_false_sound parity_eq_false_sound
      congruence_eqb_sound)

global_interpretation int_dom_numeric_queries:
  numeric_query_judgments int_less_true int_less_false int_eq_true int_eq_false
  defines int_less = int_dom_numeric_queries.query_less
    and int_eq = int_dom_numeric_queries.query_eq
  by unfold_locales
     (fact int_less_true_sound, fact int_less_false_sound,
      fact int_eq_true_sound, fact int_eq_false_sound)

end
```

16.6 Int_Transfer

```
theory Int_Transfer
  imports
    Int_Backward
    "Voblint_Analysis_Sign.Sign_Special"
    "Voblint_Analysis_Parity.Parity_Special"
    "Voblint_Nonrelational.Nonrelational_Transfer"
begin
```

Composite integer-domain transfer functions

The composite Sign/Interval/Parity/Congruence domain is one `nonrelational_ops` bundle per refinement mode, `int_dom_ops mode`, and one interpretation of `sound_nonrelational_ops` parametric in the mode derives the filters, the branch, the check classifier and every transfer function from it. This theory states only what is Int's own: the `min`/`max` primitives, the special-call dispatch, and the certificates.

What stays asymmetric is monotonicity: `refine_fix` has no monotonicity theorem, so the interpretation is the sound one, and `int_dom_mono_ops` certifies the monotone bundle for `Refine_Never` and `Refine_Once` only.

Min/Max special-call primitives

Sign, Interval, and Parity each already implement a real `min`/`max` primitive for the `Nondet_Int`/`Min`/`Max` special-call dispatch (`sign_min`/`sign_max`, `ivl_min`/`ivl_max`, `parity_min`/`parity_max`). Congruence has no such primitive: the congruence class of a `min`/`max` result is not determined by the operands' congruence classes in general, so the congruence component stays `top` – conservative, matching Congruence's own choice for `narrow` in `Voblint_Analysis_Congruence.Congruence_Warrowing`. Mode-aware refinement then applies to the raw combination exactly as it does for `plus_int_dom`/`minus_int_dom`/`times_int_dom`.

```
definition int_dom_min_raw :: "int_dom => int_dom => int_dom" where
  "int_dom_min_raw a b =
     (top :: int_dom)⦇
       int_sign := sign_min (int_sign a) (int_sign b),
       int_ivl := ivl_min (int_ivl a) (int_ivl b),
       int_parity := parity_min (int_parity a) (int_parity b)
     ⦈"

definition int_dom_max_raw :: "int_dom => int_dom => int_dom" where
  "int_dom_max_raw a b =
     (top :: int_dom)⦇
       int_sign := sign_max (int_sign a) (int_sign b),
       int_ivl := ivl_max (int_ivl a) (int_ivl b),
       int_parity := parity_max (int_parity a) (int_parity b)
     ⦈"

lemma int_dom_min_raw_sound:
  assumes "i : gamma_int_dom a" and "j : gamma_int_dom b"
  shows "min i j : gamma_int_dom (int_dom_min_raw a b)"
  using assms
  by (auto simp: gamma_int_dom_def int_dom_min_raw_def top_int_dom_ext_def
        intro: sign_min_sound ivl_min_sound parity_min_sound)

lemma int_dom_max_raw_sound:
  assumes "i : gamma_int_dom a" and "j : gamma_int_dom b"
  shows "max i j : gamma_int_dom (int_dom_max_raw a b)"
  using assms
  by (auto simp: gamma_int_dom_def int_dom_max_raw_def top_int_dom_ext_def
        intro: sign_max_sound ivl_max_sound parity_max_sound)

lemma int_dom_min_raw_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows "int_dom_min_raw a1 b1 <= int_dom_min_raw a2 b2"
  using assms
  by (auto simp: int_dom_min_raw_def less_eq_int_dom_ext_def
        intro: sign_min_combine_mono ivl_min_combine_mono parity_min_combine_mono)

lemma int_dom_max_raw_mono:
  assumes "a1 <= a2" and "b1 <= b2"
  shows "int_dom_max_raw a1 b1 <= int_dom_max_raw a2 b2"
  using assms
  by (auto simp: int_dom_max_raw_def less_eq_int_dom_ext_def
        intro: sign_max_combine_mono ivl_max_combine_mono parity_max_combine_mono)
```

```
definition int_dom_min :: "refine_mode => int_dom => int_dom => int_dom" where
  "int_dom_min mode a b = refine mode (int_dom_min_raw a b)"

definition int_dom_max :: "refine_mode => int_dom => int_dom => int_dom" where
  "int_dom_max mode a b = refine mode (int_dom_max_raw a b)"

lemma int_dom_min_sound:
  assumes "i : gamma_int_dom a" and "j : gamma_int_dom b"
  shows "min i j : gamma_int_dom (int_dom_min mode a b)"
proof -
  have "min i j : gamma_int_dom (int_dom_min_raw a b)"
    by (rule int_dom_min_raw_sound[OF assms])
  then show ?thesis
    unfolding int_dom_min_def using refine_exact by simp
qed

lemma int_dom_max_sound:
  assumes "i : gamma_int_dom a" and "j : gamma_int_dom b"
  shows "max i j : gamma_int_dom (int_dom_max mode a b)"
proof -
  have "max i j : gamma_int_dom (int_dom_max_raw a b)"
    by (rule int_dom_max_raw_sound[OF assms])
  then show ?thesis
    unfolding int_dom_max_def using refine_exact by simp
qed

lemma int_dom_min_mono:
  assumes "mode ~= Refine_Fixpoint" and "a1 <= a2" and "b1 <= b2"
  shows "int_dom_min mode a1 b1 <= int_dom_min mode a2 b2"
proof -
  have raw: "int_dom_min_raw a1 b1 <= int_dom_min_raw a2 b2"
    by (rule int_dom_min_raw_mono[OF assms(2,3)])
  show ?thesis
    unfolding int_dom_min_def
    by (rule monoD[OF refine_nonfixpoint_mono[OF assms(1)] raw])
qed

lemma int_dom_max_mono:
  assumes "mode ~= Refine_Fixpoint" and "a1 <= a2" and "b1 <= b2"
  shows "int_dom_max mode a1 b1 <= int_dom_max mode a2 b2"
proof -
  have raw: "int_dom_max_raw a1 b1 <= int_dom_max_raw a2 b2"
    by (rule int_dom_max_raw_mono[OF assms(2,3)])
  show ?thesis
    unfolding int_dom_max_def
    by (rule monoD[OF refine_nonfixpoint_mono[OF assms(1)] raw])
qed
```

Special-call dispatch

`special_int_dom` havocs on `Nondet_Int` and applies the mode-aware `int_dom_min`/`int_dom_max` to the evaluated operands. `int_dom_special_ops` packages the pair: sound in every mode, monotone outside `Refine_Fixpoint`.

```
fun special_int_dom ::
    "refine_mode => special_call => vname => (vname => int_dom) => (vname => int_dom)"
where
  "special_int_dom mode Nondet_Int x d = d(x := top)"
| "special_int_dom mode (Min a b) x d =
     d(x := int_dom_min mode (aval_int_dom mode a d) (aval_int_dom mode b d))"
| "special_int_dom mode (Max a b) x d =
     d(x := int_dom_max mode (aval_int_dom mode a d) (aval_int_dom mode b d))"

definition int_dom_special_ops :: "refine_mode ⇒ int_dom special_ops" where
  "int_dom_special_ops mode =
     (| special_min = int_dom_min mode, special_max = int_dom_max mode |)"

lemma int_dom_special_ops_simps [simp]:
  "special_min (int_dom_special_ops mode) = int_dom_min mode"
  "special_max (int_dom_special_ops mode) = int_dom_max mode"
  by (simp_all add: int_dom_special_ops_def)

lemma int_dom_sound_special_ops: "sound_minmax_ops (int_dom_special_ops mode) (aval_int_dom mode)"
  by (intro sound_minmax_ops.intro int_dom_sound_evaluator sound_minmax_ops_axioms.intro)
     (simp_all add: int_dom_min_sound int_dom_max_sound)

lemma int_dom_mono_special_ops:
```

```
  assumes "mode ≠ Refine_Fixpoint"
  shows "mono_minmax_ops (int_dom_special_ops mode) (aval_int_dom mode)"
  by (intro mono_minmax_ops.intro int_dom_sound_special_ops int_dom_mono_evaluator[OF assms]
        mono_minmax_ops_axioms.intro)
     (simp_all add: int_dom_min_mono int_dom_max_mono assms)

lemma special_int_dom_eq_transfer:
  "sound_minmax_ops.special_transfer (int_dom_special_ops mode) (aval_int_dom mode) sc x d
     = special_int_dom mode sc x d"
  by (cases sc)
     (simp_all add: sound_minmax_ops.special_transfer_def[OF int_dom_sound_special_ops])
```

The bundle, per refinement mode, and its certificates

One bundle parametric in the mode: the evaluator, the special calls and the refinement operations carry it, the queries and the whole-value element do not. Every mode is sound; only `Refine_Never` and `Refine_Once` are monotone, because `refine_fix` has no monotonicity theorem (`Int_Refinement`).

```
definition int_dom_ops :: "refine_mode ⇒ int_dom nonrelational_ops" where
  "int_dom_ops mode =
     (| n_aval = aval_int_dom mode, n_query = (|q_less = int_less, q_eq = int_eq|),
       n_refine = int_refine_ops mode, n_special = int_dom_special_ops mode |)"

lemma int_dom_ops_simps [simp]:
  "n_aval (int_dom_ops mode) = aval_int_dom mode"
  "q_less (n_query (int_dom_ops mode)) = int_less"
  "q_eq (n_query (int_dom_ops mode)) = int_eq"
  "n_refine (int_dom_ops mode) = int_refine_ops mode"
  "n_special (int_dom_ops mode) = int_dom_special_ops mode"
  by (simp_all add: int_dom_ops_def)

lemma int_check_domain:
  "sound_check_query int_less int_eq gamma_state (aval_int_dom mode)"
  by (intro sound_check_query.intro int_dom_numeric_queries.sound_numeric_queries_axioms
        int_dom_sound_evaluator)

lemma int_dom_sound_ops: "sound_nonrelational_ops (int_dom_ops mode)"
  by (rule sound_nonrelational_opsI; unfold int_dom_ops_simps int_refine_ops_simps)
     (rule int_dom_sound_special_ops int_backward_domain int_check_domain)+

lemma int_dom_mono_ops:
  assumes "mode ≠ Refine_Fixpoint"
  shows "mono_nonrelational_ops (int_dom_ops mode)"
  by (rule mono_nonrelational_opsI; unfold int_dom_ops_simps int_refine_ops_simps)
     (rule int_dom_mono_special_ops[OF assms] int_dom_backward_domain_mono[OF assms]
        int_check_domain)+

global_interpretation int_tf: sound_nonrelational_ops "int_dom_ops mode"
  rewrites "sound_minmax_ops.special_transfer (n_special (int_dom_ops mode)) (n_aval (int_dom_ops mode))
           = special_int_dom mode"
  defines assign_int_dom = int_tf.assign
    and skip_int_dom = int_tf.skip
    and body_int_dom = int_tf.body
    and event_int_dom = int_tf.event
    and return_int_dom = int_tf.ret
    and enter_frame_int_dom_for = int_tf.enter_frame_for
    and enter_int_dom_for = int_tf.enter_for
    and enter_int_dom_ci_for = int_tf.enter_ci_for
    and int_tf_abs = int_tf.tf_abs
    and afilter_int_dom = int_tf.backward.afilter
    and feasible_int_dom = int_tf.backward.feasible
    and bfilter_int_dom = int_tf.backward.bfilter
    and branch_int_dom_for = int_tf.backward.branch
    and bfilter_lifted_int_dom = int_tf.backward.bfilter_lifted
    and branch_lifted_int_dom = int_tf.backward.branch_lifted
    and int_truthy_query = int_tf.check.truthy_query
    and int_check_query = int_tf.check.check_query
    and int_classify_check = int_tf.check.classify_check
    and int_eval_answer = int_tf.check.eval_answer
    and int_checks_proven = int_tf.check.abstract_checks_proven
  by (rule int_dom_sound_ops) (intro ext, simp add: special_int_dom_eq_transfer)
```

The theorems stay under `int_tf.`: `skip_int_dom`'s soundness is `int_tf.skip_sound`, the branch's is `int_tf.backward.branch_sou` and the framework's transfer contract at every mode is `int_tf.is_sound_nonrelational_transfer`.

```
end
```

16.7 Int_Exec

```
theory Int_Exec
  imports
    Int_Transfer
begin
```

The composite integer domain on the executable carrier

The executable step and procedure entry are `Voblint_Nonrelational.Nonrelational_Ops`'s generic constructions at the bundle, `generic_tf_st_for (int_dom_ops mode)` and `generic_enter_st_for (int_dom_ops mode)`, and their agreement with the abstract transfer is `int_tf.tf_st_for_commute` and `int_tf.enter_st_for_commute`: nothing about them is the domain's to state. The mode is an argument like any other, chosen by public production reporting; it is orthogonal to equation generation and solver choice.

A declared global holds the abstraction of `0`, a local the whole-value element.

```
abbreviation cinit_int_dom_st :: "int_dom default_st" where
  "cinit_int_dom_st ≡ initial_default_st top (int_dom_of_int 0)"

end
```

16.8 Int_Sound

```
theory Int_Sound
  imports
    Int_Classify
begin
```

What Int supplies to the framework

Two reader-commutation facts and an initial-state contract. None of the three mentions a context, a routing function, a seed key, or a solver. The commutation facts are `Voblint_Analysis_Int.Int_Exec`'s `int_tf.tf_st_for_commute` and `int_tf.enter_st_for_commute`, proved once for every refinement mode.

Int's specification, its concretization and the soundness of the one against the other are not restated here. Those are constants and theorems of `dg_domain_exec`, which an assembly reaches by interpreting that locale from the commutation facts — so a per-domain copy would only rename what the locale already proves.

What the initial abstract state describes

The stores a run may start in are described by the function the entry state represents. This mentions neither a solver nor a coverage assumption — only the entry state and the global-variable predicate — so it is stated here rather than inside a solved-system context.

```
lemma int_cinit_gamma:
  "cinit_stores 𝒢 ⊆ default_st_gamma 𝒢 cinit_int_dom_st"
  by (auto simp: cinit_stores_def gamma_state_def default_st_gamma_initial
      gamma_top [where 'a = int_dom, unfolded gamma_abs_int_dom_ext])

end
```

16.9 Int_Classify

theory Int_Classify
  imports Int_Exec
begin

Int instance of the generic check-discharge interface

The check classifier is derived from Int's bundle: the Boolean recursion over `exp`, the three-way classification, and the node-indexed bridge to `checks_proven` come from the comparison queries `int_less`/`int_eq` and the mode's evaluator, through the `int_tf` interpretation in `Voblint_Analysis_Int.Int_Transfer`. Their soundness is `int_tf.check.classify_check_proved` and `int_tf.check.classify_check_refuted`.

end

16.10 Int_Fixpoint_Analyses

```
theory Int_Fixpoint_Analyses
  imports
    Int_Sound
    "Voblint_Nonrelational.Nonrel_Registration"
begin
```

Registering Int under `Refine_Fixpoint` at every context and update rule

Int under `Refine_Fixpoint` runs through the shared D/G pipeline at the unit context. The CLI runs it as a field of the combined state of `MCP_Analyses`, whose component and soundness the unit registration supplies. Each registration leaves the rule that merges a value side-effected into a global as a parameter `r`. The equation system, the solve, the result table and every soundness endpoint come from the interpreted locale; this theory only names the domain's own implementation and facts.

At the unit context

```
global_interpretation int_fixpoint_rule:
    nonrel_unit_rule "int_dom_ops Refine_Fixpoint" cinit_int_dom_st r
  for r
  by (intro nonrel_unit_ruleI nonrel_registrationI
      int_tf.sound_nonrelational_ops_axioms int_cinit_gamma)

end
```

16.11 Int_Once_Analyses

```
theory Int_Once_Analyses
  imports
    Int_Sound
    "Voblint_Nonrelational.Nonrel_Registration"
begin
```

Registering Int under `Refine_Once` at every context and update rule

Int under `Refine_Once` runs through the shared D/G pipeline at the unit context. The CLI runs it as a field of the combined state of `MCP_Analyses`, whose component and soundness the unit registration supplies. Each registration leaves the rule that merges a value side-effected into a global as a parameter `r`. The equation system, the solve, the result table and every soundness endpoint come from the interpreted locale; this theory only names the domain's own implementation and facts.

At the unit context

```
global_interpretation int_once_rule:
    nonrel_unit_rule "int_dom_ops Refine_Once" cinit_int_dom_st r
  for r
  by (intro nonrel_unit_ruleI nonrel_registrationI
      int_tf.sound_nonrelational_ops_axioms int_cinit_gamma)

end
```

16.12 Int_Never_Analyses

```
theory Int_Never_Analyses
  imports
    Int_Sound
    "Voblint_Nonrelational.Nonrel_Registration"
begin
```

Registering Int under `Refine_Never` at every context and update rule

Int under `Refine_Never` runs through the shared D/G pipeline at the unit context. The CLI runs it as a field of the combined state of `MCP_Analyses`, whose component and soundness the unit registration supplies. Each registration leaves the rule that merges a value side-effected into a global as a parameter `r`. The equation system, the solve, the result table and every soundness endpoint come from the interpreted locale; this theory only names the domain's own implementation and facts.

At the unit context

```
global_interpretation int_never_rule:
    nonrel_unit_rule "int_dom_ops Refine_Never" cinit_int_dom_st r
  for r
  by (intro nonrel_unit_ruleI nonrel_registrationI
      int_tf.sound_nonrelational_ops_axioms int_cinit_gamma)

end
```

Chapter 17

Voblint_CLI

17.1 Analysis_Config

```
theory Analysis_Config
  imports "Voblint_Solver.Globals_Rule"
begin
```

What a caller chooses

Three choices change what gets solved: which analyses run, the rule that merges contributions to a side-effected global (`globals_rule`), and how activations are told apart. Each is a plain value, and every combination is analysed: there is no default a caller inherits and no pairing that is refused. Presentation choices are absent — they select how a solved result is drawn, never what is solved, and stay with the handwritten renderers.

The analyses a caller may activate are generated from the analysis manifest into `MCP_Carrier`; this theory holds the other two choices.

A call string of length zero is an ordinary context policy: every activation shares the empty context, over an equation system that is still call-string keyed.

```
datatype context_mode = Ctx_None | Ctx_EntryState | Ctx_CallString nat
```

```
end
```

17.2 Dispatch_Carrier

```
theory Dispatch_Carrier
  imports
    "Voblint_Domain.Int_Lattice"
    "Voblint_Domain.Order_Lattice"
    "Voblint_CFG.CFG_Def"
begin
```

What each domain's published value needs to be shown and listed

A run result carries every published value in one tagged union, `abstract_value`, which `MCP_Carrier` generates from the analysis manifest with one constructor per registered analysis. What the union needs from each domain is domain content and stays here: how an order value is listed, and an injective key for every domain's value.

An order value is listed as its pairs, or `None` where the state is unreachable, so that a key can tell two apart.

```
definition order_pairs :: "relc ⇒ (vname × vname) list option" where
  "order_pairs d = (case d of RelBot ⇒ None | RelC ps ⇒ Some (sorted_list_of_set ps))"
```

A listed order reads as the order it lists. The listing and not the order is what the union carries, since only a finite listing has an injective key.

```
definition order_view_string :: "(vname × vname) list option ⇒ String.literal" where
  "order_view_string v = to_string (case v of None ⇒ RelBot | Some ps ⇒ RelC (set ps))"
```

What one analysis shows of a state, as Goblint's report shows each component of its combined state on its own: a pointwise analysis a value per variable, an analysis whose state relates variables one value for the whole state.

```
datatype 'v field_state = Field_Store "(vname × 'v) list" | Field_Whole 'v

fun map_field_state :: "('v ⇒ 'w) ⇒ 'v field_state ⇒ 'w field_state" where
  "map_field_state f (Field_Store bs) = Field_Store (map (λ(x, v). (x, f v)) bs)"
| "map_field_state f (Field_Whole v) = Field_Whole (f v)"
```

Listing a set of contexts without reading its values' order

A solved table covers a *set* of contexts at each point, and a result handed to a caller lists them. Listing needs a linear order, but a domain's value type already spends its `ord` instance on the abstraction order, under which `[0,1]` and `[2,3]` are incomparable. `order_key` is a separate type with a derived linear order, and `abstract_value_key` maps every value into it injectively. Because the map is injective, listing by it drops no context — and because it reads the value's structure rather than its rendering, no printer decides what is listed.

```
datatype order_key = Key_Int int | Key_Node cfg_node | Key_List "order_key list"

derive linorder order_key

fun sign_key :: "sign ⇒ order_key" where
  "sign_key SBot = Key_Int 0"
| "sign_key SNeg = Key_Int 1"
| "sign_key SNonPos = Key_Int 2"
| "sign_key SZero = Key_Int 3"
| "sign_key SNonNeg = Key_Int 4"
| "sign_key SPos = Key_Int 5"
| "sign_key STop = Key_Int 6"

fun parity_key :: "parity ⇒ order_key" where
  "parity_key PBot = Key_Int 0"
| "parity_key PEven = Key_Int 1"
| "parity_key POdd = Key_Int 2"
| "parity_key PTop = Key_Int 3"

fun eint_key :: "eint ⇒ order_key" where
  "eint_key MinInf = Key_List [Key_Int 0]"
| "eint_key (Fin n) = Key_List [Key_Int 1, Key_Int n]"
| "eint_key PlusInf = Key_List [Key_Int 2]"

fun ivl_key :: "ivl ⇒ order_key" where
  "ivl_key (Ivl l u) = Key_List [eint_key l, eint_key u]"

definition congruence_key :: "congruence ⇒ order_key" where
  "congruence_key v =
     (case Rep_congruence v of
        None ⇒ Key_List []
      | Some (r, m) ⇒ Key_List [Key_Int r, Key_Int m])"
```

```
definition int_dom_key :: "int_dom ⇒ order_key" where
  "int_dom_key d =
     Key_List [sign_key (int_sign d), ivl_key (int_ivl d),
               parity_key (int_parity d), congruence_key (int_congruence d)]"

definition literal_key :: "String.literal ⇒ order_key" where
  "literal_key s = Key_List (map (λc. Key_Int (of_char c)) (String.explode s))"

definition order_view_key :: "(vname × vname) list option ⇒ order_key" where
  "order_view_key v =
     (case v of
        None ⇒ Key_List []
      | Some ps ⇒ Key_List [Key_List (map (λ(a, b). Key_List [literal_key a, literal_key b]) ps)])"

lemma sign_key_inject [simp]: "sign_key a = sign_key b ⟷ a = b"
  by (cases a; cases b) simp_all

lemma parity_key_inject [simp]: "parity_key a = parity_key b ⟷ a = b"
  by (cases a; cases b) simp_all

lemma eint_key_inject [simp]: "eint_key a = eint_key b ⟷ a = b"
  by (cases a; cases b) simp_all

lemma ivl_key_inject [simp]: "ivl_key a = ivl_key b ⟷ a = b"
  by (cases a; cases b) simp_all

lemma congruence_key_inject [simp]: "congruence_key a = congruence_key b ⟷ a = b"
  unfolding congruence_key_def
  by (auto simp flip: Rep_congruence_inject split: option.splits)

lemma int_dom_key_inject [simp]: "int_dom_key a = int_dom_key b ⟷ a = b"
  unfolding int_dom_key_def by (cases a; cases b) simp

lemma literal_key_inject [simp]: "literal_key a = literal_key b ⟷ a = b"
proof
  assume "literal_key a = literal_key b"
  then have "String.explode a = String.explode b"
    unfolding literal_key_def by (auto dest: list.inj_map_strong[rotated])
  then show "a = b" by (simp add: String.explode_inject)
qed simp

lemma order_view_key_inject [simp]: "order_view_key a = order_view_key b ⟷ a = b"
proof
  assume "order_view_key a = order_view_key b"
  then show "a = b"
    unfolding order_view_key_def
    by (auto split: option.splits dest!: list.inj_map_strong[rotated])
qed simp

end
```

17.3 MCP_Field

```
theory MCP_Field
  imports
    "Voblint_Framework.Local_Spec_Product"
    "Voblint_Framework.Channel_Wrappers"
    "Voblint_Framework.Solved_Table"
    "Voblint_Exec.DG_Local_State_Exec_Refinement"
begin
```

One analysis on one field of the combined state

The combined state nests one reachability-lifted field per registered analysis in `('a, 'b) analysis_products`. An analysis runs on its field through `lift_get` and `lift_put`; the two lemmas below turn its own soundness into soundness on the field, and show that it leaves every other field alone. They are stated once for any field, so the generated carrier only cites them.

```
lemma pleft_pright_bot [simp]:
  "pleft (⊥ :: ('a::order_bot, 'b::order_bot) analysis_product) = ⊥"
  "pright (⊥ :: ('a::order_bot, 'b::order_bot) analysis_product) = ⊥"
  by (simp_all add: bot_analysis_product_def)

lemma field_component_sound:
  fixes f :: "'r::{semilattice_sup, order_bot} ⇒ 'c::order_bot"
  assumes sound: "sound_local_spec 𝒢 g cmp"
    and get_put: "⋀r v. f (u r v) = v" and get_bot: "f ⊥ = ⊥"
    and mono: "⋀r r'. r ≤ r' ⟹ f r ≤ f r'"
  shows "sound_local_spec 𝒢 (λx. g (lift_get f x)) (lens_of (lift_get f) (lift_put u) cmp)"
  by (rule lens_of_sound[OF sound, where get = "lift_get f"])
     (simp_all add: lift_get_put[OF get_put get_bot] lift_get_mono[OF mono])
```

A field's concretization that no write to another field changes is left alone by that field's component.

```
lemma field_frame:
  assumes "⋀x v. g (lift_put u x v) = g x"
  shows "mcp_frame (lens_of get (lift_put u) cmp) g"
  unfolding mcp_frame_def lens_of_def by (auto simp: assms)

lemma single_entry_exec_local_spec:
  "single_entry (exec_local_spec 𝒢 empty_pred tf_st enter_st)"
  by (simp add: single_entry_def exec_local_spec_def)

end
```

17.4 MCP_Carrier

```
theory MCP_Carrier
  imports
    "Voblint_CLI.Analysis_Config"
    "Voblint_CLI.Dispatch_Carrier"
    "Voblint_Analysis_Sign.Sign_Analyses"
    "Voblint_Analysis_Interval.Interval_Analyses"
    "Voblint_Analysis_Parity.Parity_Analyses"
    "Voblint_Analysis_Int.Int_Fixpoint_Analyses"
    "Voblint_Analysis_Int.Int_Once_Analyses"
    "Voblint_Analysis_Int.Int_Never_Analyses"
    "Voblint_Analysis_Congruence.Congruence_Analyses"
    "Voblint_Analysis_Relational.Rel_Order_Local"
    "Voblint_CLI.MCP_Field"
begin
```

The combined state of the registered analyses

Every registered analysis owns one field of the combined state. This theory names the analyses, lays out the fields, and states for each analysis how its field runs, what it describes, how it reads back and answers queries, and where it starts. Nothing here is proved beyond citing each analysis's own registration; the combination and its soundness are in `MCP_Analyses`.

A new analysis adds a constructor to each datatype below, a field to the product types, and one equation to each function that splits on the analysis.

```
datatype analysis_domain =
    Sign_Analysis
  | Interval_Analysis
  | Parity_Analysis
  | Int_Analysis "refine_mode"
  | Congruence_Analysis
  | Order_Analysis

lemma analysis_domain_cases:
  obtains "a = Sign_Analysis"
  | "a = Interval_Analysis"
  | "a = Parity_Analysis"
  | "a = (Int_Analysis Refine_Fixpoint)"
  | "a = (Int_Analysis Refine_Once)"
  | "a = (Int_Analysis Refine_Never)"
  | "a = Congruence_Analysis"
  | "a = Order_Analysis"
  by (cases a) (metis refine_mode.exhaust)+
```

One value type wide enough for every analysis

A run result crosses the dispatcher without its caller knowing which analysis produced it, so every published value in it has one type: a tagged union with one constructor per registered analysis. Each is rendered by its own analysis and listed by an injective key, so no printer decides which contexts are listed.

```
datatype abstract_value =
    SignValue "sign"
  | IntervalValue "ivl"
  | ParityValue "parity"
  | IntDomValue "int_dom"
  | IntDomOnceValue "int_dom"
  | IntDomNeverValue "int_dom"
  | CongruenceValue "congruence"
  | OrderValue "(vname × vname) list option"

fun string_of_abstract_value :: "abstract_value ⇒ String.literal" where
  "string_of_abstract_value (SignValue v) =
     to_string v"
| "string_of_abstract_value (IntervalValue v) =
     to_string v"
| "string_of_abstract_value (ParityValue v) =
     to_string v"
| "string_of_abstract_value (IntDomValue v) =
     to_string v"
| "string_of_abstract_value (IntDomOnceValue v) =
     to_string v"
| "string_of_abstract_value (IntDomNeverValue v) =
     to_string v"
```

```
| "string_of_abstract_value (CongruenceValue v) =
     to_string v"
| "string_of_abstract_value (OrderValue v) =
     order_view_string v"

fun abstract_value_key :: "abstract_value ⇒ order_key" where
  "abstract_value_key (SignValue v) =
     Key_List [Key_Int 0, sign_key v]"
| "abstract_value_key (IntervalValue v) =
     Key_List [Key_Int 1, ivl_key v]"
| "abstract_value_key (ParityValue v) =
     Key_List [Key_Int 2, parity_key v]"
| "abstract_value_key (IntDomValue v) =
     Key_List [Key_Int 3, int_dom_key v]"
| "abstract_value_key (IntDomOnceValue v) =
     Key_List [Key_Int 4, int_dom_key v]"
| "abstract_value_key (IntDomNeverValue v) =
     Key_List [Key_Int 5, int_dom_key v]"
| "abstract_value_key (CongruenceValue v) =
     Key_List [Key_Int 6, congruence_key v]"
| "abstract_value_key (OrderValue v) =
     Key_List [Key_Int 7, order_view_key v]"

lemma abstract_value_key_inject [simp]:
  "abstract_value_key a = abstract_value_key b ⟷ a = b"
  by (cases a; cases b) simp_all
```

Fields

The combined state nests binary products, so field `n` sits one `pright` deeper than field `n - 1`. The last field is the right component of the innermost product.

```
definition slot1 where
  "slot1 r = pleft r"

definition slot2 where
  "slot2 r = slot1 (pright r)"

definition slot3 where
  "slot3 r = slot2 (pright r)"

definition slot4 where
  "slot4 r = slot3 (pright r)"

definition slot5 where
  "slot5 r = slot4 (pright r)"

definition slot6 where
  "slot6 r = slot5 (pright r)"

definition slot7 where
  "slot7 r = slot6 (pright r)"

definition slot8 where
  "slot8 r = pright (pright (pright (pright (pright (pright (pright r))))))"

definition set_slot1 where
  "set_slot1 r v = Product v (pright r)"

definition set_slot2 where
  "set_slot2 r v = Product (pleft r) (set_slot1 (pright r) v)"

definition set_slot3 where
  "set_slot3 r v = Product (pleft r) (set_slot2 (pright r) v)"

definition set_slot4 where
  "set_slot4 r v = Product (pleft r) (set_slot3 (pright r) v)"

definition set_slot5 where
  "set_slot5 r v = Product (pleft r) (set_slot4 (pright r) v)"

definition set_slot6 where
  "set_slot6 r v = Product (pleft r) (set_slot5 (pright r) v)"

definition set_slot7 where
  "set_slot7 r v = Product (pleft r) (set_slot6 (pright r) v)"
```

```
definition set_slot8 where
  "set_slot8 r v = Product (slot1 r) (Product (slot2 r) (Product (slot3 r) (Product (slot4 r)
    (Product (slot5 r) (Product (slot6 r) (Product (slot7 r) v))))))"

lemmas slot_defs [simp] = slot1_def slot2_def slot3_def slot4_def slot5_def slot6_def
  slot7_def slot8_def set_slot1_def set_slot2_def set_slot3_def set_slot4_def
  set_slot5_def set_slot6_def set_slot7_def set_slot8_def

type_synonym mcp_fields =
  "(sign default_st lifted, (ivl default_st lifted, (parity default_st lifted, (int_dom
    default_st lifted, (int_dom default_st lifted, (int_dom default_st lifted, (congruence
    default_st lifted, relc) analysis_product) analysis_product) analysis_product)
    analysis_product) analysis_product) analysis_product) analysis_product"
```

The combined state wraps its fields in a type of its own. Exported code then names each class instance of the state once; over a bare nested product it would rebuild the product's instance in full at every use.

```
datatype mcp_st = MCP_St (mcp_fields: mcp_fields)

instantiation mcp_st :: ord
begin

definition less_eq_mcp_st :: "mcp_st ⇒ mcp_st ⇒ bool" where
  "less_eq_mcp_st r q ⟷ mcp_fields r ≤ mcp_fields q"

definition less_mcp_st :: "mcp_st ⇒ mcp_st ⇒ bool" where
  "less_mcp_st r q ⟷ mcp_fields r < mcp_fields q"

instance ..

end

instance mcp_st :: order
  by standard
    (auto simp: less_eq_mcp_st_def less_mcp_st_def less_le_not_le mcp_st.expand
      intro: order_trans antisym)

instantiation mcp_st :: semilattice_sup
begin

definition sup_mcp_st :: "mcp_st ⇒ mcp_st ⇒ mcp_st" where
  "sup_mcp_st r q = MCP_St (mcp_fields r ⊔ mcp_fields q)"

instance
  by standard (simp_all add: less_eq_mcp_st_def sup_mcp_st_def)

end

instantiation mcp_st :: order_bot
begin

definition bot_mcp_st :: mcp_st where
  "bot_mcp_st = MCP_St ⊥"

instance
  by standard (simp add: less_eq_mcp_st_def bot_mcp_st_def)

end

instance mcp_st :: bounded_semilattice_sup_bot ..

instantiation mcp_st :: warrowing
begin

definition widen_mcp_st :: "mcp_st ⇒ mcp_st ⇒ mcp_st" where
  "widen_mcp_st r q = MCP_St (mcp_fields r ∇ mcp_fields q)"

definition narrow_mcp_st :: "mcp_st ⇒ mcp_st ⇒ mcp_st" where
  "narrow_mcp_st r q = MCP_St (mcp_fields r △ mcp_fields q)"

instance
  by standard
    (simp_all add: less_eq_mcp_st_def widen_mcp_st_def narrow_mcp_st_def
      narrowing_class.narrow_ge narrowing_class.narrow_le
      widening_class.widen_ge1 widening_class.widen_ge2)
```

```
end

lemma MCP_St_le_MCP_St [simp]: "MCP_St a ≤ MCP_St b ⟷ a ≤ b"
  by (simp add: less_eq_mcp_st_def)

lemma mcp_fields_bot [simp]: "mcp_fields ⊥ = ⊥"
  by (simp add: bot_mcp_st_def)

lemma mcp_fields_sup [simp]: "mcp_fields (r ⊔ q) = mcp_fields r ⊔ mcp_fields q"
  by (simp add: sup_mcp_st_def)

type_synonym mcp_val =
  "(sign abs_state lifted, (ivl abs_state lifted, (parity abs_state lifted, (int_dom
    abs_state lifted, (int_dom abs_state lifted, (int_dom abs_state lifted, (congruence
    abs_state lifted, relc) analysis_product) analysis_product) analysis_product)
    analysis_product) analysis_product) analysis_product) analysis_product"

type_synonym mcp_ctx =
  "(sign list, (ivl list, (parity list, (int_dom list, (int_dom list, (int_dom list,
    (congruence list, unit list) analysis_product) analysis_product) analysis_product)
    analysis_product) analysis_product) analysis_product) analysis_product"
```

Each analysis on its own field

What the analyzer runs of one analysis: how its field steps, whether it is live or empty, how its published value answers a query and is displayed, and the values it keys a callee by. `registration_of` is the one function that splits on the analysis at runtime. The concretizations are proof-only and are not part of it: a record the exported code builds holds executable values only, so the emptiness test on published values is proof-only as well.

```
record analysis_registration =
  field_spec :: "(vname ⇒ bool) ⇒ imp_prog ⇒ mcp_st lifted local_spec"
  field_live :: "mcp_st ⇒ bool"
  field_empty :: "vname list ⇒ mcp_st ⇒ bool"
  value_answer :: "mcp_val ⇒ query ⇒ answer"
  value_display :: "mcp_val ⇒ vname list ⇒ abstract_value field_state"
  context_values :: "mcp_ctx ⇒ abstract_value list"
```

A field of the state is a field of its product, read and written through the wrapper.

