

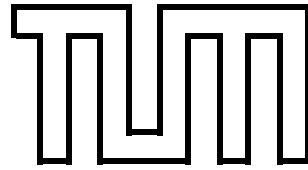
SCHOOL OF COMPUTATION, INFORMATION
AND TECHNOLOGY – INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

**Voblint: Towards a Verified Goblin-Style
Analysis Pipeline in Isabelle/HOL**

Manuel Lerchner



SCHOOL OF COMPUTATION, INFORMATION
AND TECHNOLOGY – INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Voblint: Towards a Verified Goblin-Style Analysis Pipeline in Isabelle/HOL

**Voblint: Auf dem Weg zu einer verifizierten
Analysepipeline im Stil von Goblin in
Isabelle/HOL**

Author:	Manuel Lerchner
Supervisors:	Alexandra Graß, M.Sc. Prof. Dr. Helmut Seidl
Examiner:	Prof. Dr. Francisco Javier Esparza Estaun
Submission Date:	01.12.2026

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 01.12.2026

Manuel Lerchner

Acknowledgements

I would like to thank my supervisors, Alexandra Graß and Helmut Seidl, for their guidance and support throughout this thesis. I am especially grateful to Alexandra for the many discussions, detailed feedback, and ideas that shaped the project. I greatly appreciated the freedom to explore the topic in depth and to follow directions that emerged along the way. I also thank Javier Esparza for agreeing to examine this thesis.

I am grateful to Michael Petter for helping to scope the project and shape its initial direction, and to Julian Erhard for ideas and feedback that helped align the formalization with Goblin’s design. I thank Sarah Tilscher for developing and maintaining the verified TD solver formalization on which this work builds.

I would also like to thank Tobias Nipkow for his courses *Semantics of Programming Languages* and *Functional Data Structures*, Helmut Seidl for *Program Optimization*, and Michael Petter and Julian Erhard for the practical course *Static Analysis: Automated Bug Hunting and Beyond*. These courses sparked my interest in formal verification and static analysis and ultimately led me to the topic of this thesis.

I am grateful to the members of the Chair for Programming Languages, Compiler Construction and Specification Formalisms. My connection to the chair began with the course *Functional Programming and Verification*, and it became an important part of my time at TUM. Working there twice as a teaching assistant and once as a tutor gave me insight into both teaching and research, and showed me how much I enjoy teaching formal methods.

I also acknowledge the many contributors to Goblin, whose work provided both a practical reference point and much of the inspiration for this thesis, as well as the developers of Isabelle and its surrounding open-source tooling. In particular, the Isabelle interfaces provided by AutoCorrode and Kevin Kappelmann’s MCP server made it practical to integrate AI-assisted development into this project.

Finally, I would like to thank my family and friends. My family gave me the freedom to focus fully on my work and encouraged me through