```
definition st_get :: "(mcp_fields ⇒ 'c) ⇒ mcp_st ⇒ 'c" where
  "st_get f r = f (mcp_fields r)"

definition st_put :: "(mcp_fields ⇒ 'c ⇒ mcp_fields) ⇒ mcp_st ⇒ 'c ⇒ mcp_st" where
  "st_put u r v = MCP_St (u (mcp_fields r) v)"

lemmas st_lens_defs [simp] = st_get_def st_put_def

fun registration_of :: "analysis_domain ⇒ analysis_registration" where
  "registration_of Sign_Analysis =
     (| field_spec = (λ𝒢 p. lens_of (lift_get (st_get slot1)) (lift_put (st_put set_slot1))
       (ask_assign (exec_local_spec 𝒢 (default_st_is_bot_for (declared_global_vars p))
       (generic_tf_st_for sign_ops 𝒢) (generic_enter_st_for sign_ops 𝒢)))), field_live =
       (λr. st_get slot1 r ≠ ⊥), field_empty = (λgs r. (case st_get slot1 r of Bot ⇒ True
       | Lifted st ⇒ default_st_is_bot_for gs st)), value_answer = (λv q. (case slot1 v of
       Bot ⇒ ⊤ | Lifted st ⇒ sign_eval_answer st q)), value_display = (λv vars. Field_Store
       (map (λx. (x, SignValue (case slot1 v of Bot ⇒ ⊥ | Lifted st ⇒ st x))) vars)),
       context_values = (λctx. map SignValue (slot1 ctx)) |)"
| "registration_of Interval_Analysis =
     (| field_spec = (λ𝒢 p. lens_of (lift_get (st_get slot2)) (lift_put (st_put set_slot2))
       (ask_assign (exec_local_spec 𝒢 (default_st_is_bot_for (declared_global_vars p))
       (generic_tf_st_for ivl_ops 𝒢) (generic_enter_st_for ivl_ops 𝒢)))), field_live =
       (λr. st_get slot2 r ≠ ⊥), field_empty = (λgs r. (case st_get slot2 r of Bot ⇒ True
       | Lifted st ⇒ default_st_is_bot_for gs st)), value_answer = (λv q. (case slot2 v of
       Bot ⇒ ⊤ | Lifted st ⇒ interval_eval_answer st q)), value_display = (λv vars.
       Field_Store (map (λx. (x, IntervalValue (case slot2 v of Bot ⇒ ⊥ | Lifted st ⇒
       st x))) vars)), context_values = (λctx. map IntervalValue (slot2 ctx)) |)"
| "registration_of Parity_Analysis =
     (| field_spec = (λ𝒢 p. lens_of (lift_get (st_get slot3)) (lift_put (st_put set_slot3))
       (ask_assign (exec_local_spec 𝒢 (default_st_is_bot_for (declared_global_vars p))
       (generic_tf_st_for parity_ops 𝒢) (generic_enter_st_for parity_ops 𝒢)))), field_live
       = (λr. st_get slot3 r ≠ ⊥), field_empty = (λgs r. (case st_get slot3 r of Bot ⇒
       True | Lifted st ⇒ default_st_is_bot_for gs st)), value_answer = (λv q. (case
       slot3 v of Bot ⇒ ⊤ | Lifted st ⇒ parity_eval_answer st q)), value_display = (λv
       vars. Field_Store (map (λx. (x, ParityValue (case slot3 v of Bot ⇒ ⊥ | Lifted st
       ⇒ st x))) vars)), context_values = (λctx. map ParityValue (slot3 ctx)) |)"
| "registration_of (Int_Analysis Refine_Fixpoint) =
```

```
    ⦇ field_spec = (λG p. lens_of (lift_get (st_get slot4)) (lift_put (st_put set_slot4))
      (ask_assign (exec_local_spec G (default_st_is_bot_for (declared_global_vars p))
      (generic_tf_st_for (int_dom_ops Refine_Fixpoint) G) (generic_enter_st_for
      (int_dom_ops Refine_Fixpoint) G)))), field_live = (λr. st_get slot4 r ≠ ⊥),
      field_empty = (λgs r. (case st_get slot4 r of Bot ⇒ True | Lifted st ⇒
      default_st_is_bot_for gs st)), value_answer = (λv q. (case slot4 v of Bot ⇒ ⊤ |
      Lifted st ⇒ int_eval_answer Refine_Fixpoint st q)), value_display = (λv vars.
      Field_Store (map (λx. (x, IntDomValue (case slot4 v of Bot ⇒ ⊥ | Lifted st ⇒
      st x))) vars)), context_values = (λctx. map IntDomValue (slot4 ctx)) ⦈"
| "registration_of (Int_Analysis Refine_Once) =
    ⦇ field_spec = (λG p. lens_of (lift_get (st_get slot5)) (lift_put (st_put set_slot5))
      (ask_assign (exec_local_spec G (default_st_is_bot_for (declared_global_vars p))
      (generic_tf_st_for (int_dom_ops Refine_Once) G) (generic_enter_st_for (int_dom_ops
      Refine_Once) G)))), field_live = (λr. st_get slot5 r ≠ ⊥), field_empty = (λgs r.
      (case st_get slot5 r of Bot ⇒ True | Lifted st ⇒ default_st_is_bot_for gs st)),
      value_answer = (λv q. (case slot5 v of Bot ⇒ ⊤ | Lifted st ⇒ int_eval_answer
      Refine_Once st q)), value_display = (λv vars. Field_Store (map (λx. (x,
      IntDomOnceValue (case slot5 v of Bot ⇒ ⊥ | Lifted st ⇒ st x))) vars)),
      context_values = (λctx. map IntDomOnceValue (slot5 ctx)) ⦈"
| "registration_of (Int_Analysis Refine_Never) =
    ⦇ field_spec = (λG p. lens_of (lift_get (st_get slot6)) (lift_put (st_put set_slot6))
      (ask_assign (exec_local_spec G (default_st_is_bot_for (declared_global_vars p))
      (generic_tf_st_for (int_dom_ops Refine_Never) G) (generic_enter_st_for (int_dom_ops
      Refine_Never) G)))), field_live = (λr. st_get slot6 r ≠ ⊥), field_empty = (λgs r.
      (case st_get slot6 r of Bot ⇒ True | Lifted st ⇒ default_st_is_bot_for gs st)),
      value_answer = (λv q. (case slot6 v of Bot ⇒ ⊤ | Lifted st ⇒ int_eval_answer
      Refine_Never st q)), value_display = (λv vars. Field_Store (map (λx. (x,
      IntDomNeverValue (case slot6 v of Bot ⇒ ⊥ | Lifted st ⇒ st x))) vars)),
      context_values = (λctx. map IntDomNeverValue (slot6 ctx)) ⦈"
| "registration_of Congruence_Analysis =
    ⦇ field_spec = (λG p. lens_of (lift_get (st_get slot7)) (lift_put (st_put set_slot7))
      (ask_assign (exec_local_spec G (default_st_is_bot_for (declared_global_vars p))
      (generic_tf_st_for congruence_ops G) (generic_enter_st_for congruence_ops G)))),
      field_live = (λr. st_get slot7 r ≠ ⊥), field_empty = (λgs r. (case st_get slot7 r
      of Bot ⇒ True | Lifted st ⇒ default_st_is_bot_for gs st)), value_answer = (λv q.
      (case slot7 v of Bot ⇒ ⊤ | Lifted st ⇒ congruence_eval_answer st q)), value_display
      = (λv vars. Field_Store (map (λx. (x, CongruenceValue (case slot7 v of Bot ⇒ ⊥ |
      Lifted st ⇒ st x))) vars)), context_values = (λctx. map CongruenceValue (slot7 ctx)) ⦈"
| "registration_of Order_Analysis =
    ⦇ field_spec = (λG p. lens_of (lift_get (st_get slot8)) (lift_put (st_put set_slot8))
      (order_spec (program_vars p))), field_live = (λr. st_get slot8 r ≠ ⊥), field_empty =
      (λgs r. st_get slot8 r = RelBot), value_answer = (λv q. relc_qry (slot8 v) q),
      value_display = (λv vars. Field_Whole (OrderValue (order_pairs (slot8 v)))),
      context_values = (λctx. []) ⦈"

fun part_gamma :: "(vname ⇒ bool) ⇒ analysis_domain ⇒ mcp_st lifted ⇒ store set" where
  "part_gamma G Sign_Analysis =
     (λx. gamma_lift (default_st_gamma G) (lift_get (st_get slot1) x))"
| "part_gamma G Interval_Analysis =
     (λx. gamma_lift (default_st_gamma G) (lift_get (st_get slot2) x))"
| "part_gamma G Parity_Analysis =
     (λx. gamma_lift (default_st_gamma G) (lift_get (st_get slot3) x))"
| "part_gamma G (Int_Analysis Refine_Fixpoint) =
     (λx. gamma_lift (default_st_gamma G) (lift_get (st_get slot4) x))"
| "part_gamma G (Int_Analysis Refine_Once) =
     (λx. gamma_lift (default_st_gamma G) (lift_get (st_get slot5) x))"
| "part_gamma G (Int_Analysis Refine_Never) =
     (λx. gamma_lift (default_st_gamma G) (lift_get (st_get slot6) x))"
| "part_gamma G Congruence_Analysis =
     (λx. gamma_lift (default_st_gamma G) (lift_get (st_get slot7) x))"
| "part_gamma G Order_Analysis =
     (λx. γ (lift_get (st_get slot8) x))"

fun val_empty :: "analysis_domain ⇒ mcp_val ⇒ bool" where
  "val_empty Sign_Analysis v =
     (case slot1 v of Bot ⇒ True | Lifted st ⇒ is_empty_state st)"
| "val_empty Interval_Analysis v =
     (case slot2 v of Bot ⇒ True | Lifted st ⇒ is_empty_state st)"
| "val_empty Parity_Analysis v =
     (case slot3 v of Bot ⇒ True | Lifted st ⇒ is_empty_state st)"
| "val_empty (Int_Analysis Refine_Fixpoint) v =
     (case slot4 v of Bot ⇒ True | Lifted st ⇒ is_empty_state st)"
| "val_empty (Int_Analysis Refine_Once) v =
     (case slot5 v of Bot ⇒ True | Lifted st ⇒ is_empty_state st)"
| "val_empty (Int_Analysis Refine_Never) v =
     (case slot6 v of Bot ⇒ True | Lifted st ⇒ is_empty_state st)"
```

```
| "val_empty Congruence_Analysis v =
     (case slot7 v of Bot ⇒ True | Lifted st ⇒ is_empty_state st)"
| "val_empty Order_Analysis v =
     ((slot8 v) = RelBot)"

fun val_gamma :: "analysis_domain ⇒ mcp_val ⇒ store set" where
  "val_gamma Sign_Analysis v =
     γ⊥ (slot1 v)"
| "val_gamma Interval_Analysis v =
     γ⊥ (slot2 v)"
| "val_gamma Parity_Analysis v =
     γ⊥ (slot3 v)"
| "val_gamma (Int_Analysis Refine_Fixpoint) v =
     γ⊥ (slot4 v)"
| "val_gamma (Int_Analysis Refine_Once) v =
     γ⊥ (slot5 v)"
| "val_gamma (Int_Analysis Refine_Never) v =
     γ⊥ (slot6 v)"
| "val_gamma Congruence_Analysis v =
     γ⊥ (slot7 v)"
| "val_gamma Order_Analysis v =
     γ (slot8 v)"
```

What each field publishes

A published value reads each non-relational field's stored state as a total function on variables and keeps the order field as it is.

```
definition mcp_rd :: "(vname ⇒ bool) ⇒ mcp_st ⇒ mcp_val" where
  "mcp_rd 𝒢 r = Product (map_lift (default_st_to_fun 𝒢) (st_get slot1 r)) (Product
    (map_lift (default_st_to_fun 𝒢) (st_get slot2 r)) (Product (map_lift (default_st_to_fun
    𝒢) (st_get slot3 r)) (Product (map_lift (default_st_to_fun 𝒢) (st_get slot4 r))
    (Product (map_lift (default_st_to_fun 𝒢) (st_get slot5 r)) (Product (map_lift
    (default_st_to_fun 𝒢) (st_get slot6 r)) (Product (map_lift (default_st_to_fun 𝒢)
    (st_get slot7 r)) (st_get slot8 r)))))))"

definition mcp_gamma_v :: "analysis_domain list ⇒ mcp_val ⇒ store set" where
  "mcp_gamma_v as v = (⋂a ∈ set as. val_gamma a v)"
```

Where each field starts, and what it keys a callee by

The entry state starts every active field at its analysis's own entry state and every other field at its bottom. A field no active analysis runs is never read, and keeping it at bottom makes the solver's joins, widenings and comparisons on it constant-time. Under the entry-state policy a callee is keyed by what the active fields key it by; a field no active analysis runs keys nothing.

```
definition mcp_init :: "analysis_domain list ⇒ mcp_st" where
  "mcp_init as = MCP_St (Product (if Sign_Analysis ∈ set as then Lifted cinit_sign_st
    else ⊥) (Product (if Interval_Analysis ∈ set as then Lifted cinit_ivl_st else ⊥)
    (Product (if Parity_Analysis ∈ set as then Lifted cinit_parity_st else ⊥) (Product
    (if (Int_Analysis Refine_Fixpoint) ∈ set as then Lifted cinit_int_dom_st else ⊥)
    (Product (if (Int_Analysis Refine_Once) ∈ set as then Lifted cinit_int_dom_st else
    ⊥) (Product (if (Int_Analysis Refine_Never) ∈ set as then Lifted cinit_int_dom_st
    else ⊥) (Product (if Congruence_Analysis ∈ set as then Lifted cinit_congruence_st
    else ⊥) (if Order_Analysis ∈ set as then (⊤ :: relc) else ⊥)))))))"

definition mcp_formals_route ::
  "analysis_domain list ⇒ (vname ⇒ bool) ⇒ pp ⇒ mcp_ctx ⇒ mcp_st lifted
     ⇒ call_action ⇒ mcp_ctx" where
  "mcp_formals_route as 𝒢 u ctx d ca = Product (if Sign_Analysis ∈ set as then
    exec_formals_route 𝒢 u [] (lift_get (st_get slot1) d) ca else []) (Product (if
    Interval_Analysis ∈ set as then exec_formals_route 𝒢 u [] (lift_get (st_get slot2) d) ca
    else []) (Product (if Parity_Analysis ∈ set as then exec_formals_route 𝒢 u [] (lift_get
    (st_get slot3) d) ca else []) (Product (if (Int_Analysis Refine_Fixpoint) ∈ set as then
    exec_formals_route 𝒢 u [] (lift_get (st_get slot4) d) ca else []) (Product (if
    (Int_Analysis Refine_Once) ∈ set as then exec_formals_route 𝒢 u [] (lift_get (st_get
    slot5) d) ca else []) (Product (if (Int_Analysis Refine_Never) ∈ set as then
    exec_formals_route 𝒢 u [] (lift_get (st_get slot6) d) ca else []) (Product (if
    Congruence_Analysis ∈ set as then exec_formals_route 𝒢 u [] (lift_get (st_get slot7) d)
    ca else []) (if Order_Analysis ∈ set as then [] else [])))))))"

definition mcp_root_ctx :: mcp_ctx where
  "mcp_root_ctx = Product [] (Product [] (Product [] (Product [] (Product [] (Product []
    (Product [] [])))))))"
```

A key for a whole context, injective on every context: each field's values, field by field. Listing contexts by it drops none.

```
definition mcp_ctx_key :: "mcp_ctx ⇒ order_key" where
  "mcp_ctx_key ctx = Key_List [Key_List (map abstract_value_key (map SignValue (slot1
    ctx))), Key_List (map abstract_value_key (map IntervalValue (slot2 ctx))), Key_List
    (map abstract_value_key (map ParityValue (slot3 ctx))), Key_List (map
    abstract_value_key (map IntDomValue (slot4 ctx))), Key_List (map abstract_value_key
    (map IntDomOnceValue (slot5 ctx))), Key_List (map abstract_value_key (map
    IntDomNeverValue (slot6 ctx))), Key_List (map abstract_value_key (map CongruenceValue
    (slot7 ctx))), Key_List (map (λ_. Key_List []) (slot8 ctx))]"

lemma mcp_ctx_key_inject: "mcp_ctx_key a = mcp_ctx_key b ⟹ a = b"
  unfolding mcp_ctx_key_def slot1_def slot2_def slot3_def slot4_def slot5_def slot6_def
    slot7_def slot8_def
  by (simp add: inj_def; elim conjE; (rule analysis_product.expand; simp)+)
```

What each analysis's registration supplies

Each fact below splits on the analysis and cites the matching registration.

```
lemma field_spec_sound:
  "sound_local_spec (declared_global p) (part_gamma (declared_global p) a)
     (field_spec (registration_of a) (declared_global p) p)"
  by (cases a rule: analysis_domain_cases;
      simp only: part_gamma.simps registration_of.simps analysis_registration.simps;
      rule field_component_sound[OF ask_assign_sound[OF sign_rule.exec_comp_sound]]
        field_component_sound[OF ask_assign_sound[OF interval_rule.exec_comp_sound]]
        field_component_sound[OF ask_assign_sound[OF parity_rule.exec_comp_sound]]
        field_component_sound[OF ask_assign_sound[OF int_fixpoint_rule.exec_comp_sound]]
        field_component_sound[OF ask_assign_sound[OF int_once_rule.exec_comp_sound]]
        field_component_sound[OF ask_assign_sound[OF int_never_rule.exec_comp_sound]]
        field_component_sound[OF ask_assign_sound[OF congruence_rule.exec_comp_sound]]
        field_component_sound[OF order_spec_sound];
      auto simp: less_eq_mcp_st_def less_eq_analysis_product_def)

lemma single_entry_field_spec: "single_entry (field_spec (registration_of a) 𝒢 p)"
  by (cases a rule: analysis_domain_cases) (auto intro!: single_entry_lens_of lift_put_get
    single_entry_ask_assign[OF single_entry_exec_local_spec] single_entry_order_spec)

lemma mcp_init_sound:
  "cinit_stores (declared_global p)
     ⊆ mcp_gamma_v as (mcp_rd (declared_global p) (mcp_init as))"
proof -
  have "cinit_stores (declared_global p)
          ⊆ val_gamma a (mcp_rd (declared_global p) (mcp_init as))"
    if "a ∈ set as" for a
    by (cases a rule: analysis_domain_cases)
       (use that sign_rule.init_sound interval_rule.init_sound parity_rule.init_sound
         int_fixpoint_rule.init_sound int_once_rule.init_sound int_never_rule.init_sound
         congruence_rule.init_sound in ‹simp_all add: mcp_init_def mcp_rd_def›)
  then show ?thesis by (auto simp: mcp_gamma_v_def)
qed

lemma value_answer_sound:
  "s ∈ val_gamma a v
     ⟹ eval_holds q (value_answer (registration_of a) v q) s"
  by (cases a rule: analysis_domain_cases)
     (auto split: lifted.splits intro: sign_tf.check.eval_answer_sound
       ivl_tf.check.eval_answer_sound parity_tf.check.eval_answer_sound
       int_tf.check.eval_answer_sound congruence_tf.check.eval_answer_sound
       relc_qry_sound)

end
```

17.5 MCP_Analyses

```
theory MCP_Analyses
  imports MCP_Carrier
begin
```

The registered analyses, run as one

`Voblint_CLI.MCP_Carrier` lays out one field per registered analysis and states how each field runs, describes, reads back and answers. This theory runs the active fields as one component, proves the combination sound whatever analyses are registered, and registers it at every context policy. Every proof here goes by cases over the registered analyses, so a new analysis in the manifest needs nothing new here.

Each analysis on its own field

```
lemma field_spec_frame:
  "a ≠ b ⟹ mcp_frame (field_spec (registration_of a) 𝒢 p) (part_gamma 𝒢 b)"
  by (cases a rule: analysis_domain_cases; cases b rule: analysis_domain_cases;
      simp only: part_gamma.simps registration_of.simps analysis_registration.simps;
      rule field_frame; simp add: lift_get_put_other)

lemma part_gamma_rd: "part_gamma 𝒢 a x = gamma_lift (val_gamma a) (map_lift (mcp_rd 𝒢) x)"
  by (cases a rule: analysis_domain_cases; cases x)
    (simp_all add: mcp_rd_def gamma_lift_default_st_gamma_to_fun)
```

Each analysis answers what it knows

While the system is solved, a field answers a query as its published value does: from its own publication map, through the answer its analysis registers for checks. An analysis written without a query handler thereby becomes a provider on the combined state, as every Goblint analysis answers `EvalInt` from its own domain. The answer is sound for the stores the field describes, so the handler replaces the component's own (`with_qry`) without touching a transfer.

```
definition part_answer :: "(vname ⇒ bool) ⇒ analysis_domain ⇒ mcp_st lifted ⇒ channel"
where
  "part_answer 𝒢 a x q =
     (case x of Bot ⇒ ⊤ | Lifted r ⇒ value_answer (registration_of a) (mcp_rd 𝒢 r) q)"

lemma part_answer_sound: "s ∈ part_gamma 𝒢 a x ⟹ eval_holds q (part_answer 𝒢 a x q) s"
  by (cases x) (auto simp: part_answer_def part_gamma_rd intro: value_answer_sound)

definition mcp_field ::
  "(vname ⇒ bool) ⇒ imp_prog ⇒ analysis_domain ⇒ mcp_st lifted local_spec" where
  "mcp_field 𝒢 p a = with_qry (λch. part_answer 𝒢 a) (field_spec (registration_of a) 𝒢 p)"

lemma mcp_field_sound:
  "sound_local_spec (declared_global p) (part_gamma (declared_global p) a)
     (mcp_field (declared_global p) p a)"
  unfolding mcp_field_def
  by (rule with_qry_sound[OF field_spec_sound]) (rule part_answer_sound)

lemma mcp_field_frame: "a ≠ b ⟹ mcp_frame (mcp_field 𝒢 p a) (part_gamma 𝒢 b)"
  unfolding mcp_field_def by (rule with_qry_frame[OF field_spec_frame])
```

The active analyses, run as one

The combined state is unreachable as soon as one active analysis says its field is, as Goblint's `MCP` raises `Deadcode` when one component does. The normalization changes no concretization: a field that describes nothing empties the intersection already.

```
definition mcp_norm :: "analysis_domain list ⇒ mcp_st lifted ⇒ mcp_st lifted" where
  "mcp_norm as x =
     (case x of
        Bot ⇒ Bot
      | Lifted r ⇒ if list_all (λa. field_live (registration_of a) r) as then x else Bot)"

lemma part_gamma_Bot [simp]: "part_gamma 𝒢 a Bot = {}"
  by (cases a rule: analysis_domain_cases) simp_all

lemma part_gamma_dead:
  "¬ field_live (registration_of a) r ⟹ part_gamma 𝒢 a (Lifted r) = {}"
  by (cases a rule: analysis_domain_cases) simp_all
```

```
lemma mcp_gamma_norm:
  "mcp_gamma (map (part_gamma 𝒢) as) (mcp_norm as x) = mcp_gamma (map (part_gamma 𝒢) as) x"
  by (cases x) (auto simp: mcp_norm_def mcp_gamma_def list_all_iff dest: part_gamma_dead)

definition mcp_comp ::
  "analysis_domain list ⇒ (vname ⇒ bool) ⇒ imp_prog ⇒ mcp_st lifted local_spec" where
  "mcp_comp as 𝒢 p = map_local_spec (mcp_norm as) (mcp_combine (map (mcp_field 𝒢 p) as))"

theorem mcp_comp_sound:
  assumes "distinct as" and "as ≠ []"
  shows "sound_local_spec (declared_global p)
           (mcp_gamma (map (part_gamma (declared_global p)) as))
           (mcp_comp as (declared_global p) p)"
proof -
  let ?gcs = "map (λa. (part_gamma (declared_global p) a,
                        mcp_field (declared_global p) p a)) as"
  have "sound_local_spec (declared_global p) (mcp_gamma (map fst ?gcs))
          (mcp_combine (map snd ?gcs))"
    by (rule mcp_combine_sound)
       (auto simp: mcp_field_sound assms
         intro!: mcp_independent_map mcp_field_frame)
  then show ?thesis
    unfolding mcp_comp_def
    by (intro map_local_spec_sound) (simp_all add: comp_def mcp_gamma_norm)
qed

lemma single_entry_mcp_comp: "as ≠ [] ⟹ single_entry (mcp_comp as 𝒢 p)"
  unfolding mcp_comp_def mcp_field_def
  by (intro single_entry_map_local_spec single_entry_mcp_combine)
     (auto simp: single_entry_field_spec single_entry_with_qry)
```

What the combined state publishes

A solved combined state is published field by field, each through its analysis's own publication map. Its concretization is the intersection over the active fields, and a check is decided from the meet of the active analyses' answers.

```
lemma mcp_gamma_rd:
  "as ≠ [] ⟹ mcp_gamma (map (part_gamma 𝒢) as) x
                = gamma_lift (mcp_gamma_v as) (map_lift (mcp_rd 𝒢) x)"
  by (cases x) (auto simp: mcp_gamma_def mcp_gamma_v_def part_gamma_rd)

lemma mcp_gamma_v_bot: "as ≠ [] ⟹ mcp_gamma_v as ⊥ = {}"
proof -
  have "val_gamma a ⊥ = {}" for a by (cases a rule: analysis_domain_cases) simp_all
  then show "as ≠ [] ⟹ ?thesis" by (auto simp: mcp_gamma_v_def)
qed

definition mcp_emp :: "analysis_domain list ⇒ imp_prog ⇒ mcp_st ⇒ bool" where
  "mcp_emp as p r =
     list_ex (λa. field_empty (registration_of a) (declared_global_vars p) r) as"

definition mcp_empty_v :: "analysis_domain list ⇒ mcp_val ⇒ bool" where
  "mcp_empty_v as v = list_ex (λa. val_empty a v) as"

lemma field_empty_rd:
  "field_empty (registration_of a) (declared_global_vars p) r
     = val_empty a (mcp_rd (declared_global p) r)"
  by (cases a rule: analysis_domain_cases)
     (simp_all add: mcp_rd_def
        default_st_is_bot_for_iff[where 𝒢 = "declared_global p", OF declared_global_iff]
        split: lifted.split)

lemma mcp_emp_rd: "mcp_emp as p r = mcp_empty_v as (mcp_rd (declared_global p) r)"
  by (simp add: mcp_emp_def mcp_empty_v_def field_empty_rd)

lemma val_empty_gamma: "val_empty a v ⟹ val_gamma a v = {}"
  by (cases a rule: analysis_domain_cases)
    (auto split: lifted.splits dest: is_empty_state_gamma_state_empty)

lemma mcp_empty_v_gamma: "mcp_empty_v as v ⟹ mcp_gamma_v as v = {}"
  by (auto simp: mcp_empty_v_def mcp_gamma_v_def list_ex_iff dest: val_empty_gamma)

lemma sound_emptiness_mcp: "sound_emptiness (mcp_empty_v as) (mcp_gamma_v as)"
  by (rule sound_emptinessI) (rule mcp_empty_v_gamma)
```

The combined test is not exact. Interval says `x = 2` and Parity says `x` is odd: neither field is empty, so the test does not fire, yet no store satisfies both.

```
definition mcp_contradiction :: mcp_val where
  "mcp_contradiction =
     set_slot3 (set_slot2 (⊥ :: mcp_val) (Lifted ((⊤ :: ivl abs_state)(STR ''x'' := Ivl (Fin 2) (Fin 2)))))
       (Lifted ((⊤ :: parity abs_state)(STR ''x'' := POdd)))"

lemma mcp_contradiction_not_empty:
  "¬ mcp_empty_v [Interval_Analysis, Parity_Analysis] mcp_contradiction"
proof -
  have "(λ_. 2) ∈ γ ((⊤ :: ivl abs_state)(STR ''x'' := Ivl (Fin 2) (Fin 2)))"
    by (auto simp: gamma_state_def top_ivl_def gamma_ivl_top)
  moreover have "(λ_. 1) ∈ γ ((⊤ :: parity abs_state)(STR ''x'' := POdd))"
    by (auto simp: gamma_state_def gamma_parity_top)
  ultimately show ?thesis
    by (auto simp: mcp_empty_v_def mcp_contradiction_def is_empty_state_iff_gamma_state_empty)
qed

lemma mcp_contradiction_no_store:
  "mcp_gamma_v [Interval_Analysis, Parity_Analysis] mcp_contradiction = {}"
proof (rule equals0I)
  fix s
  assume "s ∈ mcp_gamma_v [Interval_Analysis, Parity_Analysis] mcp_contradiction"
  then have "s (STR ''x'') ∈ gamma_ivl (Ivl (Fin 2) (Fin 2))"
    and "s (STR ''x'') ∈ gamma_parity POdd"
    by (auto simp: mcp_gamma_v_def mcp_contradiction_def gamma_state_def
             dest: spec[of _ "STR ''x''"])
  then show False by (simp, presburger)
qed

lemma mcp_empty_v_not_exact:
  "¬ exact_emptiness (mcp_empty_v [Interval_Analysis, Parity_Analysis])
       (mcp_gamma_v [Interval_Analysis, Parity_Analysis])"
  using exact_emptinessD [of _ _ mcp_contradiction] mcp_contradiction_not_empty
    mcp_contradiction_no_store
  by blast

definition mcp_answer :: "analysis_domain list ⇒ mcp_val ⇒ query ⇒ answer" where
  "mcp_answer as v q = fold (λa r. r ⊓ value_answer (registration_of a) v q) as ⊤"

lemma mcp_answer_fold_sound:
  "∀a ∈ set as. s ∈ val_gamma a v ⟹ eval_holds q r s
   ⟹ eval_holds q (fold (λa r. r ⊓ value_answer (registration_of a) v q) as r) s"
  by (induction as arbitrary: r) (auto intro: eval_query.inf_sound value_answer_sound)

lemma mcp_answer_sound: "s ∈ mcp_gamma_v as v ⟹ eval_holds q (mcp_answer as v q) s"
  unfolding mcp_answer_def mcp_gamma_v_def by (rule mcp_answer_fold_sound) auto

definition mcp_classify :: "analysis_domain list ⇒ exp ⇒ mcp_val ⇒ check_result" where
  "mcp_classify as = answer_check (mcp_answer as)"

lemma mcp_classify_proved:
  "mcp_classify as e v = Check_Proved ⟹ s ∈ mcp_gamma_v as v ⟹ truthy (⟦e⟧e s)"
  unfolding mcp_classify_def by (erule answer_check_proved) (rule mcp_answer_sound)

lemma mcp_classify_refuted:
  "mcp_classify as e v = Check_Refuted ⟹ s ∈ mcp_gamma_v as v
   ⟹ ¬ truthy (⟦e⟧e s)"
  unfolding mcp_classify_def by (erule answer_check_refuted) (rule mcp_answer_sound)
```

A caller activates a distinct, nonempty list of analyses. The registrations below take any list and run it as `activation` normalizes it, so they hold unconditionally; `activation` is the identity on every list a caller may pass.

```
definition activation :: "analysis_domain list ⇒ analysis_domain list" where
  "activation as = (if as = [] then [Int_Analysis Refine_Fixpoint] else remdups as)"

lemma activation_ne [simp]: "activation as ≠ []"
  by (simp add: activation_def)

lemma distinct_activation [simp]: "distinct (activation as)"
  by (simp add: activation_def)

lemma activation_id: "distinct as ⟹ as ≠ [] ⟹ activation as = as"
  by (simp add: activation_def)
```

Registering the active analyses

The active analyses run through the shared pipeline once per context policy, as one registered analysis does. What they owe the pipeline does not depend on the policy, beyond its seeds being apart from the analysis-wide global, so it is proved once here. Each registration leaves the activation list and the global update rule as parameters.

```
lemma mcp_routed_dg_analysis:
  fixes buffer_key :: 'k and global_of :: "'n ⇒ 'k"
  assumes seed_ne: "⋀v ctx. seed v ctx ≠ buffer_key"
    and seed_ne_global: "⋀v ctx n. seed v ctx ≠ global_of n"
  shows "dg_analysis (mcp_comp (activation as)) (mcp_emp (activation as)) mcp_rd
    (mcp_init (activation as)) buffer_key global_of seed
    (TD_side_rule_Interp_solve r)
    (TD_side_rule_Interp.solve_dom TYPE('k) TYPE((mcp_st lifted, mcp_st lifted) dg_state) r)
    ⊥ (mcp_classify (activation as)) (mcp_gamma_v (activation as))
    (mcp_empty_v (activation as)) (TD_side_rule_Interp_solve_c r)
    (λ𝒢 c. dg_spec_of c) (λ𝒢 d e. d) (λ𝒢 d. d) (λ𝒢 d. Bot)
    (λ𝒢 p. []) (λp ci d e. d)"
proof (rule dg_analysis_whole_stateI[OF td_certified_solver],
    goal_cases CompSound EnterSingle EmptyRd EmptyVSound SeedNe ClProved ClRefuted BotState
    Init SeedNeGlobal)
  case (CompSound p)
  have eq: "mcp_gamma (map (part_gamma (declared_global p)) (activation as))
      = (λd. gamma_lift (mcp_gamma_v (activation as)) (map_lift (mcp_rd (declared_global p)) d))"
    by (rule ext) (rule mcp_gamma_rd[OF activation_ne])
  show ?case using mcp_comp_sound[of "activation as" p] unfolding eq by simp
next
  case (EnterSingle p ci d)
  show ?case
    using single_entryD[OF single_entry_mcp_comp[OF activation_ne],
        of as "declared_global p" p
          "ls_channel (mcp_comp (activation as) (declared_global p) p) d" ci "(d, d)"]
    by (simp add: dg_pipeline.comp_entry_def)
next
  case (EmptyRd p s) show ?case by (rule mcp_emp_rd)
next
  case EmptyVSound show ?case by (rule sound_emptiness_mcp)
next
  case (SeedNe v ctx) show ?case by (rule seed_ne)
next
  case (ClProved e d s) then show ?case by (rule mcp_classify_proved)
next
  case (ClRefuted e d s) then show ?case by (rule mcp_classify_refuted)
next
  case BotState show ?case by (rule mcp_gamma_v_bot[OF activation_ne])
next
  case (Init p) show ?case by (rule mcp_init_sound)
next
  case (SeedNeGlobal v ctx n) show ?case by (rule seed_ne_global)
qed
```

At the unit context

`mcp_rule`: the analysis with one context `()`, routed by `route_unit`. Each registration names a solver key per program global, which this placement never publishes at: every global lives in a point's own state.

```
global_interpretation mcp_rule: dg_analysis
    "mcp_comp (activation as)" "mcp_emp (activation as)" mcp_rd "mcp_init (activation as)"
    Analysis_Buffer Analysis_Global Activation_Seed "λ_. route_unit" "()"
    "TD_side_rule_Interp_solve r"
    "TD_side_rule_Interp.solve_dom TYPE((vname, unit) global_unknown)
       TYPE((mcp_st lifted, mcp_st lifted) dg_state) r"
    ⊥ "mcp_classify (activation as)" "mcp_gamma_v (activation as)"
    "mcp_empty_v (activation as)" "TD_side_rule_Interp_solve_c r"
    "λ𝒢 c. dg_spec_of c" "λ𝒢 d e. d" "λ𝒢 d. d" "λ𝒢 d. Bot"
    "λ𝒢 p. []" "λp ci d e. d"
  for as r
  by (rule mcp_routed_dg_analysis) simp_all
```

At the entry-state context

An activation is keyed by the values the active analyses give its formals on entry. The context has a list per field; a field no active analysis runs keys nothing.

```
global_interpretation mcp_es_rule: dg_analysis
```

```
    "mcp_comp (activation as)" "mcp_emp (activation as)" mcp_rd "mcp_init (activation as)"
    Analysis_Buffer Analysis_Global Activation_Seed "mcp_formals_route (activation as)"
      mcp_root_ctx
    "TD_side_rule_Interp_solve r"
    "TD_side_rule_Interp.solve_dom TYPE((vname, mcp_ctx) global_unknown)
       TYPE((mcp_st lifted, mcp_st lifted) dg_state) r"
    ⊥ "mcp_classify (activation as)" "mcp_gamma_v (activation as)"
    "mcp_empty_v (activation as)" "TD_side_rule_Interp_solve_c r"
    "λ𝒢 c. dg_spec_of c" "λ𝒢 d e. d" "λ𝒢 d. d" "λ𝒢 d. Bot"
    "λ𝒢 p. []" "λp ci d e. d"
  for as r
  by (rule mcp_routed_dg_analysis) simp_all
```

At the call-string context

`mcp_cs_rule`: an activation is keyed by its last `k` call sites, routed by `cs_route k` from the empty root string.

```
global_interpretation mcp_cs_rule: dg_analysis
    "mcp_comp (activation as)" "mcp_emp (activation as)" mcp_rd "mcp_init (activation as)"
    Analysis_Buffer Analysis_Global Activation_Seed "λ_. cs_route k" "[]"
    "TD_side_rule_Interp_solve r"
    "TD_side_rule_Interp.solve_dom TYPE((vname, cfg_node list) global_unknown)
       TYPE((mcp_st lifted, mcp_st lifted) dg_state) r"
    ⊥ "mcp_classify (activation as)" "mcp_gamma_v (activation as)"
    "mcp_empty_v (activation as)" "TD_side_rule_Interp_solve_c r"
    "λ𝒢 c. dg_spec_of c" "λ𝒢 d e. d" "λ𝒢 d. d" "λ𝒢 d. Bot"
    "λ𝒢 p. []" "λp ci d e. d"
  for as k r
  by (rule mcp_routed_dg_analysis) simp_all

end
```

17.6 MCP_Split

```
theory MCP_Split
  imports MCP_Analyses
begin
```

The registered analyses with program globals on the shared channel

The ownership-split placement recombines local and global information and projects a state onto its local part, its global part, or one named global. On the combined state they are defined field by field, so a carrier that splits is a type class closed under the constructors the combined state is built from. A pointwise field splits by the location each name is stored at; the relational field relates variables across the split and stays wholly local.

```
class ownership_split = order_bot +
  fixes split_cmb :: "'a ⇒ 'a ⇒ 'a"
    and split_rl :: "'a ⇒ 'a"
    and split_global_at :: "vname ⇒ 'a ⇒ 'a"
    and split_rg :: "'a ⇒ 'a"
    and split_free :: "vname list ⇒ 'a"
  assumes split_cmb_mono: "d ≤ d' ⟹ g ≤ g' ⟹ split_cmb d g ≤ split_cmb d' g'"
    and split_recombine: "split_cmb (split_rl x) (split_rg x) = x"
    and split_cmb_rl: "split_cmb (split_rl x) g = split_cmb x g"
    and split_rl_cmb: "split_rl (split_cmb d g) = split_rl d"
    and split_global_at_mono: "d ≤ d' ⟹ split_global_at n d ≤ split_global_at n d'"
    and split_free_ge: "n ∉ set R ⟹ split_global_at n d ≤ split_free R"
    and split_global_at_bot: "split_global_at n bot = bot"
```

`split_free R` is the global half a transfer reads in place of every global outside `R`: it claims nothing about those names and leaves the names in `R` to the values actually read.

```
instantiation default_st :: ("{order_bot, order_top}") ownership_split
begin

definition split_cmb_default_st :: "'a default_st ⇒ 'a default_st ⇒ 'a default_st" where
  "split_cmb_default_st = combine_default_st"

definition split_rl_default_st :: "'a default_st ⇒ 'a default_st" where
  "split_rl_default_st = restrict_local_default_st"

definition split_rg_default_st :: "'a default_st ⇒ 'a default_st" where
  "split_rg_default_st = restrict_global_default_st"

definition split_global_at_default_st :: "vname ⇒ 'a default_st ⇒ 'a default_st" where
  "split_global_at_default_st x d =
     (bot⟨(Global_Location x) := (d⟨(Global_Location x)⟩)⟩)"

definition split_free_default_st :: "vname list ⇒ 'a default_st" where
  "split_free_default_st R = ⦇(bot, []), (top, map (λx. (x, bot)) R)⦈"

lemma split_free_default_st_get:
  "(split_free R :: 'a default_st)⟨loc⟩ =
     (case loc of Global_Location x ⇒ if x ∈ set R then bot else top | _ ⇒ bot)"
  by (cases loc) (simp_all add: split_free_default_st_def map_of_map_restrict
      restrict_map_def)

instance
proof
  fix d d' g g' :: "'a default_st"
  assume "d ≤ d'" "g ≤ g'"
  then show "split_cmb d g ≤ split_cmb d' g'"
    by (auto simp: split_cmb_default_st_def le_default_st_iff split: location.split)
next
  fix n :: vname and R and d :: "'a default_st"
  assume "n ∉ set R"
  then show "split_global_at n d ≤ split_free R"
    by (auto simp: le_default_st_iff split_global_at_default_st_def split_free_default_st_get
        split: location.split)
next
  fix n :: vname
  show "split_global_at n (bot :: 'a default_st) = bot"
    by (rule default_st_eqI) (simp add: split_global_at_default_st_def)
qed (auto simp: split_cmb_default_st_def split_rl_default_st_def split_rg_default_st_def
    split_global_at_default_st_def le_default_st_iff)

end
```

```
instantiation relc :: ownership_split
begin

definition split_cmb_relc :: "relc ⇒ relc ⇒ relc" where "split_cmb_relc d g = d"
definition split_rl_relc :: "relc ⇒ relc" where "split_rl_relc d = d"
definition split_rg_relc :: "relc ⇒ relc" where "split_rg_relc d = bot"

definition split_global_at_relc :: "vname ⇒ relc ⇒ relc" where
  "split_global_at_relc x d = bot"

definition split_free_relc :: "vname list ⇒ relc" where "split_free_relc R = bot"

instance
  by standard (simp_all add: split_cmb_relc_def split_rl_relc_def split_rg_relc_def
      split_global_at_relc_def split_free_relc_def)

end

instantiation analysis_product :: (ownership_split, ownership_split) ownership_split
begin

definition split_cmb_analysis_product ::
  "('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product" where
  "split_cmb_analysis_product d g =
     Product (split_cmb (pleft d) (pleft g)) (split_cmb (pright d) (pright g))"

definition split_rl_analysis_product ::
  "('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product" where
  "split_rl_analysis_product d = Product (split_rl (pleft d)) (split_rl (pright d))"

definition split_rg_analysis_product ::
  "('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product" where
  "split_rg_analysis_product d = Product (split_rg (pleft d)) (split_rg (pright d))"

definition split_global_at_analysis_product ::
  "vname ⇒ ('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product" where
  "split_global_at_analysis_product x d =
     Product (split_global_at x (pleft d)) (split_global_at x (pright d))"

definition split_free_analysis_product :: "vname list ⇒ ('a, 'b) analysis_product" where
  "split_free_analysis_product R = Product (split_free R) (split_free R)"

instance
proof
  fix d d' g g' :: "('a, 'b) analysis_product"
  assume "d ≤ d'" "g ≤ g'"
  then show "split_cmb d g ≤ split_cmb d' g'"
    by (simp add: split_cmb_analysis_product_def less_eq_analysis_product_def split_cmb_mono)
next
  fix x g :: "('a, 'b) analysis_product"
  show "split_cmb (split_rl x) (split_rg x) = x"
    by (cases x) (simp add: split_cmb_analysis_product_def split_rl_analysis_product_def
        split_rg_analysis_product_def split_recombine)
  show "split_cmb (split_rl x) g = split_cmb x g"
    by (simp add: split_cmb_analysis_product_def split_rl_analysis_product_def split_cmb_rl)
next
  fix d g :: "('a, 'b) analysis_product"
  show "split_rl (split_cmb d g) = split_rl d"
    by (simp add: split_cmb_analysis_product_def split_rl_analysis_product_def split_rl_cmb)
next
  fix d d' :: "('a, 'b) analysis_product" and x
  assume "d ≤ d'"
  then show "split_global_at x d ≤ split_global_at x d'"
    by (simp add: split_global_at_analysis_product_def less_eq_analysis_product_def
      split_global_at_mono)
next
  fix n :: vname and R and d :: "('a, 'b) analysis_product"
  assume "n ∉ set R"
  then show "split_global_at n d ≤ split_free R"
    by (simp add: split_global_at_analysis_product_def split_free_analysis_product_def
        less_eq_analysis_product_def split_free_ge)
next
  fix n :: vname
  show "split_global_at n (bot :: ('a, 'b) analysis_product) = bot"
    by (simp add: split_global_at_analysis_product_def bot_analysis_product_def
        split_global_at_bot)
```

```
qed

end
```

An unreachable point stays unreachable whatever global it is recombined with, and a reachable point recombined with a global nothing has published yet reads the global half at its bottom.

```
instantiation lifted :: ("{ownership_split, semilattice_sup}") ownership_split
begin

definition split_cmb_lifted :: "'a lifted ⇒ 'a lifted ⇒ 'a lifted" where
  "split_cmb_lifted d g = (case d of Bot ⇒ Bot
     | Lifted a ⇒ Lifted (split_cmb a (case g of Bot ⇒ bot | Lifted b ⇒ b)))"

definition split_rl_lifted :: "'a lifted ⇒ 'a lifted" where
  "split_rl_lifted = map_lift split_rl"

definition split_rg_lifted :: "'a lifted ⇒ 'a lifted" where
  "split_rg_lifted = map_lift split_rg"

definition split_global_at_lifted :: "vname ⇒ 'a lifted ⇒ 'a lifted" where
  "split_global_at_lifted x = map_lift (split_global_at x)"

definition split_free_lifted :: "vname list ⇒ 'a lifted" where
  "split_free_lifted R = Lifted (split_free R)"

instance
proof
  fix d d' g g' :: "'a lifted"
  assume "d ≤ d'" "g ≤ g'"
  then show "split_cmb d g ≤ split_cmb d' g'"
    by (cases d; cases d'; cases g; cases g')
       (auto simp: split_cmb_lifted_def intro: split_cmb_mono)
next
  fix x g :: "'a lifted"
  show "split_cmb (split_rl x) (split_rg x) = x"
    by (cases x) (simp_all add: split_cmb_lifted_def split_rl_lifted_def split_rg_lifted_def
        split_recombine)
  show "split_cmb (split_rl x) g = split_cmb x g"
    by (cases x) (simp_all add: split_cmb_lifted_def split_rl_lifted_def split_cmb_rl)
next
  fix d g :: "'a lifted"
  show "split_rl (split_cmb d g) = split_rl d"
    by (cases d) (simp_all add: split_cmb_lifted_def split_rl_lifted_def split_rl_cmb)
next
  fix d d' :: "'a lifted" and x
  assume "d ≤ d'"
  then show "split_global_at x d ≤ split_global_at x d'"
    by (cases d; cases d') (simp_all add: split_global_at_lifted_def split_global_at_mono)
next
  fix n :: vname and R and d :: "'a lifted"
  assume "n ∉ set R"
  then show "split_global_at n d ≤ split_free R"
    by (cases d) (simp_all add: split_global_at_lifted_def split_free_lifted_def split_free_ge)
next
  fix n :: vname
  show "split_global_at n (bot :: 'a lifted) = bot"
    by (simp add: split_global_at_lifted_def)
qed

end

lemma split_cmb_Bot [simp]:
  "split_cmb (Bot :: 'a::{ownership_split, semilattice_sup} lifted) g = Bot"
  by (simp add: split_cmb_lifted_def)

instantiation mcp_st :: ownership_split
begin

definition split_cmb_mcp_st :: "mcp_st ⇒ mcp_st ⇒ mcp_st" where
  "split_cmb_mcp_st d g = MCP_St (split_cmb (mcp_fields d) (mcp_fields g))"

definition split_rl_mcp_st :: "mcp_st ⇒ mcp_st" where
  "split_rl_mcp_st d = MCP_St (split_rl (mcp_fields d))"

definition split_rg_mcp_st :: "mcp_st ⇒ mcp_st" where
  "split_rg_mcp_st d = MCP_St (split_rg (mcp_fields d))"
```

```
definition split_global_at_mcp_st :: "vname ⇒ mcp_st ⇒ mcp_st" where
  "split_global_at_mcp_st x d = MCP_St (split_global_at x (mcp_fields d))"

definition split_free_mcp_st :: "vname list ⇒ mcp_st" where
  "split_free_mcp_st R = MCP_St (split_free R)"

instance
  by standard
    (simp_all add: split_cmb_mcp_st_def split_rl_mcp_st_def split_rg_mcp_st_def
      split_global_at_mcp_st_def split_free_mcp_st_def less_eq_mcp_st_def bot_mcp_st_def
      split_cmb_mono split_cmb_rl split_rl_cmb split_global_at_mono split_free_ge
      split_global_at_bot,
     metis split_cmb_rl split_recombine mcp_st.collapse)

end

lemmas split_mcp_st_defs = split_cmb_mcp_st_def split_rl_mcp_st_def split_rg_mcp_st_def
  split_global_at_mcp_st_def split_free_mcp_st_def
```

Reading a keyed recombination back

A keyed point's state is its local half recombined with the solved environment over the declared globals. Read back, every local comes from the point and every declared global from that global's own unknown. The lemmas below establish this for one field and lift it through the fields of the combined state.

```
lemma split_global_at_default_st_get [simp]:
  "((split_global_at x d)⟨loc⟩) =
     (if loc = Global_Location x then (d⟨loc⟩) else bot)"
  by (auto simp: split_global_at_default_st_def)

lemma view_of_default_st_get:
  "(view_of split_global_at xs E acc :: 'a::{bounded_semilattice_sup_bot, order_top} default_st)⟨loc⟩
     = acc⟨loc⟩ ⊔ (case loc of Global_Location y ⇒ if y ∈ set xs then (E y)⟨loc⟩ else bot
                        | Local_Location y ⇒ bot)"
  unfolding view_of_def
  by (induction xs arbitrary: acc)
     (auto simp: split_global_at_default_st_def ac_simps split: location.split)
```

The value a lifted state carries, `bot` when unreachable.

```
fun lift_val :: "'a::bot lifted ⇒ 'a" where
  "lift_val Bot = bot"
| "lift_val (Lifted a) = a"

lemma lift_val_sup:
  "lift_val (a ⊔ b) = lift_val a ⊔ lift_val (b :: 'a::bounded_semilattice_sup_bot lifted)"
  by (cases a; cases b) simp_all

lemma split_cmb_Lifted:
  "split_cmb (Lifted a) g = Lifted (split_cmb a (lift_val (g :: 'a::{ownership_split, semilattice_sup} lifted)))"
  by (cases g) (simp_all add: split_cmb_lifted_def)

lemma lift_val_view_of:
  "lift_val (view_of split_global_at xs E acc :: 'a::{ownership_split, bounded_semilattice_sup_bot} lifted)
     = view_of split_global_at xs (λy. lift_val (E y)) (lift_val acc)"
  unfolding view_of_def
proof (induction xs arbitrary: acc)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  have "lift_val (split_global_at x (E x)) = split_global_at x (lift_val (E x))"
    by (cases "E x") (simp_all add: split_global_at_lifted_def split_global_at_bot)
  with Cons show ?case by (simp add: lift_val_sup)
qed
```

A map between carriers that commutes with the split operations, such as the projection onto one field of a product. Read-back facts proved for a field lift along it.

```
locale split_hom =
  fixes f :: "'a::{ownership_split, bounded_semilattice_sup_bot}
              ⇒ 'b::{ownership_split, bounded_semilattice_sup_bot}"
  assumes hom_cmb: "f (split_cmb a b) = split_cmb (f a) (f b)"
    and hom_rl: "f (split_rl a) = split_rl (f a)"
    and hom_at: "f (split_global_at x a) = split_global_at x (f a)"
    and hom_sup: "f (a ⊔ b) = f a ⊔ f b"
```

```
    and hom_bot: "f bot = bot"
begin

lemma hom_view_of:
  "f (view_of split_global_at xs E acc) = view_of split_global_at xs (λy. f (E y)) (f acc)"
  unfolding view_of_def by (induction xs arbitrary: acc) (simp_all add: hom_sup hom_at)

lemma lift_get_hom: "lift_get f x = f (lift_val x)"
  by (cases x) (simp_all add: hom_bot)

end
```

The read-back of one field of a keyed recombination: a local reads from the point, a declared global from the field of that global's unknown.

```
lemma keyed_field_read:
  fixes f :: "'a::{ownership_split, bounded_semilattice_sup_bot}
              ⇒ 'c::{ownership_split, bounded_semilattice_sup_bot} lifted"
    and E :: "vname ⇒ 'a lifted"
  assumes hom: "split_hom f" and fa: "f a = Lifted st"
  shows "lift_get f (split_cmb (Lifted a) (full_view split_global_at xs E))
    = Lifted (split_cmb st (view_of split_global_at xs (λy. lift_val (lift_get f (E y))) bot))"
proof -
  interpret split_hom f by (rule hom)
  have "lift_get f (split_cmb (Lifted a) (full_view split_global_at xs E))
     = split_cmb (f a) (view_of split_global_at xs (λy. lift_get f (E y)) bot)"
    by (simp add: split_cmb_Lifted full_view_def hom_cmb hom_view_of hom_bot lift_val_view_of
       lift_get_hom)
  then show ?thesis
    using fa by (simp add: split_cmb_Lifted lift_val_view_of)
qed

lemma keyed_field_gamma:
  fixes f ::
    "'a::{ownership_split, bounded_semilattice_sup_bot} ⇒ 'c::numeric_domain default_st lifted"
    and E :: "vname ⇒ 'a lifted"
  assumes hom: "split_hom f" and fa: "f a = Lifted st" and G: "⋀y. 𝒢 y ⟷ y ∈ set xs"
  shows "s ∈ gamma_lift (default_st_gamma 𝒢) (lift_get f (split_cmb (Lifted a) (full_view split_global_at xs E)))
    ⟷ (∀y. s y ∈ γ (if y ∈ set xs then (lift_val (lift_get f (E y)))⟨Global_Location y⟩
                    else st⟨Local_Location y⟩))"
  by (simp add: keyed_field_read[OF hom fa] gamma_lift_def default_st_gamma_def gamma_state_def
     default_st_to_fun_def location_of_def G split_cmb_default_st_def view_of_default_st_get)

lemma default_st_gamma_iff:
  "s ∈ default_st_gamma 𝒢 st ⟷ (∀y. s y ∈ γ (st⟨location_of 𝒢 y⟩))"
  by (simp add: default_st_gamma_def gamma_state_def default_st_to_fun_def)
```

The frame law for one field: a store the transfer's result describes, agreeing outside the written names with a store the point described, is described by the result's locals and written globals together with the environment elsewhere.

```
lemma keyed_field_mix:
  fixes f ::
    "'a::{ownership_split, bounded_semilattice_sup_bot} ⇒ 'c::numeric_domain default_st lifted"
  assumes hom: "split_hom f" and G: "⋀y. 𝒢 y ⟷ y ∈ set xs"
    and s:
      "s ∈ gamma_lift (default_st_gamma 𝒢) (lift_get f (split_cmb l0 (full_view split_global_at xs e)))"
    and s': "s' ∈ gamma_lift (default_st_gamma 𝒢) (lift_get f r)"
    and fr: "∀x. 𝒢 x ⟶ x ∉ set W ⟶ s' x = s x"
  shows "s' ∈ gamma_lift (default_st_gamma 𝒢) (lift_get f (split_cmb (split_rl r)
          (full_view split_global_at xs (λx. if x ∈ set W then split_global_at x r else e x))))"
proof -
  interpret split_hom f by (rule hom)
  obtain ri where r: "r = Lifted ri" using s' by (cases r) simp_all
  obtain stR where fr_i: "f ri = Lifted stR" using s' r by (cases "f ri") simp_all
  obtain a0 where l0: "l0 = Lifted a0" using s by (cases l0) simp_all
  obtain st0 where fa0: "f a0 = Lifted st0"
    using s l0 by (cases "f a0") (simp_all add: split_cmb_Lifted hom_cmb)
  have rl: "split_rl r = Lifted (split_rl ri)" by (simp add: r split_rl_lifted_def)
  have frl: "f (split_rl ri) = Lifted (split_rl stR)" by (simp add: hom_rl fr_i split_rl_lifted_def)
  have s'R: "∀y. s' y ∈ γ (stR⟨location_of 𝒢 y⟩)"
    using s' by (simp add: r fr_i default_st_gamma_iff)
  have s0: "∀y. s y ∈ γ (if y ∈ set xs then (lift_val (lift_get f (e y)))⟨Global_Location y⟩
                    else st0⟨Local_Location y⟩)"
    using s by (simp add: l0 keyed_field_gamma[OF hom fa0 G])
  show ?thesis
    unfolding rl keyed_field_gamma[OF hom frl G]
```

```
  proof
    fix y
    show "s' y ∈ γ (if y ∈ set xs then (lift_val (lift_get f
        (if y ∈ set W then split_global_at y r else e y)))⟨Global_Location y⟩
        else (split_rl stR)⟨Local_Location y⟩)"
    proof (cases "y ∈ set xs")
      case False
      then show ?thesis using s'R[rule_format, of y]
        by (simp add: G location_of_def split_rl_default_st_def)
    next
      case xs: True
      show ?thesis
      proof (cases "y ∈ set W")
        case True
        then show ?thesis using xs s'R[rule_format, of y]
          by (simp add: G location_of_def r split_global_at_lifted_def hom_at fr_i)
      next
        case False
        then show ?thesis using xs s0[rule_format, of y] fr G by simp
      qed
    qed
  qed
qed
```

A state is described by its local half recombined with its own cuts.

```
lemma keyed_field_init:
  fixes f ::
    "'a::{ownership_split, bounded_semilattice_sup_bot} ⇒ 'c::numeric_domain default_st lifted"
  assumes hom: "split_hom f" and G: "⋀y. 𝒢 y ⟷ y ∈ set xs"
  shows "gamma_lift (default_st_gamma 𝒢) (lift_get f (Lifted i))
    ⊆ gamma_lift (default_st_gamma 𝒢) (lift_get f (split_cmb (Lifted i)
        (full_view split_global_at xs (λx. split_global_at x (Lifted i)))))"
proof
  interpret split_hom f by (rule hom)
  fix s assume s: "s ∈ gamma_lift (default_st_gamma 𝒢) (lift_get f (Lifted i))"
  then obtain st where fi: "f i = Lifted st" by (cases "f i") simp_all
  show "s ∈ gamma_lift (default_st_gamma 𝒢) (lift_get f (split_cmb (Lifted i)
      (full_view split_global_at xs (λx. split_global_at x (Lifted i)))))"
    using s unfolding keyed_field_gamma[OF hom fi G]
    by (auto simp: fi default_st_gamma_iff location_of_def G split_global_at_lifted_def hom_at
        split: if_splits)
qed
```

Each numeric field of the combined state is reached through a slot that commutes with the split operations, because they are defined field by field.

```
lemma split_hom_pleft: "split_hom (pleft :: ('a::{ownership_split, bounded_semilattice_sup_bot},
    'b::{ownership_split, bounded_semilattice_sup_bot}) analysis_product ⇒ 'a)"
  by unfold_locales
     (simp_all add: split_cmb_analysis_product_def split_rl_analysis_product_def
       split_global_at_analysis_product_def sup_analysis_product_def bot_analysis_product_def)

lemma split_hom_pright: "split_hom (pright :: ('a::{ownership_split, bounded_semilattice_sup_bot},
    'b::{ownership_split, bounded_semilattice_sup_bot}) analysis_product ⇒ 'b)"
  by unfold_locales
     (simp_all add: split_cmb_analysis_product_def split_rl_analysis_product_def
       split_global_at_analysis_product_def sup_analysis_product_def bot_analysis_product_def)

lemma split_hom_comp: "split_hom f ⟹ split_hom g ⟹ split_hom (f ∘ g)"
  unfolding split_hom_def by simp

lemma split_hom_mcp_fields: "split_hom mcp_fields"
  by unfold_locales (simp_all add: split_mcp_st_defs sup_mcp_st_def bot_mcp_st_def)

lemma split_hom_slots:
  "split_hom (st_get slot1)" "split_hom (st_get slot2)" "split_hom (st_get slot3)"
  "split_hom (st_get slot4)" "split_hom (st_get slot5)" "split_hom (st_get slot6)"
  "split_hom (st_get slot7)"
proof -
  note h = split_hom_comp split_hom_pleft split_hom_pright split_hom_mcp_fields
  have eqs: "st_get slot1 = pleft ∘ mcp_fields"
    "st_get slot2 = pleft ∘ pright ∘ mcp_fields"
    "st_get slot3 = pleft ∘ pright ∘ pright ∘ mcp_fields"
    "st_get slot4 = pleft ∘ pright ∘ pright ∘ pright ∘ mcp_fields"
    "st_get slot5 = pleft ∘ pright ∘ pright ∘ pright ∘ pright ∘ mcp_fields"
    "st_get slot6 = pleft ∘ pright ∘ pright ∘ pright ∘ pright ∘ pright ∘ mcp_fields"
```

```
    "st_get slot7 =
       pleft ∘ pright ∘ pright ∘ pright ∘ pright ∘ pright ∘ pright ∘ mcp_fields"
    by (simp_all add: fun_eq_iff)
  show "split_hom (st_get slot1)" "split_hom (st_get slot2)" "split_hom (st_get slot3)"
    "split_hom (st_get slot4)" "split_hom (st_get slot5)" "split_hom (st_get slot6)"
    "split_hom (st_get slot7)"
    unfolding eqs by (intro h)+
qed
```

The relational field keeps its whole state locally, so recombining leaves it alone.

```
lemma slot8_keyed:
  "lift_get (st_get slot8) (split_cmb (split_rl r) g) = lift_get (st_get slot8) r"
  by (cases r) (simp_all add: split_cmb_Lifted split_rl_lifted_def split_mcp_st_defs
      split_cmb_analysis_product_def split_rl_analysis_product_def split_cmb_relc_def
      split_rl_relc_def)

lemma slot8_keyed_init:
  "lift_get (st_get slot8) (split_cmb (Lifted i) g) = st_get slot8 i"
  by (simp add: split_cmb_Lifted split_mcp_st_defs split_cmb_analysis_product_def
      split_cmb_relc_def)

lemma part_gamma_keyed_mix:
  assumes G: "⋀y. 𝒢 y ⟷ y ∈ set xs"
    and s: "s ∈ part_gamma 𝒢 a (split_cmb l0 (full_view split_global_at xs e))"
    and s': "s' ∈ part_gamma 𝒢 a r"
    and fr: "∀x. 𝒢 x ⟶ x ∉ set W ⟶ s' x = s x"
  shows "s' ∈ part_gamma 𝒢 a (split_cmb (split_rl r)
          (full_view split_global_at xs (λx. if x ∈ set W then split_global_at x r else e x)))"
  using s s'
  by (cases a rule: analysis_domain_cases)
     (simp_all only: part_gamma.simps slot8_keyed,
      (rule keyed_field_mix[OF split_hom_slots(1) G _ _ fr]
        keyed_field_mix[OF split_hom_slots(2) G _ _ fr]
        keyed_field_mix[OF split_hom_slots(3) G _ _ fr]
          keyed_field_mix[OF split_hom_slots(4) G _ _ fr]
        keyed_field_mix[OF split_hom_slots(5) G _ _ fr]
          keyed_field_mix[OF split_hom_slots(6) G _ _ fr]
        keyed_field_mix[OF split_hom_slots(7) G _ _ fr]; assumption)+)

lemma part_gamma_keyed_init:
  assumes G: "⋀y. 𝒢 y ⟷ y ∈ set xs"
  shows "part_gamma 𝒢 a (Lifted i) ⊆ part_gamma 𝒢 a (split_cmb (Lifted i)
          (full_view split_global_at xs (λx. split_global_at x (Lifted i))))"
  by (cases a rule: analysis_domain_cases)
     (simp_all only: part_gamma.simps slot8_keyed_init,
      (rule keyed_field_init[OF split_hom_slots(1) G] keyed_field_init[OF split_hom_slots(2) G]
        keyed_field_init[OF split_hom_slots(3) G] keyed_field_init[OF split_hom_slots(4) G]
        keyed_field_init[OF split_hom_slots(5) G] keyed_field_init[OF split_hom_slots(6) G]
        keyed_field_init[OF split_hom_slots(7) G] | simp)+)
```

The registration

The keyed placement: each declared global at its own unknown `global_of x`, read and published by name. The combined analysis owes it what it owes the whole-state placement, plus the carrier laws above.

```
lemma mcp_keyed_dg_analysis:
  fixes buffer_key :: 'k and global_of :: "vname ⇒ 'k"
  assumes seed_ne: "⋀v ctx. seed v ctx ≠ buffer_key"
    and seed_ne_global: "⋀v ctx n. seed v ctx ≠ global_of n"
  shows "dg_analysis (mcp_comp (activation as)) (mcp_emp (activation as)) mcp_rd
    (mcp_init (activation as)) buffer_key global_of seed
    (TD_side_rule_Interp_solve r)
    (TD_side_rule_Interp.solve_dom TYPE('k) TYPE((mcp_st lifted, mcp_st lifted) dg_state) r)
    ⊥ (mcp_classify (activation as)) (mcp_gamma_v (activation as))
    (mcp_empty_v (activation as)) (TD_side_rule_Interp_solve_c r)
    (λp c. keyed_split_spec (declared_global p) split_cmb split_rl split_global_at split_free c)
    (λp d e. split_cmb d (full_view split_global_at (declared_global_vars p) e))
    (λ𝒢. split_rl) (λ𝒢 d. Bot)
    (λ𝒢 p. map (λx. (x, split_global_at x (Lifted (mcp_init (activation as)))))
       (declared_global_vars p))
    (λp ci d e. split_cmb d (view_of split_global_at (call_global_reads (declared_global p)
       (ci_args ci)) e (split_free (call_global_reads (declared_global p) (ci_args ci)))))"
proof (rule dg_analysis_keyedI[OF td_certified_solver],
    goal_cases CompSound EnterSingle CmbMono CmbRl RlCmb CmbBot RgMono RgFree Mix InitView
    EmptyRd EmptyVSound SeedNe SeedNeGlobal ClProved ClRefuted BotState Init)
  case (CompSound p)
```

```
  have eq: "mcp_gamma (map (part_gamma (declared_global p)) (activation as))
      = (λd. gamma_lift (mcp_gamma_v (activation as)) (map_lift (mcp_rd (declared_global p)) d))"
    by (rule ext) (rule mcp_gamma_rd[OF activation_ne])
  show ?case using mcp_comp_sound[of "activation as" p] unfolding eq by simp
next
  case (EnterSingle p ci d)
  show ?case
    using single_entryD[OF single_entry_mcp_comp[OF activation_ne],
        of as "declared_global p" p
          "ls_channel (mcp_comp (activation as) (declared_global p) p) d" ci "(d, d)"]
    by (simp add: dg_pipeline.comp_entry_def)
next
  case (CmbMono d d' g g') then show ?case by (rule split_cmb_mono)
next
  case (CmbRl x g) show ?case by (rule split_cmb_rl)
next
  case (RlCmb d g) show ?case by (rule split_rl_cmb)
next
  case (CmbBot g) show ?case by simp
next
  case (RgMono x v v') then show ?case by (rule split_global_at_mono)
next
  case (RgFree x R v) then show ?case by (rule split_free_ge)
next
  case (Mix p l0 e r s s' W)
  note eq = mcp_gamma_rd[OF activation_ne, symmetric]
  show ?case
    using Mix unfolding eq mcp_gamma_def
    by (auto intro: part_gamma_keyed_mix[where xs = "declared_global_vars p"])
next
  case (InitView p)
  note eq = mcp_gamma_rd[OF activation_ne, symmetric]
  show ?case
    unfolding eq mcp_gamma_def
    using part_gamma_keyed_init[where xs = "declared_global_vars p" and 𝒢 = "declared_global p"]
    by fastforce
next
  case (EmptyRd p s) show ?case by (rule mcp_emp_rd)
next
  case EmptyVSound show ?case by (rule sound_emptiness_mcp)
next
  case (SeedNe v ctx) show ?case by (rule seed_ne)
next
  case (SeedNeGlobal v ctx n) show ?case by (rule seed_ne_global)
next
  case (ClProved e d s) then show ?case by (rule mcp_classify_proved)
next
  case (ClRefuted e d s) then show ?case by (rule mcp_classify_refuted)
next
  case BotState show ?case by (rule mcp_gamma_v_bot[OF activation_ne])
next
  case (Init p) show ?case by (rule mcp_init_sound)
qed
```

The three context policies again, with each program global at its own unknown.

```
global_interpretation mcp_split_rule: dg_analysis
    "mcp_comp (activation as)" "mcp_emp (activation as)" mcp_rd "mcp_init (activation as)"
    Analysis_Buffer Analysis_Global Activation_Seed "λ_. route_unit" "()"
    "TD_side_rule_Interp_solve r"
    "TD_side_rule_Interp.solve_dom TYPE((vname, unit) global_unknown)
       TYPE((mcp_st lifted, mcp_st lifted) dg_state) r"
    ⊥ "mcp_classify (activation as)" "mcp_gamma_v (activation as)"
    "mcp_empty_v (activation as)" "TD_side_rule_Interp_solve_c r"
    "λp c. keyed_split_spec (declared_global p) split_cmb split_rl split_global_at split_free c"
    "λp d e. split_cmb d (full_view split_global_at (declared_global_vars p) e)"
    "λ𝒢. split_rl" "λ𝒢 d. Bot"
    "λ𝒢 p. map (λx. (x, split_global_at x (Lifted (mcp_init (activation as)))))
       (declared_global_vars p)"
    "λp ci d e. split_cmb d (view_of split_global_at (call_global_reads (declared_global p)
      (ci_args ci)) e (split_free (call_global_reads (declared_global p) (ci_args ci))))"
  for as r
  by (rule mcp_keyed_dg_analysis) simp_all

global_interpretation mcp_split_es_rule: dg_analysis
    "mcp_comp (activation as)" "mcp_emp (activation as)" mcp_rd "mcp_init (activation as)"
    Analysis_Buffer Analysis_Global Activation_Seed "mcp_formals_route (activation as)"
```

```
      mcp_root_ctx
    "TD_side_rule_Interp_solve r"
    "TD_side_rule_Interp.solve_dom TYPE((vname, mcp_ctx) global_unknown)
       TYPE((mcp_st lifted, mcp_st lifted) dg_state) r"
    ⊥ "mcp_classify (activation as)" "mcp_gamma_v (activation as)"
    "mcp_empty_v (activation as)" "TD_side_rule_Interp_solve_c r"
    "λp c. keyed_split_spec (declared_global p) split_cmb split_rl split_global_at split_free c"
    "λp d e. split_cmb d (full_view split_global_at (declared_global_vars p) e)"
    "λ𝒢. split_rl" "λ𝒢 d. Bot"
    "λ𝒢 p. map (λx. (x, split_global_at x (Lifted (mcp_init (activation as)))))
       (declared_global_vars p)"
    "λp ci d e. split_cmb d (view_of split_global_at (call_global_reads (declared_global p)
       (ci_args ci)) e (split_free (call_global_reads (declared_global p) (ci_args ci))))"
  for as r
  by (rule mcp_keyed_dg_analysis) simp_all

global_interpretation mcp_split_cs_rule: dg_analysis
    "mcp_comp (activation as)" "mcp_emp (activation as)" mcp_rd "mcp_init (activation as)"
    Analysis_Buffer Analysis_Global Activation_Seed "λ_. cs_route k" "[]"
    "TD_side_rule_Interp_solve r"
    "TD_side_rule_Interp.solve_dom TYPE((vname, cfg_node list) global_unknown)
       TYPE((mcp_st lifted, mcp_st lifted) dg_state) r"
    ⊥ "mcp_classify (activation as)" "mcp_gamma_v (activation as)"
    "mcp_empty_v (activation as)" "TD_side_rule_Interp_solve_c r"
    "λp c. keyed_split_spec (declared_global p) split_cmb split_rl split_global_at split_free c"
    "λp d e. split_cmb d (full_view split_global_at (declared_global_vars p) e)"
    "λ𝒢. split_rl" "λ𝒢 d. Bot"
    "λ𝒢 p. map (λx. (x, split_global_at x (Lifted (mcp_init (activation as)))))
       (declared_global_vars p)"
    "λp ci d e. split_cmb d (view_of split_global_at (call_global_reads (declared_global p)
       (ci_args ci)) e (split_free (call_global_reads (declared_global p) (ci_args ci))))"
  for as k r
  by (rule mcp_keyed_dg_analysis) simp_all

end
```

17.7 Analysis_Run

```
theory Analysis_Run
  imports
    Arithmetic_Diagnostics
    MCP_Split
    "HOL-Library.Code_Target_Numeral"
    "HOL-Library.Code_Abstract_Char"
begin
```

Running one configuration once

The public analysis operation. A caller hands over the three choices that decide what gets solved and a program, and receives the solve as a report: every covered point-and-context state, the contexts each call enters, the check column and the arithmetic diagnostics. One solve stands behind all of them. The report holds semantic states, never their rendering; text, graphs and report pages are built from it by a projection outside the theorems.

Nothing outside this theory decides which analyser runs: the one function below that reads the context policy names the registration that policy solves with.

The rows of a report

Points are CFG nodes and contexts are indices into the report's context list, so a consumer never has to infer which states belong together from displayed text. A row's state is the type parameter `'s`: the semantic state in a report, its rendering after a presentation has been applied.

A route lists every callee context a call enters from one caller context: an entry specification may offer several alternatives, and each may land in its own context. The empty list is a call the caller context does not take.

A state also lists, for every edge leaving its point, that edge's target and what the edge's own step makes of the state. Where the target is a join of several incoming edges – a loop head, the point after a branch – this is the only place the effect of one statement survives.

```
record 's result_state =
  state_point :: pp
  state_context :: nat
  state_value :: "'s lifted"
  state_checks :: "(exp × contextual_verdict) list"
  state_diagnostics :: "(arithmetic_obligation × contextual_verdict) list"
  state_steps :: "(pp × 's lifted) list"

record call_route =
  route_point :: pp
  route_context :: nat
  route_callee :: pname
  route_targets :: "nat list"

record result_check =
  check_point :: pp
  check_label :: check_label
  check_exp :: exp
  check_verdict :: contextual_verdict
```

A global unknown is a program global kept at its own unknown, or the seed of a procedure entry, named by the procedure and the index of the context it was entered at; a seed without a context belongs to a procedure the solve never entered.

```
datatype result_global_unknown =
    Global_Named vname
  | Global_Seed pname "nat option"

record 's result_global =
  global_unknown :: result_global_unknown
  global_state :: "'s lifted"

definition map_result_state :: "('s ⇒ 't) ⇒ 's result_state ⇒ 't result_state" where
  "map_result_state f st =
     (| state_point = state_point st,
       state_context = state_context st,
       state_value = map_lift f (state_value st),
       state_checks = state_checks st,
       state_diagnostics = state_diagnostics st,
       state_steps = map (λ(w, s). (w, map_lift f s)) (state_steps st) |)"
```

```
definition map_result_global :: "('s ⇒ 't) ⇒ 's result_global ⇒ 't result_global" where
  "map_result_global f g =
     (| global_unknown = global_unknown g,
       global_state = map_lift f (global_state g) |)"
```

The configuration

A caller's three choices travel together as one value: the analyses that run, the rule for side-effected globals, and the context policy. A configuration is valid when it activates at least one analysis and none twice.

Where a program's globals live: in every point's own state, flow-sensitively, or each at its own global unknown, as a flow-insensitive value read only by the edges that mention it.

```
datatype program_globals = Program_Globals_Flow_Sensitive | Program_Globals_Flow_Insensitive

datatype analysis_config = Analysis_Config
  (config_analyses: "analysis_domain list")
  (config_rule: globals_rule)
  (config_context: context_mode)
  (config_globals: program_globals)

definition valid_config :: "analysis_config ⇒ bool" where
  "valid_config config ⟷ config_analyses config ≠ [] ∧ distinct (config_analyses config)"
```

The report

A context is kept as the policy's own value, tagged by policy, so one report type serves every policy. The report also keeps the configuration, which names the concretization its states are read through, and the program's variables, which a rendering lists.

```
datatype report_context =
    Report_Unit
  | Report_Entry mcp_ctx
  | Report_Call_String "pp list"

record analysis_report =
  report_config :: analysis_config
  report_vars :: "vname list"
  report_cfg :: cfg
  report_contexts :: "report_context list"
  report_states :: "mcp_val result_state list"
  report_routes :: "call_route list"
  report_checks :: "result_check list"
  report_globals :: "mcp_val result_global list"
  report_diagnostics :: "arithmetic_diagnostic list"
```

One builder for every context policy

A report is built the same way whatever the context policy: list the contexts the table covers, file every covered `(point, context)` state under its context's index, ask the run which context each live call enters, and take the check and diagnostic columns off the same table. The policy contributes only how its contexts are ordered (`ctx_key`, injective) and tagged (`ctx_tag`).

Points are listed in graph order. A covered point outside the graph's node list is listed after them, so the rows hold every covered key without a separate argument about which points the solver visits.

```
fun callee_of_entry :: "cfg_node ⇒ pname" where
  "callee_of_entry (FunctionEntry p) = p"
| "callee_of_entry _ = STR ''''"

definition context_indices :: "(nat × 'c) list ⇒ 'c ⇒ nat list" where
  "context_indices indexed ctx = map fst (filter (λ(i, ctx'). ctx' = ctx) indexed)"
```

Contexts are listed in the order of an injective key, which picks each context back out of the key set by `the_elem`. HOL's own code equation for that matches only a literal one-element list, but a set built by an image or a filter is backed by a list that may repeat its element — the contexts of a context-free table are one `()` per point — and the generated code then fails with a match error. Deduplicating first is the same set.

```
lemma the_elem_set_remdups [code]:
  "the_elem (set xs) =
     (case remdups xs of
        [x] ⇒ x
      | _ ⇒ Code.abort (STR ''the_elem: not a singleton'') (λ_. the_elem (set xs)))"
proof (cases "remdups xs")
```

```
  case Nil
  then show ?thesis by simp
next
  case (Cons y ys)
  then show ?thesis
  proof (cases ys)
    case Nil
    with Cons have "set xs = {y}" by (metis set_remdups empty_set list.simps(15))
    with Cons Nil show ?thesis by simp
  next
    case (Cons z zs)
    with ‹remdups xs = y # ys› show ?thesis by simp
  qed
qed

definition ordered_by_key ::
  "('a ⇒ 'k::linorder) ⇒ 'a set ⇒ 'a list" where
  "ordered_by_key rank S =
    map
      (λk. the_elem (Set.filter (λx. rank x = k) S))
      (sorted_list_of_set (rank ‘ S))"
```

The check column: one row per compiled check edge, carrying the edge's label beside its node, its condition, and the verdict `classify_checks_verdicts` computes there. The label is what a consumer finds a source check's row by.

```
definition result_checks_of ::
    "cfg ⇒ ('ctx, 'a) solved_table ⇒ (exp ⇒ 'a ⇒ check_result) ⇒ result_check list"
where
  "result_checks_of g r classify =
     map (λ(u, a, w).
            ⦇ check_point = u, check_label = ea_check_label a, check_exp = ea_check_cond a,
              check_verdict = point_verdict r classify u (ea_check_cond a) ⦈)
       (filter (λ(u, a, w). is_EA_Check a) (cfg_intra_list g))"
```

Which callee contexts a call enters from a caller context: the one the solve itself routed it to, or none when the entered state is unreachable, which is exactly when the solver's call tree skips the seed publication.

```
definition live_targets ::
    "(pp ⇒ 'c ⇒ call_action ⇒ pname ⇒ 'c option)
       ⇒ pp ⇒ 'c ⇒ call_action ⇒ pname ⇒ 'c list" where
  "live_targets succ u ctx ca q = (case succ u ctx ca q of None ⇒ [] | Some ctx' ⇒ [ctx'])"
```

The globals column lists the global unknowns of the constraint system: with flow-insensitive program globals, one unknown per declared global, then for every procedure the seed of each context its entry was solved at. A procedure entered at no context is listed once, unreachable. Flow-sensitive program globals sit in every point's own state and have no unknown of their own.

One row: the state the table filed at a point under a context, with the verdicts of the checks and arithmetic obligations there and the steps of the edges leaving it. The check, obligation and step lists come grouped by point, built once per report.

```
definition report_row ::
    "('c, vname, mcp_val) solved_run ⇒ (exp ⇒ mcp_val ⇒ check_result)
       ⇒ (pp, exp list) rbt ⇒ (pp, arithmetic_obligation list list) rbt
       ⇒ (pp, (edge_action × pp) list) rbt ⇒ nat ⇒ 'c ⇒ pp ⇒ mcp_val result_state" where
  "report_row sr classify conds obls edges i ctx v =
     (let state = lookup_table (run_table sr) v ctx
      in ⦇ state_point = v, state_context = i,
           state_value = state,
           state_checks =
             map (λcond. (cond, classify_point classify cond state)) (group_lookup conds v),
           state_diagnostics =
             map (λobligation. (obligation,
                                classify_point classify (arithmetic_condition obligation) state))
               (concat (group_lookup obls v)),
           state_steps = map (λ(a, w). (w, run_step sr v ctx a)) (group_lookup edges v) ⦈)"

lemma report_row_simps [simp]:
  "state_point (report_row sr classify conds obls edges i ctx v) = v"
  "state_context (report_row sr classify conds obls edges i ctx v) = i"
  "state_value (report_row sr classify conds obls edges i ctx v)
     = lookup_table (run_table sr) v ctx"
  by (simp_all add: report_row_def Let_def)
```

```
definition report_of ::
    "analysis_config ⇒ ('c ⇒ order_key) ⇒ ('c ⇒ report_context)
       ⇒ (exp ⇒ mcp_val ⇒ check_result) ⇒ ('c, vname, mcp_val) solved_run ⇒ imp_prog
       ⇒ analysis_report" where
  "report_of config ctx_key ctx_tag classify sr p =
     (let g = prog_cfg p;
          r = run_table sr;
          ctxs = ordered_by_key ctx_key (snd ‘ covered_keys r);
          indexed = enumerate 0 ctxs;
          nodes = cfg_node_list g
            @ sorted_list_of_set (fst ‘ covered_keys r - set (cfg_node_list g));
          intra = cfg_intra_list g;
          steps = group_by_key (λ(u, a, w). u) (λ(u, a, w). Some (a, w)) intra;
          checks = group_by_key (λ(u, a, w). u)
            (λ(u, a, w). if is_EA_Check a then Some (ea_check_cond a) else None) intra;
          obligations = group_by_key fst (Some ∘ snd) (arithmetic_sites g);
          state_at = report_row sr classify checks obligations steps;
          route_at = (λu ca ce i ctx.
            ⦇ route_point = u, route_context = i, route_callee = callee_of_entry ce,
              route_targets =
                (case lookup_table r u ctx of
                   Bot ⇒ []
                 | Lifted _ ⇒
                     remdups (concat (map (context_indices indexed)
                       (live_targets (run_succ sr) u ctx ca (callee_of_entry ce))))) ⦈);
          seeds_of = (λf.
            (case filter (λ(i, ctx). (FunctionEntry f, ctx) ∈ covered_keys r) indexed of
               [] ⇒ [⦇ global_unknown = Global_Seed f None, global_state = Bot ⦈]
             | entered ⇒
                 map (λ(i, ctx). ⦇ global_unknown = Global_Seed f (Some i),
                                   global_state = run_seed sr f ctx ⦈)
                   entered))
      in ⦇ report_config = config,
           report_vars = program_vars p,
           report_cfg = g,
           report_contexts = map ctx_tag ctxs,
           report_states =
             concat (map (λ(i, ctx).
                           map (state_at i ctx)
                             (filter (λv. (v, ctx) ∈ covered_keys r) nodes))
                         indexed),
           report_routes =
             concat (map (λ(u, ca, ce, after).
                           map (λ(i, ctx). route_at u ca ce i ctx)
                             (filter (λ(i, ctx). (u, ctx) ∈ covered_keys r) indexed))
                       (cfg_calls_list g)),
           report_checks = result_checks_of g r classify,
           report_globals =
             (case config_globals config of
                Program_Globals_Flow_Sensitive ⇒ []
              | Program_Globals_Flow_Insensitive ⇒
                  map (λx. ⦇ global_unknown = Global_Named x, global_state = run_shared sr x ⦈)
                    (declared_global_vars p))
               @ concat (map seeds_of (prog_main_name # prog_procs p)),
           report_diagnostics = arithmetic_diagnostics g r classify ⦈)"
```

One configuration, one report

An entry-state context reads as the formal values the active analyses key it by, in activation order, and is listed in the order of those values. Those values alone do not tell every context apart — they skip the fields of inactive analyses — so the whole context breaks ties, which makes the key injective and leaves the order of contexts that the values already tell apart unchanged.

```
definition mcp_ctx_values :: "analysis_domain list ⇒ mcp_ctx ⇒ abstract_value list" where
  "mcp_ctx_values as ctx = concat (map (λa. context_values (registration_of a) ctx) as)"

definition entry_ctx_key :: "analysis_domain list ⇒ mcp_ctx ⇒ order_key" where
  "entry_ctx_key as ctx =
     Key_List [Key_List (map abstract_value_key (mcp_ctx_values as ctx)), mcp_ctx_key ctx]"

lemma entry_ctx_key_inject: "entry_ctx_key as a = entry_ctx_key as b ⟹ a = b"
  by (simp add: entry_ctx_key_def mcp_ctx_key_inject)
```

The one place the context policy is read. Each policy names the registration the active analyses run as under the chosen global update rule, solves its equations with the executable solver from the program exit at the root

context, and reads the report off the answer. The logical `None` branch of `solve_c` yields no report. Where the solve diverges, the generated code does not return at all, so that branch is not an operational timeout result.

The executable boundary, specialised once. The registrations are generic in their carrier; these three constants fix it at the combined state and leave only the context and global-key types open. Generated code then builds the carrier's equality and lattice dictionaries inside each constant rather than inline in every branch below.

The placement the configuration chooses: the component as it is, or wrapped by the keyed lifter with each program global at its own unknown and a point's state recombined against the solved global environment.

```
definition mcp_place_spec :: "program_globals ⇒ imp_prog ⇒ mcp_st lifted local_spec
    ⇒ (pp × 'c, 'k, vname, mcp_st lifted, mcp_st lifted) dg_spec" where
  "mcp_place_spec pg = (case pg of
     Program_Globals_Flow_Sensitive ⇒ (λ𝒢 c. dg_spec_of c)
   | Program_Globals_Flow_Insensitive ⇒ (λp c. keyed_split_spec (declared_global p)
       split_cmb split_rl split_global_at split_free c))"

definition mcp_place_cmb ::
    "program_globals ⇒ imp_prog ⇒ mcp_st lifted ⇒ (vname ⇒ mcp_st lifted)
       ⇒ mcp_st lifted" where
  "mcp_place_cmb pg = (case pg of
     Program_Globals_Flow_Sensitive ⇒ (λ𝒢 d e. d)
   | Program_Globals_Flow_Insensitive ⇒
       (λp d e. split_cmb d (full_view split_global_at (declared_global_vars p) e)))"

definition mcp_place_rl :: "program_globals ⇒ (vname ⇒ bool) ⇒ mcp_st lifted ⇒ mcp_st lifted" where
  "mcp_place_rl pg = (case pg of
     Program_Globals_Flow_Sensitive ⇒ (λ𝒢 d. d) | Program_Globals_Flow_Insensitive ⇒ (λ𝒢. split_rl))"

definition mcp_place_rg :: "program_globals ⇒ (vname ⇒ bool) ⇒ mcp_st lifted ⇒ mcp_st lifted" where
  "mcp_place_rg pg = (λ𝒢 d. Bot)"

definition mcp_place_inits :: "program_globals ⇒ analysis_domain list
    ⇒ (vname ⇒ bool) ⇒ imp_prog ⇒ (vname × mcp_st lifted) list" where
  "mcp_place_inits pg as = (case pg of
     Program_Globals_Flow_Sensitive ⇒ (λ𝒢 p. [])
   | Program_Globals_Flow_Insensitive ⇒
       (λ𝒢 p. map (λx. (x, split_global_at x (Lifted (mcp_init as)))) (declared_global_vars p)))"

definition mcp_place_enter ::
    "program_globals ⇒ imp_prog ⇒ call_info ⇒ mcp_st lifted ⇒ (vname ⇒ mcp_st lifted)
       ⇒ mcp_st lifted" where
  "mcp_place_enter pg = (case pg of
     Program_Globals_Flow_Sensitive ⇒ (λp ci d e. d)
   | Program_Globals_Flow_Insensitive ⇒
       (λp ci d e. split_cmb d (view_of split_global_at (call_global_reads (declared_global p)
          (ci_args ci)) e (split_free (call_global_reads (declared_global p) (ci_args ci))))))"

lemmas mcp_place_defs = mcp_place_spec_def mcp_place_cmb_def mcp_place_rl_def mcp_place_rg_def
  mcp_place_inits_def mcp_place_enter_def

definition mcp_equations ::
    "analysis_domain list ⇒ program_globals ⇒ 'k ⇒ (vname ⇒ 'k) ⇒ (pp ⇒ 'c ⇒ 'k)
       ⇒ ((vname ⇒ bool) ⇒ pp ⇒ 'c ⇒ mcp_st lifted ⇒ call_action ⇒ 'c) ⇒ 'c
       ⇒ (vname ⇒ bool) ⇒ imp_prog
       ⇒ (pp × 'c, 'k, (mcp_st lifted, mcp_st lifted) dg_state) eqsT" where
  "mcp_equations as pg buffer global seed route root =
     dg_pipeline.equations (mcp_comp as) (mcp_init as) buffer global seed route root
       (mcp_place_spec pg) (mcp_place_rg pg) (mcp_place_inits pg as)"

definition mcp_solve_c ::
    "globals_rule ⇒ (pp × 'c, 'k, (mcp_st lifted, mcp_st lifted) dg_state) eqsT ⇒ pp × 'c
       ⇒ ((pp × 'c) set × (pp × 'c + 'k ⇒ (mcp_st lifted, mcp_st lifted) dg_state)) option"
  where
  "mcp_solve_c = TD_side_rule_Interp_solve_c"

definition mcp_run_of ::
    "analysis_domain list ⇒ program_globals ⇒ (vname ⇒ 'k) ⇒ (pp ⇒ 'c ⇒ 'k)
       ⇒ ((vname ⇒ bool) ⇒ pp ⇒ 'c ⇒ mcp_st lifted ⇒ call_action ⇒ 'c)
       ⇒ (vname ⇒ bool) ⇒ imp_prog
       ⇒ (pp × 'c) set × (pp × 'c + 'k ⇒ (mcp_st lifted, mcp_st lifted) dg_state)
       ⇒ ('c, vname, mcp_val) solved_run" where
  "mcp_run_of as pg global seed route =
     dg_pipeline.solved_run_of (mcp_comp as) (mcp_emp as) mcp_rd global seed route
       (mcp_place_cmb pg) (mcp_place_rl pg) (mcp_place_enter pg)"
```

```
lemmas mcp_wrappers = mcp_equations_def mcp_solve_c_def mcp_run_of_def


fun analysis_report_of :: "analysis_config ⇒ imp_prog ⇒ analysis_report option" where
  "analysis_report_of (Analysis_Config as r Ctx_None pg) p =
     map_option
       (λsol. report_of (Analysis_Config as r Ctx_None pg) (λ_. Key_List []) (λ_. Report_Unit)
          (mcp_classify (activation as))
          (mcp_run_of (activation as) pg Analysis_Global Activation_Seed (λ_. route_unit)
             (declared_global p) p sol) p)
       (mcp_solve_c r
          (mcp_equations (activation as) pg Analysis_Buffer Analysis_Global Activation_Seed
             (λ_. route_unit) () (declared_global p) p)
          (cfg_exit (prog_cfg p), ()))"
| "analysis_report_of (Analysis_Config as r Ctx_EntryState pg) p =
     map_option
       (λsol. report_of (Analysis_Config as r Ctx_EntryState pg)
          (entry_ctx_key (activation as)) Report_Entry (mcp_classify (activation as))
          (mcp_run_of (activation as) pg Analysis_Global Activation_Seed
             (mcp_formals_route (activation as)) (declared_global p) p sol) p)
       (mcp_solve_c r
          (mcp_equations (activation as) pg Analysis_Buffer Analysis_Global Activation_Seed
             (mcp_formals_route (activation as)) mcp_root_ctx (declared_global p) p)
          (cfg_exit (prog_cfg p), mcp_root_ctx))"
| "analysis_report_of (Analysis_Config as r (Ctx_CallString k) pg) p =
     map_option
       (λsol. report_of (Analysis_Config as r (Ctx_CallString k) pg)
          (λctx. Key_List (map Key_Node ctx)) Report_Call_String
          (mcp_classify (activation as))
          (mcp_run_of (activation as) pg Analysis_Global Activation_Seed
             (λ_. cs_route k) (declared_global p) p sol) p)
       (mcp_solve_c r
          (mcp_equations (activation as) pg Analysis_Buffer Analysis_Global Activation_Seed
             (λ_. cs_route k) [] (declared_global p) p)
          (cfg_exit (prog_cfg p), []))"
```

The public operation

Four answers: an invalid configuration asks for no run, a malformed program was never analysed, the logical `None` branch of the executable solve has no report, and every other run hands back its report.

```
datatype 'r analysis_answer =
    Invalid_Activation
  | Malformed_Program
  | No_Answer
  | Analysed 'r

definition run_voblint ::
  "analysis_config ⇒ imp_prog ⇒ analysis_report analysis_answer"
where
  "run_voblint config p =
     (if ¬ valid_config config then Invalid_Activation
      else if ¬ wf_program_compile_input_exec p then Malformed_Program
      else case analysis_report_of config p of
             None ⇒ No_Answer
           | Some res ⇒ Analysed res)"

end
```

17.8 Analysis_Render

```
theory Analysis_Render
  imports Analysis_Run
begin
```

The analysis graph

The report's states are filed by point *and* context, so the graph a reader sees of them has a node per such pair, where the CFG has one per point. A flow-insensitive program global adds a node of its own. The edges are the dependencies the equations have between those unknowns: a step within a context, a call entering the contexts the report routes it to, the callee's result resuming the caller, the call's continuation, and the reads and writes of each global, by the same footprints the keyed lifter reads and publishes (`edge_global_reads`, `edge_global_writes`, `call_global_reads`). It is a projection of the report, as the rendering below is, and no soundness theorem reads it.

```
datatype unknown_node = Local_Node pp nat | Global_Node vname

datatype analysis_edge =
    Intra_Dep edge_action
  | Enter_Dep call_action
  | Combine_Dep call_action
  | Continue_Dep call_action
  | Global_Read
  | Global_Write

record analysis_graph =
  graph_nodes :: "unknown_node list"
  graph_edges :: "(unknown_node × analysis_edge × unknown_node) list"
```

The program globals the report keeps at unknowns of their own.

```
definition report_global_names :: "analysis_report ⇒ vname list" where
  "report_global_names res =
     concat (map (λg. case global_unknown g of Global_Named x ⇒ [x] | _ ⇒ []) (report_globals res))"

definition report_rows_of :: "analysis_report ⇒ (pp × nat) list" where
  "report_rows_of res = map (λst. (state_point st, state_context st)) (report_states res)"

definition report_route_targets :: "analysis_report ⇒ pp ⇒ nat ⇒ nat list" where
  "report_route_targets res u c =
     concat (map route_targets
       (filter (λr. route_point r = u ∧ route_context r = c) (report_routes res)))"

definition callee_of :: "pp ⇒ pname" where
  "callee_of entry = (case entry of FunctionEntry f ⇒ f | FunctionResult f ⇒ f | Statement _ ⇒ STR '''')"
```

What a step within one context adds: the step, and the globals it reads and writes.

```
definition intra_deps ::
    "(vname ⇒ bool) ⇒ (pp × nat) list ⇒ pp ⇒ nat ⇒ pp × edge_action × pp
     ⇒ (unknown_node × analysis_edge × unknown_node) list" where
  "intra_deps G rows u c e =
     (case e of (u', a, v) ⇒
        if u' = u ∧ (v, c) ∈ set rows
        then (Local_Node u c, Intra_Dep a, Local_Node v c)
             # map (λx. (Global_Node x, Global_Read, Local_Node v c)) (edge_global_reads G a)
             @ map (λx. (Local_Node v c, Global_Write, Global_Node x)) (edge_global_writes G a)
        else [])"
```

What a call adds: an entry into every context the report routes it to, the callee's result resuming the caller from each, and the continuation. The continuation's equation evaluates the call, so it reads the globals the arguments mention and writes a global destination or formal.

```
definition call_deps ::
    "analysis_report ⇒ (vname ⇒ bool) ⇒ (pp × nat) list ⇒ pp ⇒ nat
     ⇒ pp × call_action × pp × pp ⇒ (unknown_node × analysis_edge × unknown_node) list" where
  "call_deps res G rows u c e =
     (case e of (u', ca, entry, after) ⇒
        if u' ≠ u then []
        else
          (let targets = report_route_targets res u c;
               result = FunctionResult (callee_of entry)
           in concat (map (λt.
                (if (entry, t) ∈ set rows then [(Local_Node u c, Enter_Dep ca, Local_Node entry t)] else [])
```

```
              @ (if (result, t) ∈ set rows ∧ (after, c) ∈ set rows
                 then [(Local_Node result t, Combine_Dep ca, Local_Node after c)] else []))
              targets)
           @ (if (after, c) ∈ set rows
              then (Local_Node u c, Continue_Dep ca, Local_Node after c)
                   # (case ca of CallEdge dst pars args ⇒
                        map (λx. (Global_Node x, Global_Read, Local_Node after c))
                          (call_global_reads G args)
                        @ map (λx. (Local_Node after c, Global_Write, Global_Node x))
                            (global_names_in G (case_option [] (λx. [x]) dst @ pars)))
              else []))"

definition analysis_graph_of :: "analysis_report ⇒ analysis_graph" where
  "analysis_graph_of res =
     (let g = report_cfg res;
          rows = report_rows_of res;
          names = report_global_names res;
          G = (λx. x ∈ set names)
      in ⦇ graph_nodes = map (λ(v, c). Local_Node v c) rows @ map Global_Node names,
           graph_edges =
             concat (map (λe. concat (map (λ(u, c). intra_deps G rows u c e) rows))
               (cfg_intra_list g))
             @ concat (map (λe. concat (map (λ(u, c). call_deps res G rows u c e) rows))
                 (cfg_calls_list g))
             @ concat (map (λ(u, c).
                 if u = cfg_entry g
                 then map (λx. (Local_Node u c, Global_Write, Global_Node x)) names else [])
               rows) ⦈)"
```

Rendering a report

A report holds semantic states. What a consumer displays is a projection of it: each active analysis's part of a state, as that analysis's own value, then as a string. The projection is defined here so that generated code computes it; no soundness theorem reads it, and it changes neither the check column nor the diagnostics.

The rendered result

A rendered state is shown one active analysis at a time, in activation order, each as its own `'v field_state`, as Goblint's report shows each component of its combined state. A context reads as the formal values its policy keys it by.

```
datatype 'v analysis_context =
    Context_Unit
  | Context_Entry "'v list"
  | Context_Call_String "pp list"

type_synonym 'v analysis_view = "(analysis_domain × 'v field_state) list"
```

The states a report holds at a point, joined over its contexts: what a reader who ignores contexts can say about the point. It may be coarser than the states it joins, but every store any of them describes, it describes too (`report_sem_point_join`).

A read state maps names to values, and generated code cannot carry the order of such a function, only its join, which is pointwise. `vjoin` is that join alone, per type constructor; `join_val_sup` records that it is the lattice join, so the theorems about `report_point_join` speak about ⊔.

```
class join_val =
  fixes vjoin :: "'a ⇒ 'a ⇒ 'a"
```

A join that is the lattice join of a semilattice, for the proofs only.

```
class join_val_sup = join_val + semilattice_sup +
  assumes vjoin_sup: "vjoin x y = x ⊔ y"

instantiation "fun" :: (type, semilattice_sup) join_val
begin
definition vjoin_fun :: "('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b" where
  "vjoin_fun f g = (λx. f x ⊔ g x)"
instance ..
end

instance "fun" :: (type, semilattice_sup) join_val_sup
  by standard (simp add: vjoin_fun_def sup_fun_def)
```

```
instantiation lifted :: (join_val) join_val
begin
fun vjoin_lifted :: "'a lifted ⇒ 'a lifted ⇒ 'a lifted" where
  "vjoin_lifted Bot y = y"
| "vjoin_lifted (Lifted x) Bot = Lifted x"
| "vjoin_lifted (Lifted x) (Lifted y) = Lifted (vjoin x y)"
instance ..
end

instance lifted :: (join_val_sup) join_val_sup
proof
  fix x y :: "'a lifted"
  show "vjoin x y = x ⊔ y" by (cases x; cases y) (simp_all add: vjoin_sup)
qed

instantiation analysis_product :: (join_val, join_val) join_val
begin
definition vjoin_analysis_product ::
    "('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product ⇒ ('a, 'b) analysis_product" where
  "vjoin_analysis_product p q = Product (vjoin (pleft p) (pleft q)) (vjoin (pright p) (pright q))"
instance ..
end

instance analysis_product :: (join_val_sup, join_val_sup) join_val_sup
  by standard (simp add: vjoin_analysis_product_def sup_analysis_product_def vjoin_sup)

instantiation relc :: join_val
begin
definition vjoin_relc :: "relc ⇒ relc ⇒ relc" where "vjoin_relc = sup"
instance ..
end

instance relc :: join_val_sup
  by standard (simp add: vjoin_relc_def)

definition report_point_join :: "analysis_report ⇒ pp ⇒ mcp_val lifted" where
  "report_point_join res v =
     fold vjoin (map state_value (filter (λst. state_point st = v) (report_states res))) Bot"
```

What a point's outgoing steps produce, joined over its contexts the same way. Every context lists the point's steps in the order of its outgoing edges, so the lists are joined position by position.

```
definition report_steps_join :: "analysis_report ⇒ pp ⇒ (pp × mcp_val lifted) list" where
  "report_steps_join res v =
     (case map state_steps (filter (λst. state_point st = v) (report_states res)) of
        [] ⇒ []
      | s # ss ⇒ fold (map2 (λ(w, a) (_, b). (w, vjoin a b))) ss s)"

lemma report_point_join_sup:
  "report_point_join res v =
     fold (⊔) (map state_value (filter (λst. state_point st = v) (report_states res))) Bot"
  unfolding report_point_join_def by (simp add: vjoin_sup[abs_def])

record 'v run_result =
  res_cfg :: cfg
  res_contexts :: "'v analysis_context list"
  res_states :: "'v analysis_view result_state list"
  res_joined :: "(pp × 'v analysis_view lifted × (pp × 'v analysis_view lifted) list) list"
  res_routes :: "call_route list"
  res_checks :: "result_check list"
  res_globals :: "'v analysis_view result_global list"
  res_diagnostics :: "arithmetic_diagnostic list"
  res_graph :: analysis_graph

definition map_analysis_view :: "('v ⇒ 'w) ⇒ 'v analysis_view ⇒ 'w analysis_view" where
  "map_analysis_view f = map (λ(a, s). (a, map_field_state f s))"

definition mcp_render ::
    "analysis_domain list ⇒ vname list ⇒ mcp_val ⇒ abstract_value analysis_view" where
  "mcp_render as vars v = map (λa. (a, value_display (registration_of a) v vars)) as"
```

The printer `show` is a parameter: `string_of_abstract_value` for text, composed in the adapter with whatever post-processing its output needs.

```
fun render_context ::
    "(abstract_value ⇒ 'v) ⇒ analysis_domain list ⇒ report_context ⇒ 'v analysis_context"
where
```

```
  "render_context show as Report_Unit = Context_Unit"
| "render_context show as (Report_Entry ctx) = Context_Entry (map show (mcp_ctx_values as ctx))"
| "render_context show as (Report_Call_String us) = Context_Call_String us"

definition render_report :: "(abstract_value ⇒ 'v) ⇒ analysis_report ⇒ 'v run_result" where
  "render_report show res =
     (let as = activation (config_analyses (report_config res));
          state = map_analysis_view show ∘ mcp_render as (report_vars res)
      in ⦇ res_cfg = report_cfg res,
           res_contexts = map (render_context show as) (report_contexts res),
           res_states = map (map_result_state state) (report_states res),
           res_joined = map (λv. (v, map_lift state (report_point_join res v),
                                  map (λ(w, s). (w, map_lift state s)) (report_steps_join res v)))
                            (cfg_node_list (report_cfg res)),
           res_routes = report_routes res,
           res_checks = report_checks res,
           res_globals = map (map_result_global state) (report_globals res),
           res_diagnostics = report_diagnostics res,
           res_graph = analysis_graph_of res ⦈)"

lemma render_report_columns [simp]:
  "res_cfg (render_report show res) = report_cfg res"
  "res_checks (render_report show res) = report_checks res"
  "res_diagnostics (render_report show res) = report_diagnostics res"
  "res_routes (render_report show res) = report_routes res"
  by (simp_all add: render_report_def Let_def)
```

What the analysis graph contains

The graph's nodes are exactly the report's unknowns, and a step between two reported points of one context carries the global reads and writes its footprint names.

```
lemma local_node_in_analysis_graph:
  "Local_Node v c ∈ set (graph_nodes (analysis_graph_of res))
     ⟷ (∃st ∈ set (report_states res). state_point st = v ∧ state_context st = c)"
  by (auto simp: analysis_graph_of_def report_rows_of_def Let_def)

lemma global_node_in_analysis_graph:
  "Global_Node x ∈ set (graph_nodes (analysis_graph_of res))
     ⟷ (∃gl ∈ set (report_globals res). global_unknown gl = Global_Named x)"
proof -
  have "x ∈ set (report_global_names res)
        ⟷ (∃gl ∈ set (report_globals res). global_unknown gl = Global_Named x)"
    by (force simp: report_global_names_def split: result_global_unknown.splits)
  then show ?thesis by (auto simp: analysis_graph_of_def Let_def)
qed
```

A step of a reported context contributes all its dependencies to the graph.

```
lemma intra_deps_in_analysis_graph:
  assumes "(u, c) ∈ set (report_rows_of res)" and "e ∈ set (cfg_intra_list (report_cfg res))"
  shows "set (intra_deps (λy. y ∈ set (report_global_names res)) (report_rows_of res) u c e)
         ⊆ set (graph_edges (analysis_graph_of res))"
  using assms unfolding analysis_graph_of_def Let_def
  by (auto intro!: bexI[where x = "(u, c)"] bexI[where x = e])
```

A step both of whose points the report holds in a context reads and writes, in the graph, the globals its footprint names.

```
lemma global_deps_in_analysis_graph:
  fixes res defines "G ≡ (λy. y ∈ set (report_global_names res))"
  assumes e: "(u, a, v) ∈ set (cfg_intra_list (report_cfg res))"
    and u: "(u, c) ∈ set (report_rows_of res)" and v: "(v, c) ∈ set (report_rows_of res)"
  shows "x ∈ set (edge_global_reads G a)
         ⟹ (Global_Node x, Global_Read, Local_Node v c) ∈ set (graph_edges (analysis_graph_of res))"
    and "x ∈ set (edge_global_writes G a)
         ⟹ (Local_Node v c, Global_Write, Global_Node x) ∈ set (graph_edges (analysis_graph_of res))"
proof -
  have deps: "set (intra_deps G (report_rows_of res) u c (u, a, v))
              ⊆ set (graph_edges (analysis_graph_of res))"
    unfolding G_def by (rule intra_deps_in_analysis_graph[OF u e])
  show "x ∈ set (edge_global_reads G a)
        ⟹ (Global_Node x, Global_Read, Local_Node v c) ∈ set (graph_edges (analysis_graph_of res))"
    using deps v by (auto simp: intra_deps_def)
  show "x ∈ set (edge_global_writes G a)
        ⟹ (Local_Node v c, Global_Write, Global_Node x) ∈ set (graph_edges (analysis_graph_of res))"
    using deps v by (auto simp: intra_deps_def)
```

qed

So the graph is read off the report alone: its local nodes are the reported rows (`local_node_in_analysis_graph`), its global nodes the keyed program globals (`global_node_in_analysis_graph`), and its global edges the footprints of the keyed lifter (`global_deps_in_analysis_graph`).

end

17.9 Analysis_Run_Sound

```
theory Analysis_Run_Sound
  imports Analysis_Run
begin
```

What a configuration's table owes its report

Every configuration hands the table its registration solved to `report_of`, which takes the check column from one call of `classify_checks_verdicts` and the diagnostics from one call of `arithmetic_diagnostics` over that table. So one argument serves every configuration.

The argument asks two things of the table, bundled as `covered_table`. A store the program reaches at a point lies in the entry the table filed there under *some* context (`table_covers`), and a point has finitely many contexts. Of the classifier it asks soundness in both directions (`sound_classifier`), which does not depend on the table and is proved once for the combined state. Given both, the report built from the table is sound; the theory `Analysis_Report` states how. The context-free configuration's table is the instance at the end of this theory; the contextual ones follow in the theory after it.

Every check column is a contextual verdict report

```
lemma finite_intra_prog_cfg [simp]: "finite (intra (prog_cfg p))"
  unfolding prog_cfg_def by (simp add: compile_prog_finite)
```

Forgetting the labels, the check column is the contextual verdict report itself.

```
lemma result_checks_of_verdicts:
  "map (λchk. (check_point chk, check_exp chk, check_verdict chk)) (result_checks_of g r classify)
     = classify_checks_verdicts g r classify"
  unfolding result_checks_of_def classify_checks_verdicts_def classify_checks_ctx_def
    point_verdict_def
  by (simp add: comp_def case_prod_beta image_image)
```

Where a result has checks: one per compiled `EA_Check` edge, at that edge's source node and with its label and condition, in the order the graph lists its edges.

```
definition check_sites :: "cfg ⇒ (pp × check_label × exp) list" where
  "check_sites g =
     map (λ(u, a, v). (u, ea_check_label a, ea_check_cond a))
       (filter (λ(u, a, v). is_EA_Check a) (cfg_intra_list g))"

lemma result_checks_of_sites:
  "map (λchk. (check_point chk, check_label chk, check_exp chk)) (result_checks_of g r classify)
     = check_sites g"
  unfolding result_checks_of_def check_sites_def
  by (simp add: comp_def case_prod_beta)

lemma check_sites_memI [intro]:
  assumes "finite (intra g)" and "(v, EA_Check l cnd, w) ∈ intra g"
  shows "(v, l, cnd) ∈ set (check_sites g)"
  using assms unfolding check_sites_def by force
```

A store that reaches a point sits in one of that point's contexts

The table claim at a point is existential in the context: a store is described by the entry filed under *some* context the point was solved at — one its own call history is admitted at — and in general not by every such entry, since another activation's entry need not describe this store at all.

```
definition table_covers ::
    "('v ⇒ store set) ⇒ ('c, 'v) solved_table ⇒ pp ⇒ store ⇒ bool" where
  "table_covers gm r v s ⟷ (∃ctx st. lookup_table r v ctx = Lifted st ∧ s ∈ gm st)"

lemma table_coversI [intro]:
  "lookup_table r v ctx = Lifted st ⟹ s ∈ gm st ⟹ table_covers gm r v s"
  unfolding table_covers_def by blast
```

The published contextual soundness bounds one activation bucket by the table entry filed under that bucket's context. A caller holding a store knows only that the store reaches the point at all, so it needs the buckets to exhaust the point — the union direction, which is where a context policy pays for being total. Both readings are settled in `Voblint_CFG.Activation_Trace_Collect`: a functional policy has the union outright, a relational one — such as the entry-state policy — earns it from the existence of a context for every valid activation trace. `Bot` cannot be the entry found, since it concretizes to no store at all.

```
lemma lookup_table_covers_of_activation:
  fixes r :: "('c, 'v) solved_table"
  assumes union: "C_{G,g,S} v ⊆ (⋃c. A_{G,adm,rc,g,S} v c)"
      and sound: "⋀ctx. A_{G,adm,rc,g,S} v ctx
                    ⊆ gamma_lift gm (lookup_table r v ctx)"
      and mem: "s ∈ C_{G,g,S} v"
  obtains ctx st where "s ∈ A_{G,adm,rc,g,S} v ctx"
    and "lookup_table r v ctx = Lifted st" and "s ∈ gm st"
proof -
  from mem union obtain ctx where a: "s ∈ A_{G,adm,rc,g,S} v ctx" by blast
  with sound have g: "s ∈ gamma_lift gm (lookup_table r v ctx)" by blast
  show ?thesis
  proof (cases "lookup_table r v ctx")
    case Bot
    with g show ?thesis by simp
  next
    case (Lifted st)
    with g a show ?thesis by (intro that [of ctx st]) simp_all
  qed
qed
```

The table contract and the classifier contract

What a configuration's table has to satisfy for its report to be sound. Each configuration below and in the theory after this one is an instance of `covered_table`, and owes nothing but two facts: the table exhausts the reachable stores, over finitely many contexts per point. The classifier owes soundness in both directions (`sound_classifier`); `sound_table` is the two together.

```
definition arithmetic_safe_at :: "cfg ⇒ pp ⇒ store ⇒ bool" where
  "arithmetic_safe_at g v s ⟷
     (∀es. (v, es) ∈ set (arithmetic_expression_sites g) ⟶
       (∀e ∈ set es. ∀divisor ∈ expression_divisors e. ⟦divisor⟧_e s ≠ 0))"
```

A solved table `r` covers program `p`: every store the collecting semantics reaches at `v` lies in `gm` of the state `r` holds at `v` in some context, and each point has finitely many contexts.

```
locale covered_table =
  fixes p :: imp_prog
    and r :: "('c, 'v) solved_table"
    and gm :: "'v ⇒ store set"
  assumes finite_contexts: "⋀v. finite (table_contexts r v)"
      and covers: "⋀v s. s ∈ C_{declared_global p,prog_cfg p,cinit_stores (declared_global p)} v
                    ⟹ table_covers gm r v s"
```

`classify` is sound for `gm`: `Check_Proved` means the condition holds at every store in `gm d`, `Check_Refuted` that it fails at every one.

```
locale sound_classifier =
  fixes classify :: "exp ⇒ 'v ⇒ check_result"
    and gm :: "'v ⇒ store set"
  assumes proved: "⋀cnd d t. classify cnd d = Check_Proved ⟹ t ∈ gm d
                    ⟹ truthy (⟦cnd⟧_e t)"
      and refuted: "⋀cnd d t. classify cnd d = Check_Refuted ⟹ t ∈ gm d
                    ⟹ ¬ truthy (⟦cnd⟧_e t)"

lemma mcp_sound_classifier: "sound_classifier (mcp_classify as) (mcp_gamma_v as)"
  by unfold_locales (fact mcp_classify_proved, fact mcp_classify_refuted)
```

A covered table read through a sound classifier. Inside it, verdicts and diagnostics computed from `r` hold at every collected store.

```
locale sound_table = covered_table p r gm + sound_classifier classify gm
  for p :: imp_prog and r :: "('c, 'v) solved_table"
    and classify :: "exp ⇒ 'v ⇒ check_result" and gm :: "'v ⇒ store set"
begin
```

A point the arithmetic diagnostics do not name is safe at every store the collecting semantics admits there: no divisor evaluated at it is zero.

```
lemma arithmetic_safe:
  assumes absent: "∀d ∈ set (arithmetic_diagnostics (prog_cfg p) r classify).
      diagnostic_point d ≠ v"
    and mem: "s ∈ C_{declared_global p,prog_cfg p,cinit_stores (declared_global p)} v"
  shows "arithmetic_safe_at (prog_cfg p) v s"
proof -
```

```
  from covers[OF mem] obtain ctx st where
    look: "lookup_table r v ctx = Lifted st" and gst: "s ∈ gm st"
    unfolding table_covers_def by blast
  have safe: "⋀es e divisor. (v, es) ∈ set (arithmetic_expression_sites (prog_cfg p)) ⟹
      e ∈ set es ⟹ divisor ∈ expression_divisors e ⟹ ⟦divisor⟧e s ≠ 0"
  proof -
    fix es e divisor
    assume site: "(v, es) ∈ set (arithmetic_expression_sites (prog_cfg p))"
      and expr: "e ∈ set es" and div: "divisor ∈ expression_divisors e"
    obtain obligations obligation where
      obs: "(v, obligations) ∈ set (arithmetic_sites (prog_cfg p))"
      and ob: "obligation ∈ set obligations"
      and divisor: "arithmetic_divisor obligation = divisor"
      by (rule arithmetic_sites_divisor[OF site expr div])
    have verdict: "point_verdict r classify v (arithmetic_condition obligation)
                    = Lifted Check_Proved"
      using arithmetic_diagnostics_absent[OF obs ob absent]
        point_verdict_not_dead[OF finite_contexts look, of classify]
      by blast
    have classified: "classify (arithmetic_condition obligation) st = Check_Proved"
      by (rule point_verdict_decided[OF verdict _ look]) simp
    from proved[OF classified gst] show "⟦divisor⟧e s ≠ 0"
      by (auto simp: arithmetic_condition_def divisor split: if_splits)
  qed
  show ?thesis unfolding arithmetic_safe_at_def using safe by blast
qed

end
```

The shape every registration publishes its soundness in, with its context policy `adm` and root context `rc`: every valid activation trace carries a context, and each activation bucket is bounded by the entry filed under its context. The buckets then exhaust the point, which is all a `covered_table` needs.

```
locale activation_table =
  fixes p :: imp_prog
    and r :: "('c, 'v) solved_table"
    and gm :: "'v ⇒ store set"
    and adm :: "'c context_policy"
    and rc :: 'c
  assumes has_context:
      "⋀t. t ∈ 𝒯declared_global p,prog_cfg p,cinit_stores (declared_global p)
         ⟹ ∃c. activation_context_rel (declared_global p) adm rc (prog_cfg p) t c"
    and bucket_bound:
      "⋀u ctx. 𝒜declared_global p,adm,rc,prog_cfg p,cinit_stores (declared_global p) u ctx
         ⊆ gamma_lift gm (lookup_table r u ctx)"
    and finite_table: "finite_solved_table r"
begin

lemma covered: "covered_table p r gm"
proof (rule covered_table.intro)
  show "finite (table_contexts r v)" for v by (rule finite_table_contexts [OF finite_table])
  have union: "𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) u
                ⊆ (⋃c. 𝒜declared_global p,adm,rc,prog_cfg p,                              cinit_stores (declared_global p)
u c)" for u
    by (rule equalityD1 [OF node_collect_eq_Union_activation_of_has_context [OF has_context]])
  show "table_covers gm r v s"
    if "s ∈ 𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v"
    for v s
    by (meson lookup_table_covers_of_activation [OF union bucket_bound that] table_coversI)
qed

end
```

The context-free configuration

The unit registration keys every activation at the one context `()`, a function of the call site and the caller's context, so the buckets exhaust a point outright and each is bounded by its entry.

```
lemma mcp_rule_table:
  assumes wf: "wf_program_compile_input p"
    and cov: "mcp_rule.terminates as r (declared_global p) p"
  shows "activation_table p (mcp_rule.result as r (declared_global p) p)
          (mcp_gamma_v (activation as)) (context_policy_of_fun (λu c t. ())) ()"
proof (rule activation_table.intro, goal_cases)
  case (1 t)
```

```
  show ?case
    by (rule exI, subst activation_context_rel_of_fun_iff [OF 1]) (rule refl)
next
  case (2 u ctx)
  show ?case
    using mcp_rule.fun_route_activation_collect_sound_of_terminates [OF _ wf cov]
    unfolding mcp_rule.gamma_reader_eq_lookup by (simp add: route_unit_def)
next
  case 3
  show ?case
    using mcp_rule.vars_finite_of_terminates [OF cov]
    by (simp add: finite_solved_table_def dg_pipeline.result_def
        dg_pipeline.sol_vars_def)
qed
```

The same with program globals on the shared channel.

```
lemma mcp_split_rule_table:
  assumes wf: "wf_program_compile_input p"
    and cov: "mcp_split_rule.terminates as r (declared_global p) p"
  shows "activation_table p (mcp_split_rule.result as r (declared_global p) p)
           (mcp_gamma_v (activation as)) (context_policy_of_fun (λu c t. ())) ()"
proof (rule activation_table.intro, goal_cases)
  case (1 t)
  show ?case
    by (rule exI, subst activation_context_rel_of_fun_iff [OF 1]) (rule refl)
next
  case (2 u ctx)
  show ?case
    using mcp_split_rule.fun_route_activation_collect_sound_of_terminates [OF _ wf cov]
    unfolding mcp_split_rule.gamma_reader_eq_lookup by (simp add: route_unit_def)
next
  case 3
  show ?case
    using mcp_split_rule.vars_finite_of_terminates [OF cov]
    by (simp add: finite_solved_table_def dg_pipeline.result_def
        dg_pipeline.sol_vars_def)
qed

end
```

17.10 Analysis_Run_Ctx_Sound

```
theory Analysis_Run_Ctx_Sound
  imports Analysis_Run_Sound
begin
```

The contextual configurations' tables

A context-sensitive table has one entry per point and context. Each registration publishes its soundness in activation form: the activation buckets exhaust a point — outright for a call string, which is a function of the call site and the caller's context, and for entry state from the totality of its context relation — and each bucket is bounded by the entry filed under its context. `activation_table` records these facts with the policy that earns them, and `activation_table.covered` turns them into a `covered_table`, so each configuration below names its registration's three published facts and nothing else. Every registration takes the activation list and the global update rule as parameters, so one table per context policy covers every activation and every rule.

Entry state

```
lemma mcp_es_rule_table:
  assumes wf: "wf_program_compile_input p"
    and cov: "mcp_es_rule.terminates as r (declared_global p) p"
  shows "activation_table p (mcp_es_rule.result as r (declared_global p) p)
           (mcp_gamma_v (activation as))
           (mcp_es_rule.admitted_contexts as r (declared_global p) p) mcp_root_ctx"
proof (rule activation_table.intro, goal_cases)
  case (1 t)
  show ?case
    by (rule mcp_es_rule.entry_state_has_context_of_terminates [OF wf cov 1])
next
  case (2 u ctx)
  show ?case
    using mcp_es_rule.entry_state_activation_collect_sound_of_terminates [OF wf cov]
    unfolding mcp_es_rule.gamma_reader_eq_lookup .
next
  case 3
  show ?case
    using mcp_es_rule.vars_finite_of_terminates [OF cov]
    by (simp add: finite_solved_table_def dg_pipeline.result_def
        dg_pipeline.sol_vars_def)
qed
```

Call string

`mcp_cs_rule_table`: a terminating call-string run yields a `covered_table`, via the activation collecting soundness of `mcp_cs_rule`.

```
lemma mcp_cs_rule_table:
  assumes wf: "wf_program_compile_input p"
    and cov: "mcp_cs_rule.terminates as k r (declared_global p) p"
  shows "activation_table p (mcp_cs_rule.result as k r (declared_global p) p)
           (mcp_gamma_v (activation as))
           (context_policy_of_fun (λu ctx t. cs_context k u ctx t)) []"
proof (rule activation_table.intro, goal_cases)
  case (1 t)
  show ?case
    by (rule exI, subst activation_context_rel_of_fun_iff [OF 1]) (rule refl)
next
  case (2 u ctx)
  show ?case
    using mcp_cs_rule.fun_route_activation_collect_sound_of_terminates
            [OF cs_route_context_agree wf cov]
    unfolding mcp_cs_rule.gamma_reader_eq_lookup by simp
next
  case 3
  show ?case
    using mcp_cs_rule.vars_finite_of_terminates [OF cov]
    by (simp add: finite_solved_table_def dg_pipeline.result_def
        dg_pipeline.sol_vars_def)
qed
```

Program globals on the shared channel

The two contextual tables again, at the shared-globals registrations.

```
lemma mcp_split_es_rule_table:
  assumes wf: "wf_program_compile_input p"
    and cov: "mcp_split_es_rule.terminates as r (declared_global p) p"
  shows "activation_table p (mcp_split_es_rule.result as r (declared_global p) p)
           (mcp_gamma_v (activation as))
           (mcp_split_es_rule.admitted_contexts as r (declared_global p) p) mcp_root_ctx"
proof (rule activation_table.intro, goal_cases)
  case (1 t)
  show ?case
    by (rule mcp_split_es_rule.entry_state_has_context_of_terminates [OF wf cov 1])
next
  case (2 u ctx)
  show ?case
    using mcp_split_es_rule.entry_state_activation_collect_sound_of_terminates [OF wf cov]
    unfolding mcp_split_es_rule.gamma_reader_eq_lookup .
next
  case 3
  show ?case
    using mcp_split_es_rule.vars_finite_of_terminates [OF cov]
    by (simp add: finite_solved_table_def dg_pipeline.result_def
        dg_pipeline.sol_vars_def)
qed

lemma mcp_split_cs_rule_table:
  assumes wf: "wf_program_compile_input p"
    and cov: "mcp_split_cs_rule.terminates as k r (declared_global p) p"
  shows "activation_table p (mcp_split_cs_rule.result as k r (declared_global p) p)
           (mcp_gamma_v (activation as))
           (context_policy_of_fun (λu ctx t. cs_context k u ctx t)) []"
proof (rule activation_table.intro, goal_cases)
  case (1 t)
  show ?case
    by (rule exI, subst activation_context_rel_of_fun_iff [OF 1]) (rule refl)
next
  case (2 u ctx)
  show ?case
    using mcp_split_cs_rule.fun_route_activation_collect_sound_of_terminates
           [OF cs_route_context_agree wf cov]
    unfolding mcp_split_cs_rule.gamma_reader_eq_lookup by simp
next
  case 3
  show ?case
    using mcp_split_cs_rule.vars_finite_of_terminates [OF cov]
    by (simp add: finite_solved_table_def dg_pipeline.result_def
        dg_pipeline.sol_vars_def)
qed

end
```

17.11 Analysis_Report

```
theory Analysis_Report
  imports Analysis_Run_Ctx_Sound Analysis_Render
begin
```

What a report claims

A report is read through three sets of stores at a point `v`. $\mathcal{C}$ `v` holds the stores the program reaches at `v`. $\mathcal{R}_{\mathtt{res}}$ `v` holds the stores the report's states at `v` describe, over all of its contexts. $\mathcal{V}_{\mathtt{res}}$ `v` holds the stores in which every definite verdict the report gives at `v` is valid. The report is sound when

$\mathcal{C}$ `v` $\subseteq$ $\mathcal{R}_{\mathtt{res}}$ `v` $\subseteq$ $\mathcal{V}_{\mathtt{res}}$ `v`

and a point the report calls dead has $\mathcal{R}_{\mathtt{res}}$ `v = {}`, so no execution reaches it. The first inclusion needs the run that built the report; the second and the dead case follow from the report alone, once its verdicts agree with its states.

The stores a report describes

```
definition report_gamma :: "analysis_report ⇒ mcp_val ⇒ store set" where
  "report_gamma res = mcp_gamma_v (activation (config_analyses (report_config res)))"

definition report_classify :: "analysis_report ⇒ exp ⇒ mcp_val ⇒ check_result" where
  "report_classify res = mcp_classify (activation (config_analyses (report_config res)))"
```

The states a report holds at a point, one per context it was solved at.

```
definition report_states_at :: "analysis_report ⇒ pp ⇒ mcp_val lifted set" where
  "report_states_at res v = state_value ` {st ∈ set (report_states res). state_point st = v}"

lemma finite_report_states_at [simp]: "finite (report_states_at res v)"
  by (simp add: report_states_at_def)
```

The stores one such state describes; `Bot` describes none.

```
abbreviation report_conc :: "analysis_report ⇒ mcp_val lifted ⇒ store set" ("γ_") where
  "γres ≡ gamma_lift (report_gamma res)"

definition report_sem :: "analysis_report ⇒ pp ⇒ store set" ("ℛ_") where
  "ℛres v = (⋃d ∈ report_states_at res v. γres d)"

lemma mem_report_sem [simp]:
  "s ∈ ℛres v ⟷ (∃d. Lifted d ∈ report_states_at res v ∧ s ∈ report_gamma res d)"
  unfolding report_sem_def
proof
  assume "s ∈ (⋃dl ∈ report_states_at res v. γres dl)"
  then obtain dl where "dl ∈ report_states_at res v" and "s ∈ γres dl"
    by blast
  then show "∃d. Lifted d ∈ report_states_at res v ∧ s ∈ report_gamma res d" by (cases dl) auto
qed (metis UN_I gamma_lift_Lifted)

lemma report_semI [intro]:
  "Lifted d ∈ report_states_at res v ⟹ s ∈ report_gamma res d ⟹ s ∈ ℛres v"
  unfolding report_sem_def by (rule UN_I[of "Lifted d"]) simp_all

lemma report_semE [elim]:
  assumes "s ∈ ℛres v"
  obtains d where "Lifted d ∈ report_states_at res v" and "s ∈ report_gamma res d"
proof -
  from assms
  obtain dl where "dl ∈ report_states_at res v" and "s ∈ γres dl"
    unfolding report_sem_def by blast
  then show ?thesis by (cases dl) (auto intro: that)
qed
```

The states a report holds at a point under one of its contexts: the rows at the point whose context index names that context.

```
definition report_states_in :: "analysis_report ⇒ pp ⇒ report_context ⇒ mcp_val lifted set"
where
  "report_states_in res v rc =
     state_value ` {st ∈ set (report_states res). state_point st = v
                     ∧ state_context st < length (report_contexts res)
                     ∧ report_contexts res ! state_context st = rc}"
```

The stores a report describes at a point under one context.

```
definition report_context_sem :: "analysis_report ⇒ pp ⇒ report_context ⇒ store set"
where
  "report_context_sem res v rc = (⋃d ∈ report_states_in res v rc. γres d)"

lemma report_states_in_subset: "report_states_in res v rc ⊆ report_states_at res v"
  unfolding report_states_in_def report_states_at_def by blast

lemma report_context_sem_subset: "report_context_sem res v rc ⊆ ℛres v"
  unfolding report_context_sem_def report_sem_def using report_states_in_subset by blast
```

One state per point

`report_point_join` joins the states a report holds at a point over its contexts. Every store the report describes at the point, the joined state describes, since each component's concretization is monotone.

```
lemma gamma_lift_state_mono: "x ≤ y ⟹ γ x ⊆ γ (y :: _ abs_state lifted)"
  by (rule gamma_lift_mono[where gam = gamma_state]) (use gamma_state_mono in blast)

lemma val_gamma_mono: "v ≤ v' ⟹ val_gamma a v ⊆ val_gamma a v'"
  by (cases "(a, v)" rule: val_gamma.cases)
    (auto simp: less_eq_analysis_product_def
      intro: gamma_lift_state_mono[THEN subsetD] gamma_relc_mono[THEN subsetD])

lemma mcp_gamma_v_mono: "v ≤ v' ⟹ mcp_gamma_v as v ⊆ mcp_gamma_v as v'"
  unfolding mcp_gamma_v_def using val_gamma_mono by blast

lemma fold_sup_ge: "x ∈ set xs ∨ x ≤ acc ⟹ x ≤ fold (⊔) xs (acc :: 'a::semilattice_sup)"
  by (induction xs arbitrary: acc) (auto intro: le_supI1 le_supI2)

lemma report_states_at_le_point_join: "d ∈ report_states_at res v ⟹ d ≤ report_point_join res v"
  unfolding report_states_at_def report_point_join_sup by (auto intro!: fold_sup_ge)
```

Every store the report describes at a point, the joined state describes.

```
theorem report_sem_point_join:
  "ℛres v ⊆ γres (report_point_join res v)"
proof
  fix s assume "s ∈ ℛres v"
  then obtain d where row: "Lifted d ∈ report_states_at res v" and s: "s ∈ report_gamma res d"
    by blast
  have "γres (Lifted d)
        ⊆ γres (report_point_join res v)"
    by (rule gamma_lift_mono[OF _ report_states_at_le_point_join[OF row]])
      (simp add: report_gamma_def mcp_gamma_v_mono)
  with s show "s ∈ γres (report_point_join res v)" by auto
qed
```

Verdicts

`report_checks_at` selects the checks a report records at one program point.

```
definition report_checks_at :: "analysis_report ⇒ pp ⇒ result_check set" where
  "report_checks_at res v = {c ∈ set (report_checks res). check_point c = v}"

lemma mem_report_checks_at [simp]:
  "c ∈ report_checks_at res v ⟷ c ∈ set (report_checks res) ∧ check_point c = v"
  by (simp add: report_checks_at_def)
```

What a decided verdict says about a store: `Check_Proved` that the condition holds, `Check_Refuted` that it fails, `Check_Unknown` nothing.

```
fun verdict_holds :: "check_result ⇒ exp ⇒ store ⇒ bool" where
  "verdict_holds Check_Proved e s = truthy ([[e]]e s)"
| "verdict_holds Check_Refuted e s = (¬ truthy ([[e]]e s))"
| "verdict_holds Check_Unknown e s = True"
```

The stores a check's verdict admits. A `Dead` check claims nothing about stores; its claim, that the point is unreached, is `DEAD` below.

```
definition check_stores :: "result_check ⇒ store set" where
  "check_stores c = (case check_verdict c of
                      Decided r ⇒ {s. verdict_holds r (check_exp c) s}
                    | Dead ⇒ UNIV)"

definition verdict_stores :: "analysis_report ⇒ pp ⇒ store set" ("𝒱_") where
```

```
  "𝒱res v = (⋂c ∈ report_checks_at res v. check_stores c)"

lemma verdict_stores_eq:
  "𝒱res v =
     {s. ∀c ∈ report_checks_at res v. ∀r.
           check_verdict c = Decided r ⟶ verdict_holds r (check_exp c) s}"
  unfolding verdict_stores_def check_stores_def by (auto split: lifted.splits)

definition HAS_VERDICT :: "analysis_report ⇒ pp ⇒ exp ⇒ check_result ⇒ bool" where
  "HAS_VERDICT res v e r ⟷
     (∃c ∈ report_checks_at res v. check_exp c = e ∧ check_verdict c = Decided r)"

abbreviation PROVED :: "analysis_report ⇒ pp ⇒ exp ⇒ bool" where
  "PROVED res v e ≡ HAS_VERDICT res v e Check_Proved"

abbreviation REFUTED :: "analysis_report ⇒ pp ⇒ exp ⇒ bool" where
  "REFUTED res v e ≡ HAS_VERDICT res v e Check_Refuted"

abbreviation UNKNOWN :: "analysis_report ⇒ pp ⇒ exp ⇒ bool" where
  "UNKNOWN res v e ≡ HAS_VERDICT res v e Check_Unknown"

lemma verdict_storesD:
  "s ∈ 𝒱res v ⟹ HAS_VERDICT res v e r ⟹ verdict_holds r e s"
  unfolding verdict_stores_eq HAS_VERDICT_def by blast
```

`DEAD` is a property of a point, not of a condition: every state the report holds there is `Bot`. The states are canonical — the table they come from collapses a state its emptiness test rejects to `Bot` — so `DEAD` reads that canonical form instead of running a test. The test is sound and incomplete for a product of analyses, so the converse of `analysis_report_dead` does not hold.

```
definition DEAD :: "analysis_report ⇒ pp ⇒ bool" where
  "DEAD res v ⟷ (∀d ∈ report_states_at res v. d = Bot)"
```

A consistent report

A report is consistent when each check's verdict is the aggregate of the report's own classifier over the report's own states at the check's point. Every report `run_voblint` returns is consistent.

```
definition consistent_report :: "analysis_report ⇒ bool" where
  "consistent_report res ⟷
     (∀c ∈ set (report_checks res).
        check_verdict c
          = aggregate_verdicts
              (classify_point (report_classify res) (check_exp c)
                 ‘ report_states_at res (check_point c)))"

lemma consistent_report_decided:
  assumes cons: "consistent_report res"
    and c: "c ∈ report_checks_at res v"
    and verdict: "check_verdict c = Decided r" and known: "r ≠ Check_Unknown"
    and row: "Lifted d ∈ report_states_at res v"
  shows "report_classify res (check_exp c) d = r"
proof -
  from c have mem: "c ∈ set (report_checks res)" and pt: "check_point c = v"
    by (simp_all add: report_checks_at_def)
  have agg: "aggregate_verdicts
               (classify_point (report_classify res) (check_exp c) ‘ report_states_at res v)
             = Decided r"
    using cons mem verdict pt unfolding consistent_report_def by auto
  from aggregate_verdicts_decided_dest[OF agg known] row
  show ?thesis by fastforce
qed
```

What a consistent report claims about its own states

A sound classifier's verdict holds at every store its abstract value admits: both directions of `sound_classifier` at once, which is what lets a report vouch for a `Check_Refuted` row as much as for a `Check_Proved` one.

```
lemma (in sound_classifier) classify_verdict_holds:
  "t ∈ gm d ⟹ verdict_holds (classify cnd d) cnd t"
  by (cases "classify cnd d") (auto dest: proved refuted)

theorem analysis_report_verdicts_sound:
  assumes "consistent_report res"
  shows "ℛres v ⊆ 𝒱res v"
proof
```

```
  fix s
  assume "s ∈ ℛres v"
  then obtain d where row: "Lifted d ∈ report_states_at res v" and s: "s ∈ report_gamma res d"
    by (rule report_semE)
  show "s ∈ 𝒱res v"
    unfolding verdict_stores_eq
  proof (intro CollectI ballI allI impI)
    fix c r
    assume c: "c ∈ report_checks_at res v" and verdict: "check_verdict c = Decided r"
    have holds: "verdict_holds (report_classify res (check_exp c) d) (check_exp c) s"
      using s unfolding report_classify_def report_gamma_def
      by (rule sound_classifier.classify_verdict_holds [OF mcp_sound_classifier])
    show "verdict_holds r (check_exp c) s"
    proof (cases "r = Check_Unknown")
      case False
      with consistent_report_decided[OF assms c verdict False row] holds show ?thesis by simp
    qed simp
  qed
qed

corollary analysis_report_proved:
  assumes "consistent_report res" and "PROVED res v e" and "s ∈ ℛres v"
  shows "truthy (⟦e⟧e s)"
  using verdict_storesD[OF subsetD[OF analysis_report_verdicts_sound[OF assms(1)] assms(3)]
    assms(2)]
  by simp

corollary analysis_report_refuted:
  assumes "consistent_report res" and "REFUTED res v e" and "s ∈ ℛres v"
  shows "¬ truthy (⟦e⟧e s)"
  using verdict_storesD[OF subsetD[OF analysis_report_verdicts_sound[OF assms(1)] assms(3)]
    assms(2)]
  by simp

theorem analysis_report_dead: "DEAD res v ⟹ ℛres v = {}"
  unfolding DEAD_def report_sem_def by auto
```

Read as an emptiness test on points, `DEAD` is sound.

```
corollary sound_emptiness_DEAD: "sound_emptiness (DEAD res) ℛres"
  by (rule sound_emptinessI) (rule analysis_report_dead)
```

It is not exact. A report holding one state at a point, the combined state `mcp_contradiction` of Interval and Parity, describes no store there, yet that state is not `Bot`, so the point is not `DEAD`. The witness is a report, not one a run is shown to return.

```
definition contradiction_report :: "pp ⇒ analysis_report" where
  "contradiction_report v =
     (| report_config = Analysis_Config [Interval_Analysis, Parity_Analysis] Globals_Join Ctx_None
          Program_Globals_Flow_Sensitive,
        report_vars = [], report_cfg = undefined, report_contexts = [Report_Unit],
        report_states = [(| state_point = v, state_context = 0, state_value = Lifted mcp_contradiction,
                            state_checks = [], state_diagnostics = [], state_steps = [] |)],
        report_routes = [], report_checks = [], report_globals = [], report_diagnostics = [] |)"

lemma DEAD_not_exact:
  "¬ exact_emptiness (DEAD (contradiction_report v)) ℛcontradiction_report v"
proof -
  have rows: "report_states_at (contradiction_report v) v = {Lifted mcp_contradiction}"
    by (auto simp: report_states_at_def contradiction_report_def)
  have live: "¬ DEAD (contradiction_report v) v"
    by (simp add: DEAD_def rows)
  have "report_config (contradiction_report v)
                  = Analysis_Config [Interval_Analysis, Parity_Analysis] Globals_Join Ctx_None
                      Program_Globals_Flow_Sensitive"
    by (simp add: contradiction_report_def)
  then have empty: "ℛcontradiction_report v v = {}"
    using mcp_contradiction_no_store by (auto simp: rows report_gamma_def activation_id)
  show ?thesis
  proof
    assume "exact_emptiness (DEAD (contradiction_report v)) ℛcontradiction_report v"
    from exact_emptinessD [OF this, of v] live empty show False by simp
  qed
qed
```

A check printed dead is one presentation of a dead point.

```
lemma analysis_report_check_dead:
```

```
  assumes "consistent_report res" and "c ∈ set (report_checks res)"
    and "check_verdict c = Dead"
  shows "DEAD res (check_point c)"
proof -
  have "aggregate_verdicts
          (classify_point (report_classify res) (check_exp c) ‘ report_states_at res (check_point c))
        = Dead"
    using assms unfolding consistent_report_def by auto
  then have "∀d ∈ report_states_at res (check_point c).
               classify_point (report_classify res) (check_exp c) d = Dead"
    by (simp add: aggregate_verdicts_eq_Dead_iff)
  then show ?thesis
    unfolding DEAD_def by (metis classify_point.elims lifted.distinct(1))
qed
```

An `UNKNOWN` check makes no claim about the stores, and its point is not dead: the report holds a state there that its classifier read.

```
lemma analysis_report_unknown:
  assumes "consistent_report res" and "UNKNOWN res v e"
  shows "¬ DEAD res v"
proof
  assume dead: "DEAD res v"
  from assms(2) obtain c where c: "c ∈ set (report_checks res)" "check_point c = v"
    and verdict: "check_verdict c = Decided Check_Unknown"
    unfolding HAS_VERDICT_def report_checks_at_def by blast
  have "∀d ∈ report_states_at res v. classify_point (report_classify res) (check_exp c) d = Dead"
    using dead unfolding DEAD_def by auto
  then have "aggregate_verdicts
               (classify_point (report_classify res) (check_exp c) ‘ report_states_at res v) = Dead"
    by (simp add: aggregate_verdicts_eq_Dead_iff)
  with assms(1) c verdict show False unfolding consistent_report_def by auto
qed
```

A report built from a sound table

`report_of` lists the contexts of the table in the order of an injective key, so it lists every one, and files every covered key as a row. Its rows at a point are therefore exactly the table's entries there.

```
lemma ordered_by_key_set:
  assumes fin: "finite S" and inj: "inj_on rank S"
  shows "set (ordered_by_key rank S) = S"
proof -
  have pick: "the_elem (Set.filter (λx. rank x = rank y) S) = y" if "y ∈ S" for y
  proof -
    have "Set.filter (λx. rank x = rank y) S = {y}"
      using that inj by (auto simp: inj_on_def)
    then show ?thesis by simp
  qed
  have "set (ordered_by_key rank S) = (λk. the_elem (Set.filter (λx. rank x = k) S)) ‘ rank ‘ S"
    unfolding ordered_by_key_def using fin by simp
  also have "... = S"
    using pick by (force simp: image_comp)
  finally show ?thesis .
qed

lemma row_in_rows:
  assumes "(i, ctx) ∈ set xs" and "v ∈ set ns" and "P v ctx"
  shows "f i ctx v ∈ set (concat (map (λ(i, ctx). map (f i ctx) (filter (λv. P v ctx) ns)) xs))"
  using assms by force

lemma report_states_at_report_of:
  assumes fin: "finite (covered_keys (run_table sr))" and inj: "inj ctx_key"
  shows "report_states_at (report_of config ctx_key ctx_tag classify sr p) v
           = lookup_table (run_table sr) v ‘ table_contexts (run_table sr) v"
proof -
  define r where "r = run_table sr"
  define ctxs where "ctxs = ordered_by_key ctx_key (snd ‘ covered_keys r)"
  define nodes where "nodes = cfg_node_list (prog_cfg p)
    @ sorted_list_of_set (fst ‘ covered_keys r - set (cfg_node_list (prog_cfg p)))"
  have ctxs: "set ctxs = snd ‘ covered_keys r"
    unfolding ctxs_def r_def
    by (rule ordered_by_key_set) (use fin inj in ‹auto intro: inj_on_subset›)
  have nodes: "w ∈ set nodes" if "(w, ctx) ∈ covered_keys r" for w ctx
  proof (cases "w ∈ set (cfg_node_list (prog_cfg p))")
    case False
```

```
    with that fin show ?thesis unfolding nodes_def r_def by (force intro: rev_image_eqI)
  qed (simp add: nodes_def)
  have indexed: "∃i. (i, ctx) ∈ set (enumerate 0 ctxs)" if "ctx ∈ set ctxs" for ctx
  proof -
    from that obtain i where "i < length ctxs" and "ctxs ! i = ctx"
      by (auto simp: in_set_conv_nth)
    then have "(i, ctx) ∈ set (enumerate 0 ctxs)"
      by (metis add_0 length_enumerate nth_enumerate_eq nth_mem)
    then show ?thesis ..
  qed
  show ?thesis
  proof (intro equalityI subsetI)
    fix d
    assume "d ∈ report_states_at (report_of config ctx_key ctx_tag classify sr p) v"
    then show "d ∈ lookup_table (run_table sr) v ‘ table_contexts (run_table sr) v"
      unfolding report_states_at_def report_of_def Let_def by (auto simp: table_contexts_iff)
  next
    fix d
    assume "d ∈ lookup_table (run_table sr) v ‘ table_contexts (run_table sr) v"
    then obtain ctx where cov: "(v, ctx) ∈ covered_keys r"
      and d: "d = lookup_table r v ctx"
      by (auto simp: r_def table_contexts_iff)
    from cov ctxs have "ctx ∈ set ctxs" by force
    then obtain i where i: "(i, ctx) ∈ set (enumerate 0 ctxs)" using indexed by blast
    have w: "v ∈ set nodes" using nodes[OF cov] .
    show "d ∈ report_states_at (report_of config ctx_key ctx_tag classify sr p) v"
      unfolding report_states_at_def report_of_def Let_def r_def[symmetric] ctxs_def[symmetric]
        nodes_def[symmetric]
      apply (simp only: analysis_report.select_convs)
      apply (rule image_eqI[rotated], rule CollectI, rule conjI, rule row_in_rows[OF i w])
      using cov d by (simp_all add: r_def)
  qed
qed
```

Each entry of the table is a row of the report under its own context: the context's index into the report's context list names it.

```
lemma report_states_in_report_of:
  assumes fin: "finite (covered_keys (run_table sr))" and inj: "inj ctx_key"
    and cov: "ctx ∈ table_contexts (run_table sr) v"
  shows "lookup_table (run_table sr) v ctx
          ∈ report_states_in (report_of config ctx_key ctx_tag classify sr p) v (ctx_tag ctx)"
proof -
  define r where "r = run_table sr"
  define ctxs where "ctxs = ordered_by_key ctx_key (snd ‘ covered_keys r)"
  define nodes where "nodes = cfg_node_list (prog_cfg p)
    @ sorted_list_of_set (fst ‘ covered_keys r - set (cfg_node_list (prog_cfg p)))"
  have ctxs: "set ctxs = snd ‘ covered_keys r"
    unfolding ctxs_def r_def
    by (rule ordered_by_key_set) (use fin inj in ‹auto intro: inj_on_subset›)
  from cov have covk: "(v, ctx) ∈ covered_keys r" by (simp add: r_def)
  have w: "v ∈ set nodes"
  proof (cases "v ∈ set (cfg_node_list (prog_cfg p))")
    case False
    with covk fin show ?thesis unfolding nodes_def r_def by (force intro: rev_image_eqI)
  qed (simp add: nodes_def)
  from covk ctxs have "ctx ∈ set ctxs" by force
  then obtain i where "i < length ctxs" and nth: "ctxs ! i = ctx"
    by (auto simp: in_set_conv_nth)
  then have i: "(i, ctx) ∈ set (enumerate 0 ctxs)" and len: "i < length ctxs"
    by (auto simp: in_set_enumerate_eq)
  show ?thesis
    unfolding report_states_in_def report_of_def Let_def r_def[symmetric] ctxs_def[symmetric]
      nodes_def[symmetric]
    apply (simp only: analysis_report.select_convs)
    apply (rule image_eqI[rotated], rule CollectI, rule conjI, rule row_in_rows[OF i w])
    using covk len nth by (simp_all add: report_row_def Let_def r_def)
qed

lemma report_of_simps [simp]:
  "report_config (report_of config ctx_key ctx_tag classify sr p) = config"
  "report_cfg (report_of config ctx_key ctx_tag classify sr p) = prog_cfg p"
  "report_checks (report_of config ctx_key ctx_tag classify sr p)
     = result_checks_of (prog_cfg p) (run_table sr) classify"
  "report_diagnostics (report_of config ctx_key ctx_tag classify sr p)
     = arithmetic_diagnostics (prog_cfg p) (run_table sr) classify"
  by (simp_all add: report_of_def Let_def)
```

What a report built from a run of a program owes: its verdicts agree with its states, its states cover the collecting semantics, its diagnostics name every point where a divisor may be zero, and its check column lists the program's checks.

```
definition sound_report :: "imp_prog ⇒ analysis_report ⇒ bool" where
  "sound_report p res ⟷
     consistent_report res
   ∧ (∀v. 𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v ⊆ ℛres v)
   ∧ (∀v s. s ∈ 𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v
        ⟶ (∀d ∈ set (report_diagnostics res). diagnostic_point d ≠ v)
        ⟶ arithmetic_safe_at (prog_cfg p) v s)
   ∧ (∀d ∈ set (report_diagnostics res). diagnostic_verdict d = Check_Refuted
        ⟶ (∀s ∈ ℛres (diagnostic_point d).
             ¬ truthy (⟦arithmetic_condition (diagnostic_obligation d)⟧e s)))
   ∧ map (λc. (check_point c, check_label c, check_exp c)) (report_checks res)
       = check_sites (prog_cfg p)"
```

Contexts under one name

Each context policy has a context type of its own, and the report files every context under one type, `report_context`, through an injective tag. Renaming a policy's contexts through the tag keeps which contexts a trace carries, so the activation buckets survive the renaming unchanged.

```
definition map_context_policy :: "('c ⇒ 'd) ⇒ 'c context_policy ⇒ 'd context_policy" where
  "map_context_policy f adm u d ci s es = (⋃c ∈ f -‘ {d}. f ‘ adm u c ci s es)"

lemma admits_call_context_map_policy:
  assumes inj: "inj f"
  shows "admits_call_context 𝒢 g (map_context_policy f adm) u (f c) q s es d'
         ⟷ (∃c'. d' = f c' ∧ admits_call_context 𝒢 g adm u c q s es c')"
  unfolding admits_call_context_def map_context_policy_def
  using inj by (auto simp: inj_eq) blast+

lemma activation_context_rel_map_policy:
  assumes inj: "inj f"
  shows "activation_context_rel 𝒢 (map_context_policy f adm) (f c0) g t d
         ⟷ (∃c. d = f c ∧ activation_context_rel 𝒢 adm c0 g t c)"
proof
  assume "activation_context_rel 𝒢 (map_context_policy f adm) (f c0) g t d"
  then show "∃c. d = f c ∧ activation_context_rel 𝒢 adm c0 g t c"
  proof induction
    case (Root xs)
    then show ?case by blast
  next
    case (Call caller x p es x' xs)
    then obtain c where c: "x = f c" "activation_context_rel 𝒢 adm c0 g caller c"
      by blast
    with Call.hyps(2) obtain c' where "x' = f c'"
      and "admits_call_context 𝒢 g adm (sink_node caller) c p (sink_store caller) es c'"
      by (auto simp: admits_call_context_map_policy [OF inj])
    with c(2) show ?case by blast
  next
    case (Resume current x callee xs)
    then show ?case by blast
  qed
next
  assume "∃c. d = f c ∧ activation_context_rel 𝒢 adm c0 g t c"
  then obtain c where d: "d = f c" and rel: "activation_context_rel 𝒢 adm c0 g t c"
    by blast
  from rel show "activation_context_rel 𝒢 (map_context_policy f adm) (f c0) g t d"
    unfolding d
  proof induction
    case (Root xs)
    then show ?case by blast
  next
    case (Call caller x p es x' xs)
    then show ?case
      by (blast intro: admits_call_context_map_policy [OF inj, THEN iffD2])
  next
    case (Resume current x callee xs)
    then show ?case by blast
  qed
qed
```

What a report guarantees per context, for a policy over report contexts: every valid activation trace carries

one of the policy's contexts, and each context's activation bucket lies in the stores the report's states for that context describe. A context the report holds no state for at a point has an empty bucket there.

```
definition context_sound ::
    "imp_prog ⇒ analysis_report ⇒ report_context context_policy ⇒ report_context ⇒ bool"
where
  "context_sound p res adm rc ⟷
     (∀t ∈ 𝒯declared_global p,prog_cfg p,cinit_stores (declared_global p)·
        ∃c. activation_context_rel (declared_global p) adm rc (prog_cfg p) t c)
   ∧ (∀v c. 𝒜declared_global p,adm,rc,prog_cfg p,cinit_stores (declared_global p) v c
         ⊆ report_context_sem res v c)"

lemma context_sound_point:
  "context_sound p res adm rc
   ⟹ 𝒜declared_global p,adm,rc,prog_cfg p,cinit_stores (declared_global p) v c
        ⊆ ℛres v"
  unfolding context_sound_def using report_context_sem_subset by blast
```

The context policy a configuration runs with, under its report contexts, and the context of `main`. The entry-state policy is read off the program's own solve.

```
definition run_policy :: "analysis_config ⇒ imp_prog ⇒ report_context context_policy" where
  "run_policy config p =
     (case config_context config of
        Ctx_None ⇒
          map_context_policy (λ_ :: unit. Report_Unit) (context_policy_of_fun (λu c t. ()))
      | Ctx_CallString k ⇒
          map_context_policy Report_Call_String
            (context_policy_of_fun (λu ctx t. cs_context k u ctx t))
      | Ctx_EntryState ⇒
          map_context_policy Report_Entry
            (case config_globals config of
               Program_Globals_Flow_Sensitive ⇒
                 mcp_es_rule.admitted_contexts (config_analyses config) (config_rule config)
                   (declared_global p) p
             | Program_Globals_Flow_Insensitive ⇒
                 mcp_split_es_rule.admitted_contexts (config_analyses config) (config_rule config)
                   (declared_global p) p))"

definition run_root :: "analysis_config ⇒ report_context" where
  "run_root config =
     (case config_context config of
        Ctx_None ⇒ Report_Unit
      | Ctx_CallString k ⇒ Report_Call_String []
      | Ctx_EntryState ⇒ Report_Entry mcp_root_ctx)"

lemma report_of_sound:
  assumes act: "activation_table p (run_table sr) gm adm rc"
    and fin: "finite (covered_keys (run_table sr))" and inj: "inj ctx_key"
    and tag: "inj ctx_tag"
    and cl: "classify = mcp_classify (activation (config_analyses config))"
    and gm: "gm = mcp_gamma_v (activation (config_analyses config))"
  shows "sound_report p (report_of config ctx_key ctx_tag classify sr p)
       ∧ context_sound p (report_of config ctx_key ctx_tag classify sr p)
           (map_context_policy ctx_tag adm) (ctx_tag rc)"
proof -
  note cov = activation_table.covered [OF act]
  have st: "sound_table p (run_table sr) classify gm"
    by (rule sound_table.intro [OF cov]) (unfold cl gm, rule mcp_sound_classifier)
  let ?res = "report_of config ctx_key ctx_tag classify sr p"
  note rows = report_states_at_report_of[OF fin inj]
  have cons: "consistent_report ?res"
    unfolding consistent_report_def
    by (auto simp: result_checks_of_def point_verdict_def rows report_classify_def cl image_comp
        comp_def)
  have covers: "s ∈ ℛ?res v"
    if "s ∈ 𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v" for v s
  proof -
    from covered_table.covers[OF cov that] obtain ctx d
      where look: "lookup_table (run_table sr) v ctx = Lifted d" and s: "s ∈ gm d"
      unfolding table_covers_def by blast
    then have "Lifted d ∈ report_states_at ?res v"
      unfolding rows by (metis image_eqI lookup_table_LiftedD)
    with s show ?thesis by (auto simp: report_gamma_def gm)
  qed
  have safe: "arithmetic_safe_at (prog_cfg p) v s"
```

```
    if "s ∈ 𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v"
      and "∀d ∈ set (report_diagnostics ?res). diagnostic_point d ≠ v" for v s
    using sound_table.arithmetic_safe[OF st _ that(1)] that(2) by simp
  have refuted: "¬ truthy (⟦arithmetic_condition (diagnostic_obligation d)⟧e s)"
    if d: "d ∈ set (report_diagnostics ?res)" and r: "diagnostic_verdict d = Check_Refuted"
      and s: "s ∈ ℛ?res (diagnostic_point d)" for d s
  proof -
    let ?e = "arithmetic_condition (diagnostic_obligation d)"
    have pv: "point_verdict (run_table sr) classify (diagnostic_point d) ?e = Decided Check_Refuted"
      using arithmetic_diagnostics_refuted d r by simp
    from s obtain st where row: "Lifted st ∈ report_states_at ?res (diagnostic_point d)"
      and sg: "s ∈ report_gamma ?res st"
      by (rule report_semE)
    then obtain ctx where "lookup_table (run_table sr) (diagnostic_point d) ctx = Lifted st"
      unfolding rows by force
    then have "classify ?e st = Check_Refuted"
      by (rule point_verdict_decided[OF pv, rotated]) simp
    with sg show ?thesis
      unfolding cl report_gamma_def report_of_simps by (blast dest: mcp_classify_refuted)
  qed
  have sr: "sound_report p ?res"
    unfolding sound_report_def
    using cons covers safe refuted by (auto simp: result_checks_of_sites)
  have ctx: "context_sound p ?res (map_context_policy ctx_tag adm) (ctx_tag rc)"
    unfolding context_sound_def
  proof (intro conjI ballI allI subsetI)
    fix t
    assume t: "t ∈ 𝒯declared_global p,prog_cfg p,cinit_stores (declared_global p)"
    from activation_table.has_context [OF act t]
    show "∃c. activation_context_rel (declared_global p) (map_context_policy ctx_tag adm)
            (ctx_tag rc) (prog_cfg p) t c"
      by (auto simp: activation_context_rel_map_policy [OF tag])
  next
    fix v d s
    assume "s ∈ 𝒜declared_global p,map_context_policy ctx_tag adm,ctx_tag rc,prog_cfg p,                    cinit_stores (dec
v d"
    then obtain t where t: "t ∈ 𝒯declared_global p,prog_cfg p,cinit_stores (declared_global p)"
        "sink_node t = v" "sink_store t = s"
      and rel: "activation_context_rel (declared_global p) (map_context_policy ctx_tag adm)
                (ctx_tag rc) (prog_cfg p) t d"
      unfolding activation_collect_def by blast
    from rel obtain c where d: "d = ctx_tag c"
      and "activation_context_rel (declared_global p) adm rc (prog_cfg p) t c"
      by (auto simp: activation_context_rel_map_policy [OF tag])
    with t have "s ∈ 𝒜declared_global p,adm,rc,prog_cfg p,cinit_stores (declared_global p) v c"
      unfolding activation_collect_def by blast
    with activation_table.bucket_bound [OF act]
    have "s ∈ gamma_lift gm (lookup_table (run_table sr) v c)" by blast
    then obtain st where look: "lookup_table (run_table sr) v c = Lifted st" and s: "s ∈ gm st"
      by (cases "lookup_table (run_table sr) v c") auto
    then have "Lifted st ∈ report_states_in ?res v d"
      unfolding d using report_states_in_report_of[OF fin inj lookup_table_LiftedD[OF look]]
      by simp
    with s show "s ∈ report_context_sem ?res v d"
      unfolding report_context_sem_def by (auto simp: report_gamma_def gm intro!: bexI)
  qed
  show ?thesis using sr ctx by blast
qed
```

The one argument that reads the context policy. Each policy's report comes from its registration's executable solve, whose answer is the solve the registration's soundness theorem is about; the rest is `report_of_sound`.

```
lemma inj_call_string_key: "inj (λctx. Key_List (map Key_Node ctx))"
  by (rule injI) (simp add: inj_def)

lemma inj_entry_ctx_key: "inj (entry_ctx_key as)"
  by (rule injI) (rule entry_ctx_key_inject)

lemma analysis_report_of_sound:
  assumes wf: "wf_program_compile_input p"
    and some: "analysis_report_of config p = Some res"
  shows "sound_report p res ∧ context_sound p res (run_policy config p) (run_root config)"
proof (cases config)
  case (Analysis_Config as r ctx pg)
  show ?thesis
  proof (cases ctx)
```

```
  case Ctx_None
  show ?thesis
  proof (cases pg)
    case Program_Globals_Flow_Sensitive
    from some obtain sol where
      sol: "TD_side_rule_Interp_solve_c r (mcp_rule.equations as (declared_global p) p)
            (mcp_rule.root_query p) = Some sol"
      and res: "res = report_of config (λ_. Key_List []) (λ_. Report_Unit)
                  (mcp_classify (activation as))
                  (mcp_rule.solved_run_of as (declared_global p) p sol) p"
      by (auto simp: Analysis_Config Ctx_None Program_Globals_Flow_Sensitive
        dg_pipeline.root_query_def
          mcp_wrappers mcp_place_defs)
    note run = mcp_rule.solve_c_run[OF sol]
    have pol: "run_policy config p
               = map_context_policy (λ_ :: unit. Report_Unit) (context_policy_of_fun (λu c t. ()))"
      "run_root config = Report_Unit"
      by (simp_all add: run_policy_def run_root_def Analysis_Config Ctx_None)
    show ?thesis
      unfolding res run(2) pol
      by (rule report_of_sound[where rc = "()", OF _ _ _ _ _ refl])
         (use mcp_rule_table[OF wf run(1)] mcp_rule.vars_finite_of_terminates[OF run(1)] in
           ‹simp_all add: Analysis_Config finite_solved_table_def dg_pipeline.result_def
              dg_pipeline.sol_vars_def inj_def›)
  next
    case Program_Globals_Flow_Insensitive
    from some obtain sol where
      sol: "TD_side_rule_Interp_solve_c r (mcp_split_rule.equations as (declared_global p) p)
            (mcp_split_rule.root_query p) = Some sol"
      and res: "res = report_of config (λ_. Key_List []) (λ_. Report_Unit)
                  (mcp_classify (activation as))
                  (mcp_split_rule.solved_run_of as (declared_global p) p sol) p"
      by (auto simp: Analysis_Config Ctx_None Program_Globals_Flow_Insensitive
        dg_pipeline.root_query_def
          mcp_wrappers mcp_place_defs)
    note run = mcp_split_rule.solve_c_run[OF sol]
    have pol: "run_policy config p
               = map_context_policy (λ_ :: unit. Report_Unit) (context_policy_of_fun (λu c t. ()))"
      "run_root config = Report_Unit"
      by (simp_all add: run_policy_def run_root_def Analysis_Config Ctx_None)
    show ?thesis
      unfolding res run(2) pol
      by (rule report_of_sound[where rc = "()", OF _ _ _ _ _ refl])
         (use mcp_split_rule_table[OF wf run(1)]
            mcp_split_rule.vars_finite_of_terminates[OF run(1)] in
           ‹simp_all add: Analysis_Config finite_solved_table_def dg_pipeline.result_def
              dg_pipeline.sol_vars_def inj_def›)
  qed
next
  case Ctx_EntryState
  show ?thesis
  proof (cases pg)
    case Program_Globals_Flow_Sensitive
    from some obtain sol where
      sol: "TD_side_rule_Interp_solve_c r (mcp_es_rule.equations as (declared_global p) p)
            (mcp_es_rule.root_query p) = Some sol"
      and res: "res = report_of config (entry_ctx_key (activation as)) Report_Entry
                  (mcp_classify (activation as))
                  (mcp_es_rule.solved_run_of as (declared_global p) p sol) p"
      by (auto simp: Analysis_Config Ctx_EntryState Program_Globals_Flow_Sensitive
          dg_pipeline.root_query_def mcp_wrappers mcp_place_defs)
    note run = mcp_es_rule.solve_c_run[OF sol]
    have pol: "run_policy config p
               = map_context_policy Report_Entry
                   (mcp_es_rule.admitted_contexts as r (declared_global p) p)"
      "run_root config = Report_Entry mcp_root_ctx"
      by (simp_all add: run_policy_def run_root_def Analysis_Config Ctx_EntryState
          Program_Globals_Flow_Sensitive)
    show ?thesis
      unfolding res run(2) pol
      by (rule report_of_sound[OF _ _ inj_entry_ctx_key _ _ refl])
         (use mcp_es_rule_table[OF wf run(1)] mcp_es_rule.vars_finite_of_terminates[OF run(1)] in
           ‹simp_all add: Analysis_Config finite_solved_table_def dg_pipeline.result_def
              dg_pipeline.sol_vars_def inj_def›)
  next
    case Program_Globals_Flow_Insensitive
```

```
      from some obtain sol where
        sol: "TD_side_rule_Interp_solve_c r (mcp_split_es_rule.equations as (declared_global p) p)
                (mcp_split_es_rule.root_query p) = Some sol"
        and res: "res = report_of config (entry_ctx_key (activation as)) Report_Entry
                    (mcp_classify (activation as))
                    (mcp_split_es_rule.solved_run_of as (declared_global p) p sol) p"
        by (auto simp: Analysis_Config Ctx_EntryState Program_Globals_Flow_Insensitive
            dg_pipeline.root_query_def mcp_wrappers mcp_place_defs)
      note run = mcp_split_es_rule.solve_c_run[OF sol]
      have pol: "run_policy config p
                  = map_context_policy Report_Entry
                      (mcp_split_es_rule.admitted_contexts as r (declared_global p) p)"
        "run_root config = Report_Entry mcp_root_ctx"
        by (simp_all add: run_policy_def run_root_def Analysis_Config Ctx_EntryState
            Program_Globals_Flow_Insensitive)
      show ?thesis
        unfolding res run(2) pol
        by (rule report_of_sound[OF _ _ inj_entry_ctx_key _ _ refl])
           (use mcp_split_es_rule_table[OF wf run(1)]
              mcp_split_es_rule.vars_finite_of_terminates[OF run(1)] in
             ‹simp_all add: Analysis_Config finite_solved_table_def dg_pipeline.result_def
                dg_pipeline.sol_vars_def inj_def›)
    qed
  next
    case (Ctx_CallString k)
    show ?thesis
    proof (cases pg)
      case Program_Globals_Flow_Sensitive
      from some obtain sol where
        sol: "TD_side_rule_Interp_solve_c r (mcp_cs_rule.equations as k (declared_global p) p)
                (mcp_cs_rule.root_query p) = Some sol"
        and res: "res = report_of config (λctx. Key_List (map Key_Node ctx)) Report_Call_String
                    (mcp_classify (activation as))
                    (mcp_cs_rule.solved_run_of as k (declared_global p) p sol) p"
        by (auto simp: Analysis_Config Ctx_CallString Program_Globals_Flow_Sensitive
            dg_pipeline.root_query_def mcp_wrappers mcp_place_defs)
      note run = mcp_cs_rule.solve_c_run[OF sol]
      have pol: "run_policy config p
                  = map_context_policy Report_Call_String
                      (context_policy_of_fun (λu ctx t. cs_context k u ctx t))"
        "run_root config = Report_Call_String []"
        by (simp_all add: run_policy_def run_root_def Analysis_Config Ctx_CallString)
      show ?thesis
        unfolding res run(2) pol
        by (rule report_of_sound[OF _ _ inj_call_string_key _ _ refl])
           (use mcp_cs_rule_table[OF wf run(1)] mcp_cs_rule.vars_finite_of_terminates[OF run(1)] in
             ‹simp_all add: Analysis_Config finite_solved_table_def dg_pipeline.result_def
                dg_pipeline.sol_vars_def inj_def›)
    next
      case Program_Globals_Flow_Insensitive
      from some obtain sol where
        sol: "TD_side_rule_Interp_solve_c r (mcp_split_cs_rule.equations as k (declared_global p) p)
                (mcp_split_cs_rule.root_query p) = Some sol"
        and res: "res = report_of config (λctx. Key_List (map Key_Node ctx)) Report_Call_String
                    (mcp_classify (activation as))
                    (mcp_split_cs_rule.solved_run_of as k (declared_global p) p sol) p"
        by (auto simp: Analysis_Config Ctx_CallString Program_Globals_Flow_Insensitive
            dg_pipeline.root_query_def mcp_wrappers mcp_place_defs)
      note run = mcp_split_cs_rule.solve_c_run[OF sol]
      have pol: "run_policy config p
                  = map_context_policy Report_Call_String
                      (context_policy_of_fun (λu ctx t. cs_context k u ctx t))"
        "run_root config = Report_Call_String []"
        by (simp_all add: run_policy_def run_root_def Analysis_Config Ctx_CallString)
      show ?thesis
        unfolding res run(2) pol
        by (rule report_of_sound[OF _ _ inj_call_string_key _ _ refl])
           (use mcp_split_cs_rule_table[OF wf run(1)]
              mcp_split_cs_rule.vars_finite_of_terminates[OF run(1)] in
             ‹simp_all add: Analysis_Config finite_solved_table_def dg_pipeline.result_def
                dg_pipeline.sol_vars_def inj_def›)
    qed
  qed
qed
```

A well-formed report

What a reader of the report relies on beyond its verdicts: every context index names a listed context, a point has at most one row per context, and a row's check and obligation columns list exactly the report's checks and the graph's arithmetic obligations at the row's point, each with the verdict of the row's own state. No soundness theorem needs this; a renderer does.

```
definition obligations_at :: "cfg ⇒ pp ⇒ arithmetic_obligation list" where
  "obligations_at g v = concat (map snd (filter (λ(u, obs). u = v) (arithmetic_sites g)))"

lemma group_lookup_arithmetic_sites:
  "concat (group_lookup (group_by_key fst (Some ∘ snd) (arithmetic_sites g)) v)
     = obligations_at g v"
proof -
  have "List.map_filter (λx. if fst x = v then (Some ∘ snd) x else None) xs
         = map snd (filter (λ(u, obs). u = v) xs)"
    for xs :: "(pp × arithmetic_obligation list) list"
    by (induction xs) (auto simp: List.map_filter_simps)
  then show ?thesis unfolding group_lookup_group_by_key obligations_at_def by simp
qed

definition well_formed_report :: "analysis_report ⇒ bool" where
  "well_formed_report res ⟷
     (∀st ∈ set (report_states res). state_context st < length (report_contexts res))
   ∧ distinct (map (λst. (state_point st, state_context st)) (report_states res))
   ∧ (∀rt ∈ set (report_routes res).
       route_context rt < length (report_contexts res)
     ∧ (∀i ∈ set (route_targets rt). i < length (report_contexts res)))
   ∧ (∀st ∈ set (report_states res).
       state_checks st
         = map (λc. (check_exp c,
                     classify_point (report_classify res) (check_exp c) (state_value st)))
             (filter (λc. check_point c = state_point st) (report_checks res))
     ∧ state_diagnostics st
         = map (λob. (ob,
                      classify_point (report_classify res) (arithmetic_condition ob)
                        (state_value st)))
             (obligations_at (report_cfg res) (state_point st)))"
```

With a consistent report, the check column is the aggregate of the rows: a check's verdict joins the verdicts the rows at its point give its condition.

```
lemma well_formed_check_verdict:
  assumes cons: "consistent_report res" and wf: "well_formed_report res"
    and c: "c ∈ set (report_checks res)"
  shows "check_verdict c
          = aggregate_verdicts
              {verdict | st verdict. st ∈ set (report_states res)
                 ∧ state_point st = check_point c ∧ (check_exp c, verdict) ∈ set (state_checks st)}"
proof -
  let ?cl = "classify_point (report_classify res) (check_exp c)"
  have rows: "state_checks st
                = map (λc. (check_exp c, classify_point (report_classify res) (check_exp c)
                                           (state_value st)))
                    (filter (λc'. check_point c' = state_point st) (report_checks res))"
    if "st ∈ set (report_states res)" for st
    using wf that unfolding well_formed_report_def by blast
  have "{verdict | st verdict. st ∈ set (report_states res)
           ∧ state_point st = check_point c ∧ (check_exp c, verdict) ∈ set (state_checks st)}
         = ?cl ‘ report_states_at res (check_point c)"
  proof (intro equalityI subsetI)
    fix x
    assume "x ∈ {verdict | st verdict. st ∈ set (report_states res)
              ∧ state_point st = check_point c ∧ (check_exp c, verdict) ∈ set (state_checks st)}"
    then obtain st where st: "st ∈ set (report_states res)" and pt: "state_point st = check_point c"
      and mem: "(check_exp c, x) ∈ set (state_checks st)" by blast
    from mem rows[OF st] have "x = ?cl (state_value st)" by auto
    with st pt show "x ∈ ?cl ‘ report_states_at res (check_point c)"
      unfolding report_states_at_def by blast
  next
    fix x
    assume "x ∈ ?cl ‘ report_states_at res (check_point c)"
    then obtain st where st: "st ∈ set (report_states res)" and pt: "state_point st = check_point c"
      and x: "x = ?cl (state_value st)"
      unfolding report_states_at_def by blast
    have "(check_exp c, x) ∈ set (state_checks st)"
```

```
      unfolding rows[OF st] using c pt x by force
    with st pt show "x ∈ {verdict | st verdict. st ∈ set (report_states res)
             ∧ state_point st = check_point c ∧ (check_exp c, verdict) ∈ set (state_checks st)}"
      by blast
  qed
  with cons c show ?thesis unfolding consistent_report_def by simp
qed

lemma distinct_indexed_rows:
  "distinct ns
     ⟹ distinct (concat (map (λ(i, x). map (λv. (v, i)) (filter (λv. P v x) ns))
                         (enumerate n xs)))"
proof (induction xs arbitrary: n)
  case (Cons x xs)
  have "n < i" if "(w, i) ∈ set (concat (map (λ(i, x). map (λv. (v, i)) (filter (λv. P v x) ns))
                                 (enumerate (Suc n) xs)))" for w i
    using that by (auto simp: in_set_enumerate_eq)
  with Cons.IH[of "Suc n"] Cons.prems show ?case
    by (fastforce simp: distinct_map inj_on_def)
qed simp

lemma map_check_exp_result_checks_of:
  "map check_exp (filter (λc. check_point c = v) (result_checks_of g r classify))
     = group_lookup (group_by_key (λ(u, a, w). u)
         (λ(u, a, w). if is_EA_Check a then Some (ea_check_cond a) else None) (cfg_intra_list g)) v"
proof -
  have "map check_exp (filter (λc. check_point c = v)
          (map (λ(u, a, w). ⦇ check_point = u, check_label = ea_check_label a,
                              check_exp = ea_check_cond a, check_verdict = V u a ⦈)
             (filter (λ(u, a, w). is_EA_Check a) es)))
        = List.map_filter (λx. if (λ(u, a, w). u) x = v
                               then (λ(u, a, w). if is_EA_Check a then Some (ea_check_cond a)
                                                 else None) x
                               else None) es" for es V
    by (induction es) (auto simp: List.map_filter_simps)
  then show ?thesis
    unfolding result_checks_of_def group_lookup_group_by_key .
qed

lemma report_of_well_formed:
  assumes cl: "classify = mcp_classify (activation (config_analyses config))"
  shows "well_formed_report (report_of config ctx_key ctx_tag classify sr p)"
proof -
  let ?res = "report_of config ctx_key ctx_tag classify sr p"
  define r where "r = run_table sr"
  define g where "g = prog_cfg p"
  define ctxs where "ctxs = ordered_by_key ctx_key (snd ‘ covered_keys r)"
  define nodes where "nodes = cfg_node_list g
    @ sorted_list_of_set (fst ‘ covered_keys r - set (cfg_node_list g))"
  define checks where "checks = group_by_key (λ(u, a, w). u)
    (λ(u, a, w). if is_EA_Check a then Some (ea_check_cond a) else None) (cfg_intra_list g)"
  define obls where "obls = group_by_key fst (Some ∘ snd) (arithmetic_sites g)"
  define steps where "steps = group_by_key (λ(u, a, w). u) (λ(u, a, w). Some (a, w))
    (cfg_intra_list g)"
  have classify: "report_classify ?res = classify"
    by (simp add: report_classify_def cl)
  have contexts: "length (report_contexts ?res) = length ctxs"
    by (simp add: report_of_def Let_def ctxs_def r_def)
  have states: "report_states ?res
    = concat (map (λ(i, ctx). map (report_row sr classify checks obls steps i ctx)
                                (filter (λv. (v, ctx) ∈ covered_keys r) nodes))
              (enumerate 0 ctxs))"
    by (simp add: report_of_def Let_def ctxs_def r_def nodes_def g_def checks_def obls_def
                  steps_def)
  have in_range: "state_context st < length ctxs" if "st ∈ set (report_states ?res)" for st
    using that by (auto simp: states in_set_enumerate_eq)
  have "distinct (cfg_node_list g)" by (simp add: cfg_node_list_def)
  then have "distinct nodes"
    unfolding nodes_def by (cases "finite (fst ‘ covered_keys r - set (cfg_node_list g))") auto
  then have distinct:
    "distinct (map (λst. (state_point st, state_context st)) (report_states ?res))"
    using distinct_indexed_rows[of nodes "λv ctx. (v, ctx) ∈ covered_keys r" 0 ctxs]
    by (simp add: states map_concat case_prod_unfold comp_def)
  have routes: "route_context rt < length ctxs ∧ (∀i ∈ set (route_targets rt). i < length ctxs)"
    if "rt ∈ set (report_routes ?res)" for rt
    using that
```

```
    by (auto simp: report_of_def Let_def ctxs_def r_def context_indices_def in_set_enumerate_eq
             split: lifted.splits)
  have rows: "state_checks st
                = map (λc. (check_exp c, classify_point classify (check_exp c) (state_value st)))
                    (filter (λc. check_point c = state_point st) (report_checks ?res))
              ∧ state_diagnostics st
                  = map (λob. (ob, classify_point classify (arithmetic_condition ob)
                                     (state_value st)))
                      (obligations_at g (state_point st))"
    if "st ∈ set (report_states ?res)" for st
  proof -
    from that obtain i ctx v where st: "st = report_row sr classify checks obls steps i ctx v"
      by (auto simp: states)
    have "map check_exp (filter (λc. check_point c = v) (report_checks ?res))
            = group_lookup checks v" (is "map check_exp ?cs = _")
      by (simp add: map_check_exp_result_checks_of checks_def g_def)
    moreover have "map (λc. (check_exp c, classify_point classify (check_exp c) (lookup_table r v ctx)))
                     ?cs
                   = map (λcond. (cond, classify_point classify cond (lookup_table r v ctx)))
                       (map check_exp ?cs)"
      by simp
    ultimately have "map (λc. (check_exp c, classify_point classify (check_exp c) (lookup_table r v ctx)))
                       ?cs
                     = map (λcond. (cond, classify_point classify cond (lookup_table r v ctx)))
                         (group_lookup checks v)"
      by simp
    moreover have "concat (group_lookup obls v) = obligations_at g v"
      unfolding obls_def by (rule group_lookup_arithmetic_sites)
    ultimately show ?thesis unfolding st by (auto simp: report_row_def Let_def r_def)
  qed
  have cfg: "report_cfg ?res = g" by (simp add: g_def)
  show ?thesis
    unfolding well_formed_report_def classify contexts cfg
    using in_range distinct routes rows by blast
qed

lemma analysis_report_of_well_formed:
  "analysis_report_of config p = Some res ⟹ well_formed_report res"
  by (induct config p rule: analysis_report_of.induct) (auto intro: report_of_well_formed)

end
```

17.12 Analysis_Certified

```
theory Analysis_Certified
  imports Analysis_Report "Voblint_Compile.Elaborated_Compilation"
begin
```

What an analysis report guarantees about the program

`run_voblint` answers every configuration. Its analysed answer is a report that the theorems below read through $\mathcal{R}_{\texttt{res}}$ `v`, `verdict_stores` and the CAPS queries, and nothing else: no premise asks for termination, coverage of the solve or well-formedness, since an analysed answer was only given for a well-formed program whose solve the executable solver completed.

An analysed answer is a sound report

```
lemma run_voblint_AnalysedE [elim]:
  assumes "run_voblint config p = Analysed res"
  obtains "valid_config config" and "wf_program_compile_input_exec p"
    and "analysis_report_of config p = Some res"
  using assms unfolding run_voblint_def by (auto split: if_splits option.splits)

lemma analysis_report_of_fields:
  "analysis_report_of config p = Some res
     ⟹ report_config res = config ∧ report_cfg res = prog_cfg p"
  by (induct config p rule: analysis_report_of.induct) auto

lemma run_voblint_config:
  "run_voblint config p = Analysed res ⟹ report_config res = config"
  by (auto dest: analysis_report_of_fields)

lemma run_voblint_cfg:
  "run_voblint config p = Analysed res ⟹ report_cfg res = prog_cfg p"
  by (auto dest: analysis_report_of_fields)

lemma run_voblint_report_sound:
  assumes "run_voblint config p = Analysed res"
  shows "sound_report p res ∧ context_sound p res (run_policy config p) (run_root config)"
  using assms
  by (elim run_voblint_AnalysedE)
     (rule analysis_report_of_sound [OF wf_program_compile_input_exec_sound])

theorem run_voblint_sound:
  assumes "run_voblint config p = Analysed res"
  shows "sound_report p res"
  using run_voblint_report_sound [OF assms] by blast
```

The common point of the source-level results. Whatever configuration produced the report, the policy it runs with gives every valid activation trace a context, and every context's activation bucket lies in what the report's states for that context describe. The context-free coverage below and the per-context spine both start here.

```
theorem run_voblint_contexts:
  assumes "run_voblint config p = Analysed res"
  shows "context_sound p res (run_policy config p) (run_root config)"
  using run_voblint_report_sound [OF assms] by blast
```

Per context: the stores every valid activation trace carrying context `c` reaches at `v`, under the policy the run uses, lie in the stores the report's states for `c` at `v` describe. A context the report lists no state for at `v` therefore reaches no store there.

```
theorem run_voblint_context_covers:
  assumes "run_voblint config p = Analysed res"
  shows "𝒜declared_global p,run_policy config p,run_root config,prog_cfg p,           cinit_stores (declared_global p)
v c
          ⊆ report_context_sem res v c"
  using run_voblint_contexts [OF assms] unfolding context_sound_def by blast

corollary run_voblint_context_absent:
  assumes "run_voblint config p = Analysed res" and "report_states_in res v c = {}"
  shows "𝒜declared_global p,run_policy config p,run_root config,prog_cfg p,           cinit_stores (declared_global p)
v c = {}"
  using run_voblint_context_covers [OF assms(1), of v c] assms(2)
  unfolding report_context_sem_def by blast
```

So a report's context column is exhaustive at each point: by `run_voblint_context_absent`, an activation in a context the report lists no state for at a point never reaches that point.

```
lemma run_voblint_consistent:
  "run_voblint config p = Analysed res ⟹ consistent_report res"
  using run_voblint_sound unfolding sound_report_def by blast

lemma run_voblint_well_formed:
  "run_voblint config p = Analysed res ⟹ well_formed_report res"
  by (auto intro: analysis_report_of_well_formed)
```

The report contract in one statement: an analysed answer was given for a valid configuration and a well-formed program, and is a well-formed, sound report for exactly that configuration and the program's compiled graph. The theorems below are its consequences.

```
theorem run_voblint_report_contract:
  assumes "run_voblint config p = Analysed res"
  shows "valid_config config" "wf_program_compile_input_exec p"
    and "report_config res = config" "report_cfg res = prog_cfg p"
    and "well_formed_report res" "sound_report p res"
  using assms
  by (auto elim: run_voblint_AnalysedE
      simp: run_voblint_config run_voblint_cfg run_voblint_well_formed run_voblint_sound)

theorem run_voblint_covers:
  assumes "run_voblint config p = Analysed res"
  shows "𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v ⊆ ℛres v"
proof -
  note ctx = run_voblint_contexts [OF assms, unfolded context_sound_def]
  have "𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v
          = (⋃c. 𝒜declared_global p,run_policy config p,run_root config,prog_cfg p,                    cinit_stores (declar
v c)"
    by (rule node_collect_eq_Union_activation_of_has_context) (use ctx in blast)
  also have "... ⊆ ℛres v"
    using context_sound_point [OF run_voblint_contexts [OF assms]] by blast
  finally show ?thesis .
qed
```

Every store the program reaches at a point lies in the report's semantics there, and so in its verdict semantics.

```
corollary run_voblint_collect_sound:
  assumes "run_voblint config p = Analysed res"
  shows "𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v
           ⊆ 𝒱res v"
  using run_voblint_covers[OF assms]
    analysis_report_verdicts_sound[OF run_voblint_consistent[OF assms]]
  by blast

corollary run_voblint_proved:
  assumes "run_voblint config p = Analysed res" and "PROVED res v e"
    and "s ∈ 𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v"
  shows "truthy (⟦e⟧e s)"
  using assms run_voblint_covers[OF assms(1)]
  using analysis_report_proved run_voblint_consistent by blast

corollary run_voblint_refuted:
  assumes "run_voblint config p = Analysed res" and "REFUTED res v e"
    and "s ∈ 𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v"
  shows "¬ truthy (⟦e⟧e s)"
  using assms run_voblint_covers[OF assms(1)]
  using analysis_report_refuted run_voblint_consistent by blast

corollary run_voblint_dead_unreached:
  assumes "run_voblint config p = Analysed res" and "DEAD res v"
  shows "𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v = {}"
  using run_voblint_covers[OF assms(1)] analysis_report_dead[OF assms(2)] by blast

corollary run_voblint_dead_check_unreached:
  assumes "run_voblint config p = Analysed res"
    and "chk ∈ set (report_checks res)" and "check_verdict chk = Dead"
  shows "𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) (check_point chk) = {}"
  using assms
  by (intro run_voblint_dead_unreached analysis_report_check_dead run_voblint_consistent)

theorem run_voblint_arithmetic_safe:
  assumes "run_voblint config p = Analysed res"
    and "s ∈ 𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v"
    and "∀d ∈ set (report_diagnostics res). diagnostic_point d ≠ v"
```

```
  shows "arithmetic_safe_at (prog_cfg p) v s"
  using run_voblint_sound[OF assms(1)] assms(2,3) unfolding sound_report_def by blast
```

An arithmetic error is the dual: wherever a collected store meets it, the divisor is zero.

```
theorem run_voblint_arithmetic_refuted:
  assumes "run_voblint config p = Analysed res"
    and "d ∈ set (report_diagnostics res)" and "diagnostic_verdict d = Check_Refuted"
    and "s ∈ 𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) (diagnostic_point d)"
  shows "⟦arithmetic_divisor (diagnostic_obligation d)⟧e s = 0"
proof -
  have "s ∈ ℛres (diagnostic_point d)"
    using run_voblint_sound[OF assms(1)] assms(4) unfolding sound_report_def by blast
  then have "¬ truthy (⟦arithmetic_condition (diagnostic_obligation d)⟧e s)"
    using run_voblint_sound[OF assms(1)] assms(2,3) unfolding sound_report_def by blast
  then show ?thesis using arithmetic_condition_safe by blast
qed

corollary run_voblint_arithmetic_intra_safe:
  assumes "run_voblint config p = Analysed res"
    and "s ∈ 𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v"
    and "∀d ∈ set (report_diagnostics res). diagnostic_point d ≠ v"
    and "(v, action, w) ∈ intra (prog_cfg p)"
    and "e ∈ set (edge_expressions action)"
    and "divisor ∈ expression_divisors e"
  shows "⟦divisor⟧e s ≠ 0"
  using run_voblint_arithmetic_safe[OF assms(1-3)]
    arithmetic_expression_sites_intra[OF finite_intra_prog_cfg assms(4)] assms(5,6)
  unfolding arithmetic_safe_at_def by blast
```

Whatever the configuration, the report lists one check per compiled check, at the check's node and with its label and condition, in graph order.

```
corollary run_voblint_check_sites:
  assumes "run_voblint config p = Analysed res"
  shows "map (λchk. (check_point chk, check_label chk, check_exp chk)) (report_checks res)
         = check_sites (prog_cfg p)"
  using run_voblint_sound[OF assms] unfolding sound_report_def by blast

lemma run_voblint_wf:
  "run_voblint config p = Analysed res ⟹ wf_program_compile_input p"
  by (blast intro: wf_program_compile_input_exec_sound)
```

The endpoint

Run the program, stop wherever you like, and ask any configuration for a report. The configuration the run reaches is observed at some node (`at_node`): it lifts to the program's elaboration `elab_pt`, whose annotated command names the node, which the residual alone could not, since two equal branches leave equal residuals at different nodes. At every node it is observed at, the store in your hands is one the collecting semantics admits, the report's semantics contains it, and every definite verdict the report gives there holds of it.

```
theorem run_voblint_source_sound:
  fixes p :: imp_prog and s0 s :: store
  defines G_def: "𝒢 ≡ declared_global p"
      and Pi_def: "Π ≡ prog_table p"
      and g_def: "g ≡ prog_cfg p"
  assumes s0: "s0 ∈ cinit_stores 𝒢"
      and run: "𝒢, Π ⊢ (main_body Π, s0, []) →p* (residual, s, frs)"
      and ans: "run_voblint config p = Analysed res"
  shows "(∃v. p, s0 ⊢ (residual, s, frs) ▷ v)
       ∧ (∀v. p, s0 ⊢ (residual, s, frs) ▷ v ⟶
                s ∈ 𝒞𝒢,g,cinit_stores 𝒢 v
              ∧ s ∈ ℛres v
              ∧ s ∈ 𝒱res v)"
proof -
  define E where "E = elab_pt (prog_table p) (prog_procs p)"
  from ans have wf: "wf_program_compile_input p" by (rule run_voblint_wf)
  have cfg: "compile_prog E (prog_procs p) = g"
    unfolding E_def g_def prog_cfg_def by (rule compile_prog_elab_pt[OF wf])
  have ept: "erase_pt E = Π" and mb: "erase (main_body E) = main_body Π"
    unfolding E_def Pi_def by (rule erase_elab_pt [OF wf])+
  have point_collected: "v = point c ∧ s ∈ 𝒞𝒢,g,cinit_stores 𝒢 v"
    if runc: "𝒢, E ⊢ (main_body E, s0, []) →p* (c, s, frs)"
      and m: "E, g, prog_main_name ⊢ (c, s, frs) ≈ (v, s, stk)" for c v stk
  proof -
```

```
    from source_reaches_node_collect
           [OF wf_compile_input_elab_pt [OF wf, folded E_def] s0 [unfolded G_def]
               runc [unfolded G_def]]
    obtain v' stk'
      where m': "E, g, prog_main_name ⊢ (c, s, frs) ≈ (v', s, stk')"
        and mem: "s ∈ 𝒞declared_global p,g,cinit_stores (declared_global p) v'"
      unfolding cfg by blast
    have "v = point c" and "v' = point c"
      by (rule csim_point_compile_prog [OF wf m [unfolded E_def g_def prog_cfg_def
            compile_prog_elab_pt [OF wf, symmetric]]],
          rule csim_point_compile_prog [OF wf m' [unfolded E_def g_def prog_cfg_def
            compile_prog_elab_pt [OF wf, symmetric]]])
    with mem show ?thesis unfolding G_def by simp
  qed
  have ex: "∃v. p, s0 ⊢ (residual, s, frs) ▷ v"
  proof -
    from psteps_lift [of 𝒢 E, unfolded ept, OF run mb]
    obtain c where runc: "𝒢, E ⊢ (main_body E, s0, []) →p* (c, s, frs)"
      and ec: "erase c = residual"
      by blast
    from source_reaches_node_collect
           [OF wf_compile_input_elab_pt [OF wf, folded E_def] s0 [unfolded G_def]
               runc [unfolded G_def]]
    obtain v stk where m: "E, g, prog_main_name ⊢ (c, s, frs) ≈ (v, s, stk)"
      unfolding cfg by blast
    with point_collected [OF runc m] runc ec show ?thesis
      unfolding at_node.simps E_def G_def g_def by blast
  qed
  have all: "s ∈ 𝒞𝒢,g,cinit_stores 𝒢 v ∧ s ∈ ℛres v ∧ s ∈ 𝒱res v"
    if at: "p, s0 ⊢ (residual, s, frs) ▷ v" for v
  proof -
    from at obtain c stk where runc: "𝒢, E ⊢ (main_body E, s0, []) →p* (c, s, frs)"
      and m: "E, g, prog_main_name ⊢ (c, s, frs) ≈ (v, s, stk)"
      unfolding at_node.simps E_def G_def g_def by blast
    from point_collected [OF runc m] run_voblint_covers [OF ans]
      run_voblint_collect_sound [OF ans]
    show ?thesis unfolding G_def g_def by blast
  qed
  from ex all show ?thesis by blast
qed
```

The same endpoint without contexts: the state the report holds at the node, joined over its contexts, describes the store too.

```
corollary run_voblint_source_sound_joined:
  fixes p :: imp_prog and s0 s :: store
  defines G_def: "𝒢 ≡ declared_global p"
      and Pi_def: "Π ≡ prog_table p"
      and g_def: "g ≡ prog_cfg p"
  assumes s0: "s0 ∈ cinit_stores 𝒢"
      and run: "𝒢, Π ⊢ (main_body Π, s0, []) →p* (residual, s, frs)"
      and ans: "run_voblint config p = Analysed res"
  shows "∃v stk. Π, g, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
                ∧ s ∈ γres (report_point_join res v)"
  using run_voblint_source_sound[OF s0[unfolded G_def] run[unfolded G_def Pi_def] ans]
    at_node_csim [OF run_voblint_wf [OF ans]] report_sem_point_join
  unfolding Pi_def g_def by blast
```

The same endpoint, read at a check. A run about to execute `Check l e` finds a check labelled `l` for `e` in the report, listed at a node this very store reaches, and that check's verdict holds of the store.

```
theorem run_voblint_check_sound:
  fixes p :: imp_prog and s0 s :: store
  defines G_def: "𝒢 ≡ declared_global p"
      and Pi_def: "Π ≡ prog_table p"
      and g_def: "g ≡ prog_cfg p"
  assumes s0: "s0 ∈ cinit_stores 𝒢"
      and run: "𝒢, Π ⊢ (main_body Π, s0, []) →p* (residual, s, frs)"
      and chk: "next_check residual = Some (l, e)"
      and ans: "run_voblint config p = Analysed res"
  shows "∃c ∈ set (report_checks res). check_label c = l ∧ check_exp c = e
          ∧ s ∈ 𝒞𝒢,g,cinit_stores 𝒢 (check_point c)
          ∧ check_verdict c ≠ Dead
          ∧ (check_verdict c = Decided Check_Proved ⟶ truthy (⟦e⟧e s))
          ∧ (check_verdict c = Decided Check_Refuted ⟶ ¬ truthy (⟦e⟧e s))"
proof -
```

```
  from run_voblint_source_sound [OF s0 [unfolded G_def] run [unfolded G_def Pi_def] ans]
  obtain v where at: "p, s0 ⊢ (residual, s, frs) ▷ v"
      and mem: "s ∈ 𝒞declared_global p,prog_cfg p,cinit_stores (declared_global p) v"
      and sem: "s ∈ ℛres v"
      and verdicts: "s ∈ 𝒱res v"
    by blast
  from at_node_csim [OF run_voblint_wf [OF ans] at]
  obtain stk where m: "prog_table p, prog_cfg p, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)"
    by blast
  from csim_next_check_edge [OF m chk]
  have "(v, l, e) ∈ set (check_sites (prog_cfg p))" by auto
  then obtain c
    where c: "c ∈ set (report_checks res)" and "check_point c = v"
      and "check_label c = l" and "check_exp c = e"
    unfolding run_voblint_check_sites [OF ans, symmetric] by auto
  moreover have "check_verdict c ≠ Dead"
  proof
    assume "check_verdict c = Dead"
    from analysis_report_check_dead[OF run_voblint_consistent[OF ans] c this]
    have "ℛres v = {}" using ‹check_point c = v› analysis_report_dead by blast
    with sem show False by blast
  qed
  moreover have holds: "verdict_holds r e s" if "check_verdict c = Decided r" for r
  proof (rule verdict_storesD [OF verdicts])
    show "HAS_VERDICT res v e r"
      unfolding HAS_VERDICT_def
      using c ‹check_point c = v› ‹check_exp c = e› that by auto
  qed
  ultimately show ?thesis
    using mem holds [of Check_Proved] holds [of Check_Refuted] unfolding G_def g_def by auto
qed
```

Read by label, which is how a report finds a source check's row. When no two rows share a label, the row labelled `l` is the one the theorem above found, so its verdict holds of the store. Distinct labels are a premise rather than a fact about every report: a procedure called from several sites is compiled once, so its checks are listed once, but two source checks given the same label would share it. The premise is decidable on the report, so a consumer checks it before trusting a lookup. What stays outside this development is the label itself: that the parser wrote each check's own source position into it.

```
corollary run_voblint_labelled_check_sound:
  fixes p :: imp_prog and s0 s :: store
  assumes s0: "s0 ∈ cinit_stores (declared_global p)"
      and run: "declared_global p, prog_table p
                  ⊢ (main_body (prog_table p), s0, [])
                    →p* (residual, s, frs)"
      and chk: "next_check residual = Some (l, e)"
      and ans: "run_voblint config p = Analysed res"
      and distinct: "distinct (map check_label (report_checks res))"
  shows "∃c ∈ set (report_checks res). check_label c = l"
    and "∀c ∈ set (report_checks res). check_label c = l ⟶
           check_exp c = e
           ∧ check_verdict c ≠ Dead
           ∧ (check_verdict c = Decided Check_Proved ⟶ truthy (⟦e⟧e s))
           ∧ (check_verdict c = Decided Check_Refuted ⟶ ¬ truthy (⟦e⟧e s))"
proof -
  from run_voblint_check_sound [OF s0 run chk ans]
  obtain c0 where c0: "c0 ∈ set (report_checks res)" "check_label c0 = l" "check_exp c0 = e"
    and v0: "check_verdict c0 ≠ Dead"
      "check_verdict c0 = Decided Check_Proved ⟶ truthy (⟦e⟧e s)"
      "check_verdict c0 = Decided Check_Refuted ⟶ ¬ truthy (⟦e⟧e s)"
    by blast
  from c0 show "∃c ∈ set (report_checks res). check_label c = l" by blast
  have inj: "inj_on check_label (set (report_checks res))"
    using distinct by (simp add: distinct_map)
  show "∀c ∈ set (report_checks res). check_label c = l ⟶
           check_exp c = e
           ∧ check_verdict c ≠ Dead
           ∧ (check_verdict c = Decided Check_Proved ⟶ truthy (⟦e⟧e s))
           ∧ (check_verdict c = Decided Check_Refuted ⟶ ¬ truthy (⟦e⟧e s))"
  proof (intro ballI impI)
    fix c
    assume "c ∈ set (report_checks res)" and "check_label c = l"
    with c0 inj have "c = c0" by (metis inj_onD)
    with c0 v0 show "check_exp c = e
           ∧ check_verdict c ≠ Dead
           ∧ (check_verdict c = Decided Check_Proved ⟶ truthy (⟦e⟧e s))
```

```
          ∧ (check_verdict c = Decided Check_Refuted ⟶ ¬ truthy (⟦e⟧ₑ s))"
      by blast
  qed
qed
```

The semantic spine

The endpoint above starts at the collecting semantics. The spine keeps the contexts the report erases, for the policy the configuration runs with: a source run is represented by a valid activation trace `t`, the policy gives `t` a context `c`, the run's store lies in the bucket of `c`, the buckets together are the collecting semantics, and the report and its verdicts contain that. It follows from `run_voblint_contexts` and holds for every configuration.

```
theorem run_voblint_spine:
  fixes p :: imp_prog and s0 s :: store and config :: analysis_config
  defines G_def: "𝒢 ≡ declared_global p"
      and Pi_def: "Π ≡ prog_table p"
      and g_def: "g ≡ prog_cfg p"
      and S_def: "S ≡ cinit_stores (declared_global p)"
      and adm_def: "adm ≡ run_policy config p"
      and rc_def: "rc ≡ run_root config"
  assumes s0: "s0 ∈ S"
      and run: "𝒢, Π ⊢ (main_body Π, s0, []) →ₚ* (residual, s, frs)"
      and ans: "run_voblint config p = Analysed res"
  shows "∃v stk t c. Π, g, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
          ∧ activation_trace_repr 𝒢 g S (v, s, stk) t
          ∧ activation_context_rel 𝒢 adm rc g t c
          ∧ s ∈ 𝒜𝒢,adm,rc,g,S v c
          ∧ (⋃c'. 𝒜𝒢,adm,rc,g,S v c') = 𝒞𝒢,g,S v
          ∧ 𝒞𝒢,g,S v ⊆ ℛres v
          ∧ ℛres v ⊆ 𝒱res v"
proof -
  have has_ctx: "⋀t. t ∈ 𝒯𝒢,g,S ⟹ ∃c. activation_context_rel 𝒢 adm rc g t c"
    using run_voblint_contexts [OF ans]
    unfolding context_sound_def G_def g_def S_def adm_def rc_def by blast
  have buckets: "⋀v. (⋃c'. 𝒜𝒢,adm,rc,g,S v c') = 𝒞𝒢,g,S v"
    by (rule node_collect_eq_Union_activation_of_has_context [OF has_ctx, symmetric])
  have cfg: "g = compile_prog Π (prog_procs p)"
    unfolding g_def Pi_def by (rule prog_cfg_def)
  from source_store_in_activation_collect
        [OF run_voblint_wf [OF ans, folded G_def Pi_def] s0 run has_ctx [unfolded cfg]]
  have witness: "∃v stk t c. Π, g, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
          ∧ activation_trace_repr 𝒢 g S (v, s, stk) t
          ∧ activation_context_rel 𝒢 adm rc g t c
          ∧ s ∈ 𝒜𝒢,adm,rc,g,S v c"
    unfolding cfg .
  from witness buckets run_voblint_covers [OF ans]
       analysis_report_verdicts_sound [OF run_voblint_consistent [OF ans]]
  show ?thesis unfolding G_def g_def S_def by blast
qed
end
```

17.13 Trace_Run

```
theory Trace_Run
  imports
    Analysis_Render
    "Voblint_Solver.Solver_Trace"
begin
```

The solver trace of one run

`Voblint_Solver.Solver_Trace` gives the executable solver a traced copy. This theory adds the two things only a run knows: which context each call is routed to, and how to read the solver's unknowns and values back as the result reads them. It assembles them into `run_voblint_traced`, which is `run_voblint` in the logic and the only exported entry point whose code reaches `trace_event`. `run_voblint` itself keeps its own code equations and runs no hook.

Routing

A route receives the calling unknown's node and context and the entry state, and returns the callee's context. The traced run wraps each context policy's route so that it reports that triple.

```
datatype ('x, 'l, 'c) route_event = Ev_Route 'x 'l 'c

definition traced_route ::
    "(pp ⇒ 'c ⇒ 'l ⇒ call_action ⇒ 'c) ⇒ pp ⇒ 'c ⇒ 'l ⇒ call_action ⇒ 'c" where
  "traced_route route u ctx d ca =
    (let c = route u ctx d ca; _ = trace_event STR ''route'' (λ_. Ev_Route (u, ctx) d c) in c)"

lemma traced_route_eq [simp]: "traced_route route = route"
  by (simp add: fun_eq_iff traced_route_def trace_event_def Let_def)
```

Reading unknowns and values back

The solver is generic in its unknowns and values. Where a run fixes them, it hands the trace the functions that read them as the result does: a context as the result shows it, a global unknown as the node-owned buffer, a program global, or the seed of a procedure entry in a context, a solver value's local part as a state, a global unknown's value as the globals it describes, and an entry state as a state. With shared program globals a local value holds its globals at bottom by construction, which the emptiness check would read as unreachable; it is shown without that check and without the globals, and a program global's unknown shows that global alone.

```
datatype 'c trace_key = Trace_Buffer | Trace_Global vname | Trace_Seed cfg_node 'c

datatype ('c, 'g, 'd, 'l) trace_printers = Trace_Printers
  "'c ⇒ abstract_value analysis_context"
  "'g ⇒ 'c trace_key"
  "'d ⇒ abstract_value analysis_view lifted"
  "vname option ⇒ 'd ⇒ abstract_value analysis_view lifted"
  "'l ⇒ abstract_value analysis_view lifted"

fun key_of_global_unknown :: "(vname, 'c) global_unknown ⇒ 'c trace_key" where
  "key_of_global_unknown Analysis_Buffer = Trace_Buffer"
| "key_of_global_unknown (Analysis_Global x) = Trace_Global x"
| "key_of_global_unknown (Activation_Seed n c) = Trace_Seed n c"

definition mcp_trace_printers ::
    "program_globals ⇒ analysis_domain list ⇒ imp_prog
       ⇒ ('c ⇒ abstract_value analysis_context)
       ⇒ ('c, (vname, 'c) global_unknown, (mcp_st lifted, mcp_st lifted) dg_state, mcp_st lifted)
           trace_printers" where
  "mcp_trace_printers pg as p ctx_view =
     (let raw = (λvs d. map_lift (mcp_render (activation as) vs)
                          (map_lift (mcp_rd (declared_global p)) d));
          view = (λd. raw (program_vars p) (canonicalize_lift (mcp_emp (activation as) p) d));
          local = (case pg of
                     Program_Globals_Flow_Sensitive ⇒ view
                   | Program_Globals_Flow_Insensitive ⇒
                       raw (filter (λx. ¬ declared_global p x) (program_vars p)));
          globals = (λn. case n of
                       None ⇒ filter (declared_global p) (program_vars p)
                     | Some x ⇒ [x])
      in Trace_Printers ctx_view key_of_global_unknown (λd. local (dg_local d))
           (λn d. raw (globals n) (dg_global d)) local)"
```

Each context mode fixes the types of its contexts and global unknowns; they are spelled out, since a type variable left free in a code equation's right-hand side makes code export drop the equation.

The traced report is `analysis_report_of` with a run event carrying the printers, the traced solve `solve_c_traced` in place of `mcp_solve_c`, and every route wrapped by `traced_route`.

```
fun analysis_report_traced :: "analysis_config ⇒ imp_prog ⇒ analysis_report option" where
  "analysis_report_traced (Analysis_Config as r Ctx_None pg) p =
     (let _ = trace_event STR ''run''
              (λ_. mcp_trace_printers pg as p (λ_ :: unit. Context_Unit)) in
      map_option
       (λsol. report_of (Analysis_Config as r Ctx_None pg) (λ_. Key_List []) (λ_. Report_Unit)
          (mcp_classify (activation as))
          (mcp_run_of (activation as) pg Analysis_Global Activation_Seed
             (λ_. traced_route route_unit) (declared_global p) p sol) p)
       (solve_c_traced r
          (mcp_equations (activation as) pg Analysis_Buffer Analysis_Global Activation_Seed
             (λ_. traced_route route_unit) () (declared_global p) p)
          (cfg_exit (prog_cfg p), ())))"
| "analysis_report_traced (Analysis_Config as r Ctx_EntryState pg) p =
     (let _ = trace_event STR ''run''
              (λ_. mcp_trace_printers pg as p
                     (λctx. Context_Entry (mcp_ctx_values (activation as) ctx))) in
      map_option
       (λsol. report_of (Analysis_Config as r Ctx_EntryState pg)
          (entry_ctx_key (activation as)) Report_Entry (mcp_classify (activation as))
          (mcp_run_of (activation as) pg Analysis_Global Activation_Seed
             (λG. traced_route (mcp_formals_route (activation as) G)) (declared_global p) p sol) p)
       (solve_c_traced r
          (mcp_equations (activation as) pg Analysis_Buffer Analysis_Global Activation_Seed
             (λG. traced_route (mcp_formals_route (activation as) G)) mcp_root_ctx
             (declared_global p) p)
          (cfg_exit (prog_cfg p), mcp_root_ctx)))"
| "analysis_report_traced (Analysis_Config as r (Ctx_CallString k) pg) p =
     (let _ = trace_event STR ''run''
              (λ_. mcp_trace_printers pg as p Context_Call_String) in
      map_option
       (λsol. report_of (Analysis_Config as r (Ctx_CallString k) pg)
          (λctx. Key_List (map Key_Node ctx)) Report_Call_String
          (mcp_classify (activation as))
          (mcp_run_of (activation as) pg Analysis_Global Activation_Seed
             (λ_. traced_route (cs_route k)) (declared_global p) p sol) p)
       (solve_c_traced r
          (mcp_equations (activation as) pg Analysis_Buffer Analysis_Global Activation_Seed
             (λ_. traced_route (cs_route k)) [] (declared_global p) p)
          (cfg_exit (prog_cfg p), [])))"

lemma analysis_report_traced_eq: "analysis_report_traced config p = analysis_report_of config p"
  by (induct config p rule: analysis_report_of.induct)
     (simp_all add: Let_def trace_event_def solve_c_traced_eq mcp_solve_c_def)
```

The traced run

The run the command-line tool's `--trace` and the playground's trace replay call. It answers exactly as `run_voblint` (`run_voblint_traced_eq`), so every theorem about `run_voblint` covers its answer.

```
definition run_voblint_traced ::
  "analysis_config ⇒ imp_prog ⇒ analysis_report analysis_answer"
where
  "run_voblint_traced config p =
     (if ¬ valid_config config then Invalid_Activation
      else if ¬ wf_program_compile_input_exec p then Malformed_Program
      else case analysis_report_traced config p of
             None ⇒ No_Answer
           | Some res ⇒ Analysed res)"

theorem run_voblint_traced_eq: "run_voblint_traced = run_voblint"
  by (simp add: fun_eq_iff run_voblint_traced_def run_voblint_def analysis_report_traced_eq)
```

The hook

`trace_event` is `()` in the logic. The exported OCaml calls `Solver_trace_hook.emit` instead, which records the event only when tracing is on, forcing the suspension then and only then, and returns `()`. This mapping is trusted in the same way as every other target-language mapping: the hook must return, raise nothing and leave

the solver's values alone. Only the traced constants above name `trace_event`, so only `run_voblint_traced` relies on this mapping. No mapping exists for the Eval target, so evaluation in proofs runs the equation.

code_printing constant `trace_event` ⇀ `(OCaml) "Solver'_trace'_hook.emit"`

end

17.14 Arithmetic_Diagnostics

```
theory Arithmetic_Diagnostics
  imports "Voblint_Framework.Contextual_Check_Report"
begin
```

Division and remainder occurrences

Diagnostics inspect all expression operands, matching VIMP's total, pure expression model. They neither change execution nor assume short-circuit evaluation for logical operators. Lists retain repeated syntactically equal occurrences.

```
datatype arithmetic_obligation = Arithmetic_Obligation
  (arithmetic_operation: exp)
  (arithmetic_divisor: exp)

fun arithmetic_obligations :: "exp ⇒ arithmetic_obligation list" where
  "arithmetic_obligations (N n) = []"
| "arithmetic_obligations (V x) = []"
| "arithmetic_obligations (Not a) = arithmetic_obligations a"
| "arithmetic_obligations (And a b) =
     arithmetic_obligations a @ arithmetic_obligations b"
| "arithmetic_obligations (Or a b) =
     arithmetic_obligations a @ arithmetic_obligations b"
| "arithmetic_obligations (Div a b) =
     arithmetic_obligations a @ arithmetic_obligations b @
       [Arithmetic_Obligation (Div a b) b]"
| "arithmetic_obligations (Mod a b) =
     arithmetic_obligations a @ arithmetic_obligations b @
       [Arithmetic_Obligation (Mod a b) b]"
| "arithmetic_obligations (Plus a b) =
     arithmetic_obligations a @ arithmetic_obligations b"
| "arithmetic_obligations (Minus a b) =
     arithmetic_obligations a @ arithmetic_obligations b"
| "arithmetic_obligations (Times a b) =
     arithmetic_obligations a @ arithmetic_obligations b"
| "arithmetic_obligations (Less a b) =
     arithmetic_obligations a @ arithmetic_obligations b"
| "arithmetic_obligations (LessEq a b) =
     arithmetic_obligations a @ arithmetic_obligations b"
| "arithmetic_obligations (Greater a b) =
     arithmetic_obligations a @ arithmetic_obligations b"
| "arithmetic_obligations (GreaterEq a b) =
     arithmetic_obligations a @ arithmetic_obligations b"
| "arithmetic_obligations (Eq a b) =
     arithmetic_obligations a @ arithmetic_obligations b"
| "arithmetic_obligations (NotEq a b) =
     arithmetic_obligations a @ arithmetic_obligations b"
```

The set specification ignores occurrence numbering and names every denominator appearing anywhere in an expression. The extraction theorem connects it to the occurrence-preserving list consumed by the report.

```
fun expression_divisors :: "exp ⇒ exp set" where
  "expression_divisors (N n) = {}"
| "expression_divisors (V x) = {}"
| "expression_divisors (Not a) = expression_divisors a"
| "expression_divisors (And a b) =
     expression_divisors a ∪ expression_divisors b"
| "expression_divisors (Or a b) =
     expression_divisors a ∪ expression_divisors b"
| "expression_divisors (Div a b) =
     insert b (expression_divisors a ∪ expression_divisors b)"
| "expression_divisors (Mod a b) =
     insert b (expression_divisors a ∪ expression_divisors b)"
| "expression_divisors (Plus a b) =
     expression_divisors a ∪ expression_divisors b"
| "expression_divisors (Minus a b) =
     expression_divisors a ∪ expression_divisors b"
| "expression_divisors (Times a b) =
     expression_divisors a ∪ expression_divisors b"
| "expression_divisors (Less a b) =
     expression_divisors a ∪ expression_divisors b"
| "expression_divisors (LessEq a b) =
     expression_divisors a ∪ expression_divisors b"
| "expression_divisors (Greater a b) =
     expression_divisors a ∪ expression_divisors b"
```

```
| "expression_divisors (GreaterEq a b) =
     expression_divisors a ∪ expression_divisors b"
| "expression_divisors (Eq a b) =
     expression_divisors a ∪ expression_divisors b"
| "expression_divisors (NotEq a b) =
     expression_divisors a ∪ expression_divisors b"

lemma arithmetic_obligations_divisors:
  "arithmetic_divisor ‘ set (arithmetic_obligations e) = expression_divisors e"
  by (induction e) auto

definition arithmetic_condition :: "arithmetic_obligation ⇒ exp" where
  "arithmetic_condition obligation = NotEq (arithmetic_divisor obligation) (N 0)"

lemma arithmetic_condition_safe:
  "truthy (⟦arithmetic_condition obligation⟧e s) ⟷
    ⟦arithmetic_divisor obligation⟧e s ≠ 0"
  by (auto simp: arithmetic_condition_def split: if_splits)
```

Sites and their solved classifications

`arithmetic_expression_sites` collects the expressions each CFG edge and call evaluates; `arithmetic_sites` turns them into the divisor obligations the solved table then classifies.

Only suppress a negative guard edge when the same point has a matching positive guard edge. All other expression-bearing edges remain in the report.

```
definition arithmetic_expression_sites :: "cfg ⇒ (pp × exp list) list" where
  "arithmetic_expression_sites g =
     (let edges = cfg_intra_list g
      in map (λ(u, a, v). (u, edge_expressions a))
           (filter (λ(u, a, v). case a of EA_AssumeNot e ⇒
             ¬ list_ex (λ(u', a', v'). u = u' ∧ a' = EA_Assume e) edges
             | _ ⇒ True) edges)) @
       map (λ(u, ca, ce, k). (u, ce_args ca)) (cfg_calls_list g)"

definition arithmetic_sites :: "cfg ⇒ (pp × arithmetic_obligation list) list" where
  "arithmetic_sites g =
     map (λ(v, es). (v, concat (map arithmetic_obligations es)))
       (arithmetic_expression_sites g)"

datatype arithmetic_diagnostic = Arithmetic_Diagnostic
  (diagnostic_point: pp)
  (diagnostic_occurrence: nat)
  (diagnostic_obligation: arithmetic_obligation)
  (diagnostic_verdict: check_result)

fun arithmetic_diagnostic_of ::
    "pp ⇒ nat ⇒ arithmetic_obligation ⇒ contextual_verdict
       ⇒ arithmetic_diagnostic list" where
  "arithmetic_diagnostic_of v i obligation (Lifted Check_Refuted) =
     [Arithmetic_Diagnostic v i obligation Check_Refuted]"
| "arithmetic_diagnostic_of v i obligation (Lifted Check_Unknown) =
     [Arithmetic_Diagnostic v i obligation Check_Unknown]"
| "arithmetic_diagnostic_of v i obligation _ = []"

definition arithmetic_diagnostics ::
    "cfg ⇒ ('ctx, 'a) solved_table ⇒ (exp ⇒ 'a ⇒ check_result)
       ⇒ arithmetic_diagnostic list" where
  "arithmetic_diagnostics g r classify =
     concat (map (λ(v, obligations).
       concat (map (λ(i, obligation).
         arithmetic_diagnostic_of v i obligation
           (point_verdict r classify v (arithmetic_condition obligation)))
         (zip [0..<length obligations] obligations))) (arithmetic_sites g))"

lemma arithmetic_expression_sites_intra:
  assumes "finite (intra g)" and "(v, a, w) ∈ intra g"
  shows "(v, edge_expressions a) ∈ set (arithmetic_expression_sites g)"
proof -
  let ?edges = "cfg_intra_list g"
  have mem: "(v, a, w) ∈ set ?edges" using assms by simp
  have kept: "⋀b t. (v, b, t) ∈ set ?edges ⟹
      (case b of EA_AssumeNot e ⇒
        ¬ list_ex (λ(u', a', v'). v = u' ∧ a' = EA_Assume e) ?edges
        | _ ⇒ True) ⟹
```

```
      (v, edge_expressions b) ∈ set (arithmetic_expression_sites g)"
    unfolding arithmetic_expression_sites_def
    by (force simp: Let_def image_iff)
  show ?thesis
  proof (cases "∃e. a = EA_AssumeNot e")
    case False
    with mem show ?thesis by (intro kept[of a w]) (auto split: edge_action.splits)
  next
    case True
    then obtain e where a: "a = EA_AssumeNot e" by blast
    show ?thesis
    proof (cases "list_ex (λ(u', a', v'). v = u' ∧ a' = EA_Assume e) ?edges")
      case False
      from kept[OF mem] False a show ?thesis by simp
    next
      case True
      then obtain t where pos: "(v, EA_Assume e, t) ∈ set ?edges"
        by (auto simp: list_ex_iff)
      from kept[OF pos] a show ?thesis by simp
    qed
  qed
qed

lemma arithmetic_expression_sites_call:
  assumes "finite (calls g)" and "(v, ca, ce, k) ∈ calls g"
  shows "(v, ce_args ca) ∈ set (arithmetic_expression_sites g)"
  using assms unfolding arithmetic_expression_sites_def
  by (force simp: image_iff)

lemma arithmetic_sites_divisor:
  assumes "(v, es) ∈ set (arithmetic_expression_sites g)"
    and "e ∈ set es" and "divisor ∈ expression_divisors e"
  obtains obligations obligation where "(v, obligations) ∈ set (arithmetic_sites g)"
    and "obligation ∈ set obligations" and "arithmetic_divisor obligation = divisor"
proof -
  from assms(3) obtain obligation where
    ob: "obligation ∈ set (arithmetic_obligations e)"
      "arithmetic_divisor obligation = divisor"
    using arithmetic_obligations_divisors[of e] by force
  have site: "(v, concat (map arithmetic_obligations es)) ∈ set (arithmetic_sites g)"
    using assms(1) by (auto simp: arithmetic_sites_def)
  have "obligation ∈ set (concat (map arithmetic_obligations es))"
    using ob assms(2) by auto
  from that[OF site this ob(2)] show thesis .
qed

lemma arithmetic_diagnostics_absent:
  assumes site: "(v, obligations) ∈ set (arithmetic_sites g)"
    and ob: "obligation ∈ set obligations"
    and absent: "∀d ∈ set (arithmetic_diagnostics g r classify). diagnostic_point d ≠ v"
  shows "point_verdict r classify v (arithmetic_condition obligation) = Bot ∨
    point_verdict r classify v (arithmetic_condition obligation) = Lifted Check_Proved"
proof -
  have "obligation ∈ set (map snd (zip [0..<length obligations] obligations))"
    using ob by simp
  then obtain pair where pair: "pair ∈ set (zip [0..<length obligations] obligations)"
    and snd: "snd pair = obligation"
    unfolding set_map by blast
  have indexed: "(fst pair, obligation) ∈ set (zip [0..<length obligations] obligations)"
    using pair snd by (cases pair) auto
  define i where "i = fst pair"
  have indexed: "(i, obligation) ∈ set (zip [0..<length obligations] obligations)"
    using indexed unfolding i_def .
  let ?verdict = "point_verdict r classify v (arithmetic_condition obligation)"
  have included: "set (arithmetic_diagnostic_of v i obligation ?verdict)
      ⊆ set (arithmetic_diagnostics g r classify)"
    using site indexed
    unfolding arithmetic_diagnostics_def
    by (force simp only: set_concat set_map image_iff case_prod_beta)
  have point: "⋀d. d ∈ set (arithmetic_diagnostic_of v i obligation ?verdict) ⟹
      diagnostic_point d = v"
  proof -
    fix d
    assume d: "d ∈ set (arithmetic_diagnostic_of v i obligation ?verdict)"
    show "diagnostic_point d = v"
    proof (cases ?verdict)
```

```
      case Bot
      with d show ?thesis by simp
    next
      case (Lifted verdict)
      with d show ?thesis by (cases verdict) auto
    qed
  qed
  have empty: "arithmetic_diagnostic_of v i obligation ?verdict = []"
    using included point absent
    by (meson empty_subsetI set_empty2 subset_antisym subset_iff)
  show ?thesis
  proof (cases "point_verdict r classify v (arithmetic_condition obligation)")
    case Bot
    then show ?thesis by simp
  next
    case (Lifted verdict)
    with empty show ?thesis by (cases verdict) auto
  qed
qed
```

An error names a point whose divisor condition the solved table refutes.

```
lemma arithmetic_diagnostic_of_refuted:
  assumes "d ∈ set (arithmetic_diagnostic_of v i obligation verdict)"
    and "diagnostic_verdict d = Check_Refuted"
  shows "verdict = Lifted Check_Refuted ∧ diagnostic_point d = v
    ∧ diagnostic_obligation d = obligation"
  using assms by (cases "(v, i, obligation, verdict)" rule: arithmetic_diagnostic_of.cases) auto

lemma arithmetic_diagnostics_refuted:
  assumes "d ∈ set (arithmetic_diagnostics g r classify)"
    and "diagnostic_verdict d = Check_Refuted"
  shows "point_verdict r classify (diagnostic_point d)
      (arithmetic_condition (diagnostic_obligation d)) = Lifted Check_Refuted"
  using assms unfolding arithmetic_diagnostics_def
  by (auto dest!: arithmetic_diagnostic_of_refuted)

end
```

Chapter 18

Voblint_Codegen

18.1 Voblint_Codegen

theory Voblint_Codegen
  imports
    "Voblint_CLI.Trace_Run"
begin

Code export surface

Top-level definitions normally supply their own code equations. Locale definitions need care: an inline `[code]` attribute is applied through interpretation and does not register the generic constant for direct calls with abstract locale parameters. Register those equations after the defining locale's `end`, in the theory that owns them, as the grouped `declare ... [code]` in `Voblint_Result.DG_Analysis` does. Keep this export theory focused on roots.

If a locale theorem still has assumptions, it is not an unconditional code equation. Discharge them through an interpretation or derive a code equation for the concrete caller. A successful attribute declaration alone does not establish executability: check its warnings, export the actual roots, and compile the OCaml.

Use `[code]` to retain a named function. `[code_unfold]` rewrites its uses during preprocessing and can duplicate its body in generated callers; it is not a substitute for a missing code equation.

This session owns executable exports; the examples session proves and demonstrates the exported definitions without materializing generated code.

One export block, deliberately. The external OCaml regression driver under `codegen/regression/ocaml/` needs a handful of CFG-inspection constants the CLI never calls; a second, narrower block would not make any surface narrower, because Isabelle emits the reachable transitive closure of whatever is named. Those constants are named here instead, so there is one generated artifact for one analysis.

This is still not the project's public API boundary, but it is not nothing either. Three things are decided separately here:

- the roots decide what the emitted OCaml *signature* exposes, and how much of it is transparent rather than abstract;
- their transitive closure decides what the *implementation* contains, which no shortening of the root list reduces;
- `module_name` decides the *packaging*.

So trimming a root narrows what a client can name and match on, while leaving the emitted code the same size. A supported external surface — one that could rename a constructor, or hide a representation behind an eliminator — belongs in the handwritten OCaml facade `cli/voblint.ml` rather than in the shape of this list.

The roots below are the *intended callable surface*: what handwritten OCaml under `cli/`, `codegen/regression/ocaml/` and `tests/property/` actually calls. Everything else in the emitted module is serializer-reachable implementation detail, still present and still callable — the signature narrows with the root list, the code does not. So the intent recorded here is not enforced. Handwritten OCaml names the export through the facade `cli/voblint.ml` (module `Voblint`), which re-exports this module's signature unchanged: the root list still decides what the facade can expose, and the facade is the one place a later, narrower surface would be stated.

Analysis entry goes through `run_voblint`, which checks the activation list and well-formedness and runs the requested analyses, update rule and context policy. Every combination is answered, so the CLI never decides legality. `run_voblint_traced` is the same function in the logic (`run_voblint_traced_eq`); only its code reports solver steps through the trace hook, and the CLI calls it only when a trace is requested.

The last group of roots is there for signature visibility rather than for dispatch. A constant the serializer does not consider public is emitted but left out of the module signature, and a datatype it does not consider public stays abstract, which makes it unmatchable. So anything handwritten OCaml names — even only to take it apart — has to be a root: `Bot`/`Lifted`, which the CLI matches to tell a dead point from a live verdict, and `prog_table`/`prog_main`/`prog_procs`, which the OCaml program printer reads on a round trip.

Nothing calls these on the CLI's analysis path, so a reading of the list as "the callable surface" alone would drop them, and the resulting break shows up not here but in an OCaml consumer, as an unbound value or a match on an abstract type.

`module_name Generated` puts the whole reachable program into one OCaml module rather than one module per contributing Isabelle theory. The alternative — letting the serializer split by theory — does not survive contact with this program: OCaml's single-file output emits modules in dependency order and cannot express a cycle,

and the theories here are mutually dependent at code level (the executable state is instantiated at the solver's own widening/narrowing classes, and the CFG-specific solver instantiation needs `cfg_node` back). Even a split that Isabelle accepts can fail later in `ocamlfind ocamlopt`, on a type-class dictionary field that module-signature inference does not expose across a boundary the unsplit default never had.

So the generated internals are monolithic, and this says so directly instead of arriving there by remapping every contributing theory onto one name by hand. Modularity, if wanted, belongs in the handwritten facade over `Generated`, `cli/voblint.ml`, which today includes it whole.

Two further modules are emitted regardless: `Bit_Shifts` and `Str_Literal` are HOL's own runtime support, injected as literal target code rather than generated from constants here.

The list below is grouped by what each name is *for*, because the groups answer different questions and only one of them is an operation.

- **Run.** `run_voblint`, and its traced twin `run_voblint_traced`. Everything a caller can ask the analyser to do goes through one of them; the untraced one reaches no hand-written target code.
- **Result.** `analysis_answer`'s cases, which a caller must tell apart, the projection `render_report` that turns an analysed report into displayable data, and the readers of that data: contexts, states, routes, checks, globals and diagnostics. The records reach OCaml abstract, readable only through their selectors, so a field added later cannot break a consumer that matched on field order. A caller maps `render_report` over the answer with `map_analysis_answer`, passing the value printer it wants: the CLI composes `string_of_abstract_value` with its decoding of ASCII symbol tokens.
- **Ask.** The values naming a request: which domain, which rule merges side-effected globals, which context policy. A caller constructs these, so their constructors are public.
- **Program.** What the frontend builds an `imp_prog` out of: the command and expression constructors, the program builder, and the numeral and character conversions that bridge OCaml's integers to HOL's.
- **Inspect.** Structural readers with no consumer on the CLI's analysis path. They exist for the OCaml program printer and the CFG regressions, and are listed apart so that reading the surface as "what the CLI calls" does not quietly drop them.
- **Trace.** The solver's events and the readers a run hands the tracer (`Voblint_CLI.Trace_Run`). The CLI's trace renderer matches on them, so they must be public.

export_code

```
— Run
run_voblint run_voblint_traced

— Result: answers, the report's rendering, contexts, states, routes, checks, globals, diagnostics, the analysis graph
Invalid_Activation Malformed_Program No_Answer Analysed map_analysis_answer
render_report
res_cfg res_contexts res_states res_joined res_routes res_checks res_globals res_diagnostics
res_graph graph_nodes graph_edges Local_Node Global_Node
Intra_Dep Enter_Dep Combine_Dep Continue_Dep Global_Read Global_Write
Context_Unit Context_Entry Context_Call_String
state_point state_context state_value state_checks state_diagnostics state_steps
Field_Store Field_Whole
route_point route_context route_callee route_targets
check_point check_label check_exp check_verdict
global_unknown global_state Global_Named Global_Seed
diagnostic_point diagnostic_occurrence diagnostic_obligation diagnostic_verdict
arithmetic_operation arithmetic_divisor
Check_Proved Check_Refuted Check_Unknown
Bot Lifted
cfg_entry cfg_node_list

— Ask: configuration of domains, global update rule, context, program globals
Analysis_Config
Program_Globals_Flow_Sensitive Program_Globals_Flow_Insensitive
Sign_Analysis Interval_Analysis Int_Analysis Refine_Fixpoint Refine_Once Refine_Never
Parity_Analysis Congruence_Analysis Order_Analysis
Globals_Join Globals_Per_Origin Globals_Warrow Globals_Warrow_Per_Origin Globals_Bounded_Narrowing
Ctx_None Ctx_EntryState Ctx_CallString

— Program: what the frontend builds an input out of
mk_program gproc_decl_ext
```

```
  SKIP Assign Seq gcom.If While Return Check gcom.Call
  N V Plus Minus Times Div Mod
  exp.Not And Or Less exp.Eq
  Statement FunctionEntry FunctionResult
  int_of_integer nat_of_integer integer_of_int integer_of_nat

  — Inspect: for the OCaml program printer and the CFG regressions, not the CLI
  declared_global_vars
  prog_table prog_main prog_procs
  prog_cfg cfg_intra_list cfg_calls_list prog_stmt_post_order
  EA_Nop EA_Assign EA_Special EA_Assume EA_AssumeNot EA_Body EA_Ret EA_Check
  CallEdge Nondet_Int

  — Trace: the events the solver reports and the readers a run hands the tracer
  Ev_Start Ev_Stop Ev_Query Ev_Query_Wpoint Ev_Iterate_From_Query Ev_Add_Infl Ev_Answer
  Ev_Query_Global Ev_Answer_Global Ev_Iterate Ev_Eq Ev_Rhs Ev_Still_Unstable Ev_Widen Ev_Sol
  Ev_Wpoint_Remove Ev_Wpoint_Clear Ev_Update Ev_Iterate_Changed Ev_Side Ev_Update_Global
    Ev_Destabilize
  Ev_Stable_Remove
  Ev_Route Trace_Printers Trace_Buffer Trace_Global Trace_Seed string_of_abstract_value Inl Inr

  in OCaml module_name Generated file_prefix "Voblint_Generated"

end
```

Appendix A

End-to-end certificate

A.1 Example_End_To_End_Certificate

```
theory Example_End_To_End_Certificate
  imports "Voblint_VIMP.VIMP_Notation" "Voblint_CLI.Analysis_Certified"
begin
```

One configuration, one program, no premise left standing

`run_voblint_source_sound` is stated for an arbitrary configuration and an arbitrary program, so on its own it says nothing about any particular run: a caller still owes the answer. This theory computes it, by evaluation, for one program at one configuration — the product domain, a call-string context of length one, and always-join named explicitly rather than defaulted — and reads the theorem's conclusion off the result.

Three dimensions at once is the point. A composite domain exercises the product's transfer; a call-string policy keys the solved table by call history; and naming a discipline the pairing does not default to shows the certificate travels with the solver actually run, not with the one the registry prefers.

The program

`bump` is called twice with different actuals, so a call string of length one files two activations of it under two contexts. The check reads the second result.

```
definition certificate_demo_prog :: imp_prog where
  "certificate_demo_prog = program {
     fun bump(n) { return n + 1; }
     fun main() {
       a = bump(1);
       b = bump(41);
       __voblint_check(0 < b);
     }
   }"

lemma certificate_demo_wf: "wf_program_compile_input_exec certificate_demo_prog"
  by eval
```

The configuration: Int at its most precise refinement, the joining update rule, and call strings of length one.

```
abbreviation certificate_analyses :: "analysis_domain list" where
  "certificate_analyses ≡ [Int_Analysis Refine_Fixpoint]"

abbreviation certificate_config :: analysis_config where
  "certificate_config ≡ Analysis_Config certificate_analyses Globals_Join (Ctx_CallString 1)
     Program_Globals_Flow_Sensitive"
```

What the configuration answers

Running `certificate_config` on `certificate_demo_prog` by evaluation: the single check at `Statement 4` comes back `Check_Proved`.

```
lemma certificate_demo_report:
  "(case run_voblint certificate_config
            certificate_demo_prog of
      Analysed res ⇒
        map (λchk. (check_point chk, check_verdict chk)) (report_checks res)
          = [(Statement 4, Decided Check_Proved)]
    | _ ⇒ False)"
  by eval
```

The endpoint below reads a result, so the answer is named rather than only inspected: `Analysed` is what this configuration returns on this program, and the check above is the check column it returns.

```
lemma certificate_demo_analysed:
  "∃res. run_voblint certificate_config
           certificate_demo_prog = Analysed res
       ∧ map (λchk. (check_point chk, check_verdict chk)) (report_checks res)
           = [(Statement 4, Decided Check_Proved)]"
  using certificate_demo_report
  by (cases "run_voblint certificate_config
               certificate_demo_prog")
     auto
```

One source run, constructed

The endpoint speaks about a source execution, so this one is built rather than assumed: start every variable at `0`, run both calls and the check, and end with `a = 2` and `b = 42`. Two facts about the source semantics are needed and are general rather than about this program — a check steps to `SKIP` without touching the store, and a call whose body is a single `Return` completes in three steps: enter the callee, publish the return value, and pop the frame writing the destination.

```
lemma pcompletes_Check: "pcompletes 𝒢 Π (VIMP_Proc.gcom.Check () l cond) s s"
  by (rule star.step) (rule pstep.Check, rule star.refl)

lemma pcompletes_Call_return:
  assumes p: "Π p = Some decl"
      and bd: "body decl = VIMP_Proc.gcom.Return () (Some e)"
      and arity: "length actuals = length (formals decl)"
      and dist: "distinct (formals decl)"
      and glob: "¬ 𝒢 ret_var"
  shows "pcompletes 𝒢 Π (VIMP_Proc.gcom.Call () dst p actuals) s
          (combine_assign dst
             (⟦e⟧e (bind_formals (formals decl) (map (λa. ⟦a⟧e s) actuals)
                       (enter_state 𝒢 s)))
             (combine_env 𝒢 s
                (bind_formals (formals decl) (map (λa. ⟦a⟧e s) actuals)
                   (enter_state 𝒢 s))))"
proof -
  let ?callee = "bind_formals (formals decl) (map (λa. ⟦a⟧e s) actuals) (enter_state 𝒢 s)"
  let ?entered =
    "VIMP_Proc.gcom.Seq (VIMP_Proc.gcom.Return () (Some e)) (VIMP_Proc.gcom.Restore ())"
  let ?unwinding = "VIMP_Proc.gcom.Seq (VIMP_Proc.gcom.Unwind ()) (VIMP_Proc.gcom.Restore ())"
  have c1: "𝒢, Π ⊢ (VIMP_Proc.gcom.Call () dst p actuals, s, [])
             →p (?entered, ?callee, [Frame s dst])"
    using arity bd dist p pstep_Call by fastforce
  have c2: "𝒢, Π ⊢ (?entered, ?callee, [Frame s dst])
             →p (?unwinding, ?callee(ret_var := ⟦e⟧e ?callee), [Frame s dst])"
    by (rule pstep.Seq2) (rule pstep.ReturnSome)
  have c3: "𝒢, Π ⊢ (?unwinding, ?callee(ret_var := ⟦e⟧e ?callee), [Frame s dst])
             →p (VIMP_Proc.gcom.SKIP (),
              combine_assign dst (⟦e⟧e ?callee) (combine_env 𝒢 s ?callee), [])"
    using pstep.UnwindAct [where 𝒢 = 𝒢 and Π = Π and dst = dst and fr = s and frs = "[]"
                           and s = "?callee(ret_var := ⟦e⟧e ?callee)"]
    by (simp add: glob)
  show ?thesis
    using c1 c2 c3 by (meson star.refl star.step)
qed

lemma certificate_demo_table:
  "prog_table certificate_demo_prog (STR ''bump'')
     = Some (⦇formals = [STR ''n''],
               body = VIMP_Proc.gcom.Return () (Some (Plus (V (STR ''n'')) (N 1)))⦈)"
  by (simp add: certificate_demo_prog_def prog_main_name_def)

lemma certificate_demo_main:
  "main_body (prog_table certificate_demo_prog)
     = VIMP_Proc.gcom.Seq
         (VIMP_Proc.gcom.Seq (VIMP_Proc.gcom.Call () (Some (STR ''a'')) (STR ''bump'') [N 1])
            (VIMP_Proc.gcom.Call () (Some (STR ''b'')) (STR ''bump'') [N 41]))
         (VIMP_Proc.gcom.Check () (0, 0) (Less (N 0) (V (STR ''b''))))"
  by (simp add: certificate_demo_prog_def main_body_def prog_main_name_def)

lemma certificate_demo_no_globals: "declared_global certificate_demo_prog = (λ_. False)"
  by (simp add: certificate_demo_prog_def fun_eq_iff)

lemma certificate_demo_call:
  "pcompletes (declared_global certificate_demo_prog) (prog_table certificate_demo_prog)
     (VIMP_Proc.gcom.Call () (Some x) (STR ''bump'') [N k]) s (s(x := k + 1))"
  using pcompletes_Call_return
          [where Π = "prog_table certificate_demo_prog" and p = "STR ''bump''"
             and decl = "(⦇formals = [STR ''n''],
                          body = VIMP_Proc.gcom.Return () (Some (Plus (V (STR ''n'')) (N 1)))⦈)"
             and e = "Plus (V (STR ''n'')) (N 1)" and actuals = "[N k]" and dst = "Some x"
             and 𝒢 = "declared_global certificate_demo_prog" and s = s,
           OF certificate_demo_table]
  by (simp add: certificate_demo_no_globals enter_state_def combine_env_def)

lemma certificate_demo_calls:
  "pcompletes (declared_global certificate_demo_prog) (prog_table certificate_demo_prog)
```

```
    (VIMP_Proc.gcom.Seq (VIMP_Proc.gcom.Call () (Some (STR ''a'')) (STR ''bump'') [N 1])
       (VIMP_Proc.gcom.Call () (Some (STR ''b'')) (STR ''bump'') [N 41]))
    (λ_. 0) ((λ_. 0)(STR ''a'' := 2, STR ''b'' := 42))"
proof -
  have c1: "pcompletes (declared_global certificate_demo_prog)
             (prog_table certificate_demo_prog)
             (VIMP_Proc.gcom.Call () (Some (STR ''a'')) (STR ''bump'') [N 1]) (λ_. 0)
             ((λ_. 0)(STR ''a'' := 2))"
    using certificate_demo_call [where x = "STR ''a''" and k = 1 and s = "λ_. 0"] by simp
  have c2: "pcompletes (declared_global certificate_demo_prog)
             (prog_table certificate_demo_prog)
             (VIMP_Proc.gcom.Call () (Some (STR ''b'')) (STR ''bump'') [N 41])
             ((λ_. 0)(STR ''a'' := 2))
             ((λ_. 0)(STR ''a'' := 2, STR ''b'' := 42))"
    using certificate_demo_call
            [where x = "STR ''b''" and k = 41 and s = "(λ_. 0)(STR ''a'' := 2)"]
    by simp
  show ?thesis by (rule pcompletes_Seq [OF c1 c2])
qed

lemma certificate_demo_run:
  "pcompletes (declared_global certificate_demo_prog) (prog_table certificate_demo_prog)
     (main_body (prog_table certificate_demo_prog)) (λ_. 0)
     ((λ_. 0)(STR ''a'' := 2, STR ''b'' := 42))"
  unfolding certificate_demo_main
  by (rule pcompletes_Seq [OF certificate_demo_calls pcompletes_Check])
```

The same run stopped one step short of the check: both calls are done and `Check` is the command about to run.

```
lemma certificate_demo_to_check:
  "declared_global certificate_demo_prog, prog_table certificate_demo_prog ⊢ (main_body (prog_table certificate_demo_prog),
λ_. 0, [])
     →_p* (VIMP_Proc.gcom.Check () (0, 0) (Less (N 0) (V (STR ''b''))),
      (λ_. 0)(STR ''a'' := 2, STR ''b'' := 42), [])"
  unfolding certificate_demo_main
  using psteps_Seq2 [OF certificate_demo_calls] by (meson Seq1 star.step star.refl star_trans)

lemma certificate_demo_init: "(λ_. 0) ∈ cinit_stores (declared_global certificate_demo_prog)"
  by (simp add: cinit_stores_def)
```

The same run in the graph, at a node this theory can name

`csim` is structural and not functional, so the endpoint below can only say *some* node represents where the run stopped. Naming the check's own node takes a second witness, built in the graph rather than in the source: the collecting semantics at `Statement 4` really does contain the store this program computes, which is what makes the verdict claim there bite.

Walking it takes one hop the collecting vocabulary does not close over step by step: `node_collect` is closed under intra edges but not under a call or a return alone, since a return has to pop the trace's *own* caller. A call and its matching return together do compose, and that is `node_collect_call_return_step` in `Voblint_CFG.Activation_Trace_Collect`; the two hops through `bump` below are its two instances.

```
lemma certificate_demo_entry_eval:
  "cfg_entry (prog_cfg certificate_demo_prog) = FunctionEntry (STR ''main'')"
  by (simp only: prog_cfg_def cfg_entry_compile_prog prog_main_name_def)

lemma certificate_demo_intra_eval:
  "intra (prog_cfg certificate_demo_prog) =
     {(FunctionEntry (STR ''bump''), EA_Body (STR ''bump''), Statement 0),
      (Statement 0, EA_Ret (Some (Plus (V (STR ''n'')) (N 1))) (STR ''bump''),
       FunctionResult (STR ''bump'')),
      (Statement 1, EA_Ret None (STR ''bump''), FunctionResult (STR ''bump'')),
      (FunctionEntry (STR ''main''), EA_Body (STR ''main''), Statement 2),
      (Statement 5, EA_Ret None (STR ''main''), FunctionResult (STR ''main'')),
      (Statement 4, EA_Check (0, 0) (Less (N 0) (V (STR ''b''))), Statement 5)}"
  unfolding prog_cfg_def by eval

lemma certificate_demo_calls_eval:
  "calls (prog_cfg certificate_demo_prog) =
     {(Statement 3, CallEdge (Some (STR ''b'')) [STR ''n''] [N 41],
       FunctionEntry (STR ''bump''), Statement 4),
      (Statement 2, CallEdge (Some (STR ''a'')) [STR ''n''] [N 1],
       FunctionEntry (STR ''bump''), Statement 3)}"
  unfolding prog_cfg_def by eval
```

A call to `bump` runs its whole body inside the callee's own activation: enter with the formal bound, take the body

edge, then the return edge, which is where the returned value is published.

```
lemma certificate_demo_bump_body:
  "intra_path (prog_cfg certificate_demo_prog)
     (FunctionEntry (STR ''bump''), (λ_. 0)(STR ''n'' := k))
     (FunctionResult (STR ''bump''), ((λ_. 0)(STR ''n'' := k))(ret_var := k + 1))"
proof (rule star_trans)
  show "intra_path (prog_cfg certificate_demo_prog)
          (FunctionEntry (STR ''bump''), (λ_. 0)(STR ''n'' := k))
          (Statement 0, (λ_. 0)(STR ''n'' := k))"
    by (rule intra_path_single [where a = "EA_Body (STR ''bump'')"])
       (auto simp: certificate_demo_intra_eval)
next
  show "intra_path (prog_cfg certificate_demo_prog)
          (Statement 0, (λ_. 0)(STR ''n'' := k))
          (FunctionResult (STR ''bump''), ((λ_. 0)(STR ''n'' := k))(ret_var := k + 1))"
    by (rule intra_path_single
          [where a = "EA_Ret (Some (Plus (V (STR ''n'')) (N 1))) (STR ''bump'')"])
       (auto simp: certificate_demo_intra_eval)
qed
```

The two store maps a call crosses, at this program's one procedure: entry resets the locals and binds the formal, and the return writes the published value into the destination while the caller keeps its own locals. No globals are declared, so nothing survives entry but the binding.

```
lemma certificate_demo_enter:
  "call_enter (declared_global certificate_demo_prog) (CallEdge dst [STR ''n''] [e]) s
     = (λ_. 0)(STR ''n'' := ⟦e⟧ₑ s)"
  by (simp add: certificate_demo_no_globals call_enter_CallEdge enter_binding_def
      enter_frame_def fun_eq_iff)

lemma certificate_demo_combine:
  "combine_collect (declared_global certificate_demo_prog) (Some x) s
     (((λ_. 0)(STR ''n'' := k))(ret_var := v)) = s(x := v)"
  by (simp add: certificate_demo_no_globals combine_collect_def combine_env_def
      ret_var_def fun_eq_iff)

lemma certificate_demo_reaches_check:
  "(λ_. 0)(STR ''a'' := 2, STR ''b'' := 42)
     ∈ 𝒞declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global certificate_demo_pro
(Statement 4)"
proof -
  have e0: "(λ_. 0) ∈ 𝒞declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global cert
                        (FunctionEntry (STR ''main''))"
    using node_collect_init
            [OF certificate_demo_init,
             where 𝒢 = "declared_global certificate_demo_prog"
               and g = "prog_cfg certificate_demo_prog"]
    by (simp add: certificate_demo_entry_eval)
  have s2:
    "(λ_. 0) ∈ 𝒞declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global certificate_
(Statement 2)"
    by (rule node_collect_intra_step [OF e0, where a = "EA_Body (STR ''main'')"])
       (auto simp: certificate_demo_intra_eval)
  have ce1: "(Statement 2, CallEdge (Some (STR ''a'')) [STR ''n''] [N 1],
              FunctionEntry (STR ''bump''), Statement 3)
               ∈ calls (prog_cfg certificate_demo_prog)"
    by (simp add: certificate_demo_calls_eval)
  have bd1: "intra_path (prog_cfg certificate_demo_prog)
               (FunctionEntry (STR ''bump''),
                call_enter (declared_global certificate_demo_prog)
                  (CallEdge (Some (STR ''a'')) [STR ''n''] [N 1]) (λ_. 0))
               (FunctionResult (STR ''bump''), ((λ_. 0)(STR ''n'' := 1))(ret_var := 1 + 1))"
    using certificate_demo_bump_body [where k = 1]
    by (simp add: certificate_demo_enter)
  have s3: "(λ_. 0)(STR ''a'' := 2)
              ∈ 𝒞declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global certificate
(Statement 3)"
    using node_collect_call_return_step [OF s2 ce1 bd1]
    by (simp add: certificate_demo_combine)
  have ce2: "(Statement 3, CallEdge (Some (STR ''b'')) [STR ''n''] [N 41],
              FunctionEntry (STR ''bump''), Statement 4)
               ∈ calls (prog_cfg certificate_demo_prog)"
    by (simp add: certificate_demo_calls_eval)
  have bd2: "intra_path (prog_cfg certificate_demo_prog)
               (FunctionEntry (STR ''bump''),
```

```
              call_enter (declared_global certificate_demo_prog)
                (CallEdge (Some (STR ''b'')) [STR ''n''] [N 41]) ((λ_. 0)(STR ''a'' := 2)))
             (FunctionResult (STR ''bump''), ((λ_. 0)(STR ''n'' := 41))(ret_var := 41 + 1))"
    using certificate_demo_bump_body [where k = 41]
    by (simp add: certificate_demo_enter)
  show ?thesis
    using node_collect_call_return_step [OF s3 ce2 bd2]
    by (simp add: certificate_demo_combine)
qed
```

What the solve owes

The solver's own run is the first. `dg_pipeline.root_query` has no code equation of its own — its type does not mention the domain, so the generator cannot see its sort hypothesis — and a call-string bound is a runtime argument, so no registration inlined it here. Unfolding it once is what the reflection below needs to reach `eval`.

```
lemma certificate_demo_solve_c:
  "TD_side_rule_Interp_solve_c Globals_Join
     (dg_pipeline.equations (mcp_comp (activation certificate_analyses))
        (mcp_init (activation certificate_analyses))
        Analysis_Buffer (Analysis_Global :: vname ⇒ (vname, cfg_node list) global_unknown)
        Activation_Seed (λ_. cs_route 1) []
        (λ𝒢 c. dg_spec_of c) (λ𝒢 d. Bot) (λ𝒢 p. [])
        (declared_global certificate_demo_prog) certificate_demo_prog)
     (dg_pipeline.root_query [] certificate_demo_prog) ≠ None"
  unfolding dg_pipeline.root_query_def by eval

lemma certificate_demo_terminates:
  "mcp_cs_rule.terminates certificate_analyses 1 Globals_Join
     (declared_global certificate_demo_prog) certificate_demo_prog"
  by (rule mcp_cs_rule.terminates_of_solve_c [OF certificate_demo_solve_c])
```

The theorems below owe no termination fact: an analysed answer exists only where the executable solve returned. The lemma above evaluates termination anyway, because the table lemma below is stated about this configuration's own solve.

What the printed verdict means at that node

The check the configuration lists carries its own expression, so it is worth reading back in full rather than by point and verdict alone.

```
lemma certificate_demo_report_full:
  "(case run_voblint certificate_config
            certificate_demo_prog of
      Analysed res ⇒
        map (λchk. (check_point chk, check_exp chk, check_verdict chk)) (report_checks res)
          = [(Statement 4, Less (N 0) (V (STR ''b'')), Decided Check_Proved)]
    | _ ⇒ False)"
  by eval

lemma certificate_demo_check:
  assumes ans: "run_voblint certificate_config
                  certificate_demo_prog = Analysed res"
  shows "∃chk. report_checks res = [chk] ∧ check_point chk = Statement 4
             ∧ check_exp chk = Less (N 0) (V (STR ''b''))
             ∧ check_verdict chk = Decided Check_Proved"
  using certificate_demo_report_full assms by (cases "report_checks res") auto
```

The table this configuration's own solve builds, and the fact that it is sound for this program: `analysis_report_of` builds its report from exactly this table.

```
abbreviation certificate_demo_table where
  "certificate_demo_table ≡
     mcp_cs_rule.result certificate_analyses 1 Globals_Join (declared_global certificate_demo_prog)
       certificate_demo_prog"

lemma certificate_demo_covered_table:
  "covered_table certificate_demo_prog certificate_demo_table
     (mcp_gamma_v (activation certificate_analyses))"
  by (rule activation_table.covered [OF mcp_cs_rule_table
        [OF wf_program_compile_input_exec_sound [OF certificate_demo_wf]
            certificate_demo_terminates]])
```

The endpoint's report conjunct, at the check's node rather than at an existential witness: the store the program

computes lies in the report's semantics at that node, that is, in the state filed there under the context the run's own call string produced.

```
lemma certificate_demo_report_covers_at_check:
  assumes ans: "run_voblint certificate_config certificate_demo_prog = Analysed res"
  shows "(λ_. 0)(STR ''a'' := 2, STR ''b'' := 42) ∈ 𝓡res (Statement 4)"
  using run_voblint_covers [OF ans] certificate_demo_reaches_check by blast
```

And the endpoint's verdict conjunct at the same node: the `Check_Proved` the report lists is not a claim about the analyzer's table but about every execution the semantics admits there.

```
lemma certificate_demo_verdicts_at_check:
  assumes ans: "run_voblint certificate_config
                  certificate_demo_prog = Analysed res"
      and mem:
        "s ∈ 𝓒declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global certificate_de
(Statement 4)"
  shows "s ∈ 𝓥res (Statement 4)"
  using run_voblint_collect_sound [OF ans] mem by blast

theorem certificate_demo_check_semantically_true:
  assumes ans: "run_voblint certificate_config
                  certificate_demo_prog = Analysed res"
  shows "∀s ∈ 𝓒declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global certificate_
(Statement 4).
           truthy (⟦Less (N 0) (V (STR ''b''))⟧e s)"
proof (intro ballI)
  fix s :: store
  assume mem:
    "s ∈ 𝓒declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global certificate_demo_p
(Statement 4)"
  obtain chk where r: "report_checks res = [chk]" and rp: "check_point chk = Statement 4"
    and re: "check_exp chk = Less (N 0) (V (STR ''b''))"
    and rv: "check_verdict chk = Decided Check_Proved"
    using certificate_demo_check [OF ans] by blast
  from certificate_demo_verdicts_at_check [OF ans mem] r rp re rv
  show "truthy (⟦Less (N 0) (V (STR ''b''))⟧e s)"
    unfolding verdict_stores_eq by simp
qed
```

Everything above, in one place and with every witness named. The generic endpoint is existential in the node and the frame stack because `csim` is structural; here the node is `Statement 4`, the store is the one the program computes, and the answer is the one `run_voblint` returns — so each conjunct of the headline theorem can be read off separately rather than inferred from an existential.

```
theorem certificate_demo_full_certificate:
  obtains res where
    "run_voblint certificate_config
       certificate_demo_prog = Analysed res"
    "map (λchk. (check_point chk, check_exp chk, check_verdict chk)) (report_checks res)
       = [(Statement 4, Less (N 0) (V (STR ''b'')), Decided Check_Proved)]"
    "pcompletes (declared_global certificate_demo_prog) (prog_table certificate_demo_prog)
       (main_body (prog_table certificate_demo_prog)) (λ_. 0)
       ((λ_. 0)(STR ''a'' := 2, STR ''b'' := 42))"
    "(λ_. 0)(STR ''a'' := 2, STR ''b'' := 42)
       ∈ 𝓒declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global certificate_demo_p
(Statement 4)"
    "(λ_. 0)(STR ''a'' := 2, STR ''b'' := 42) ∈ 𝓡res (Statement 4)"
    "(λ_. 0)(STR ''a'' := 2, STR ''b'' := 42) ∈ 𝓥res (Statement 4)"
    "truthy (⟦Less (N 0) (V (STR ''b''))⟧e ((λ_. 0)(STR ''a'' := 2, STR ''b'' := 42)))"
proof (cases "run_voblint certificate_config
                certificate_demo_prog")
  case (Analysed res)
  then have checks:
    "map (λchk. (check_point chk, check_exp chk, check_verdict chk)) (report_checks res)
       = [(Statement 4, Less (N 0) (V (STR ''b'')), Decided Check_Proved)]"
    using certificate_demo_report_full by simp
  have true_here:
    "truthy (⟦Less (N 0) (V (STR ''b''))⟧e ((λ_. 0)(STR ''a'' := 2, STR ''b'' := 42)))"
    using certificate_demo_check_semantically_true [OF Analysed]
          certificate_demo_reaches_check
    by blast
  show thesis
    by (rule that [OF Analysed checks certificate_demo_run certificate_demo_reaches_check
                      certificate_demo_report_covers_at_check [OF Analysed]
                      certificate_demo_verdicts_at_check
```

```
                      [OF Analysed certificate_demo_reaches_check] true_here])
qed (use certificate_demo_report in simp_all)
```

The endpoint, with nothing left to assume

Nothing is left to assume. The run is the one built above and the answer is the one computed above, so the endpoint's conclusion holds outright: the store the program actually ends with lies in the report's semantics at its node, in the state the always-join solve filed under that activation's own call string, and the `Check_Proved` check the configuration listed is true of it.

```
theorem certificate_demo_source_certified:
  "∃res v.
     run_voblint certificate_config
       certificate_demo_prog = Analysed res
   ∧ map (λchk. (check_point chk, check_verdict chk)) (report_checks res)
       = [(Statement 4, Decided Check_Proved)]
   ∧ certificate_demo_prog, (λ_. 0) ⊢ (SKIP (), (λ_. 0)(STR ''a'' := 2, STR ''b'' := 42), []) ▷ v
   ∧ (λ_. 0)(STR ''a'' := 2, STR ''b'' := 42)
       ∈ 𝒞declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global certificate_demo_p
v
   ∧ (λ_. 0)(STR ''a'' := 2, STR ''b'' := 42) ∈ ℛres v
   ∧ (λ_. 0)(STR ''a'' := 2, STR ''b'' := 42) ∈ 𝒱res v"
proof (cases "run_voblint certificate_config
                certificate_demo_prog")
  case (Analysed res)
  then have checks: "map (λchk. (check_point chk, check_verdict chk)) (report_checks res)
                       = [(Statement 4, Decided Check_Proved)]"
    using certificate_demo_report by simp
  from run_voblint_source_sound [OF certificate_demo_init certificate_demo_run Analysed]
  obtain v where "certificate_demo_prog, (λ_. 0)
                    ⊢ (SKIP (), (λ_. 0)(STR ''a'' := 2, STR ''b'' := 42), []) ▷ v"
    and "(λ_. 0)(STR ''a'' := 2, STR ''b'' := 42)
           ∈ 𝒞declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global certificate_de
v"
    and "(λ_. 0)(STR ''a'' := 2, STR ''b'' := 42) ∈ ℛres v"
    and "(λ_. 0)(STR ''a'' := 2, STR ''b'' := 42) ∈ 𝒱res v"
    by blast
  with Analysed checks show ?thesis by - (intro exI conjI; assumption)
qed (use certificate_demo_report in simp_all)
```

The check theorem, with nothing left to assume. The run stopped just before the check finds the report's check for the checked condition at a node that store reaches, and the `Check_Proved` in it is true of the store.

```
theorem certificate_demo_check_listed_sound:
  "∃res. run_voblint certificate_config
           certificate_demo_prog = Analysed res
     ∧ (∃chk ∈ set (report_checks res). check_exp chk = Less (N 0) (V (STR ''b''))
          ∧ (λ_. 0)(STR ''a'' := 2, STR ''b'' := 42)
              ∈ 𝒞declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global certificate
(check_point chk)
          ∧ check_verdict chk = Decided Check_Proved
          ∧ truthy (⟦Less (N 0) (V (STR ''b''))⟧e
                      ((λ_. 0)(STR ''a'' := 2, STR ''b'' := 42))))"
proof (cases "run_voblint certificate_config
                certificate_demo_prog")
  case (Analysed res)
  have checks: "map (λchk. (check_point chk, check_verdict chk)) (report_checks res)
                  = [(Statement 4, Decided Check_Proved)]"
    using certificate_demo_report Analysed by simp
  from run_voblint_check_sound
       [OF certificate_demo_init certificate_demo_to_check next_check.simps(1) Analysed]
  obtain chk
    where listed: "chk ∈ set (report_checks res)"
      and re: "check_exp chk = Less (N 0) (V (STR ''b''))"
      and mem: "(λ_. 0)(STR ''a'' := 2, STR ''b'' := 42)
                  ∈ 𝒞declared_global certificate_demo_prog,prog_cfg certificate_demo_prog,cinit_stores (declared_global certifi
(check_point chk)"
      and pr: "check_verdict chk = Decided Check_Proved
                 ⟶ truthy (⟦Less (N 0) (V (STR ''b''))⟧e
                             ((λ_. 0)(STR ''a'' := 2, STR ''b'' := 42)))"
    by blast
  have "(check_point chk, check_verdict chk) ∈ set [(Statement 4, Decided Check_Proved)]"
    unfolding checks [symmetric] using listed by auto
  then have "check_verdict chk = Decided Check_Proved" by simp
  with Analysed listed re mem pr show ?thesis by blast
```

```
qed (use certificate_demo_report in simp_all)

end
```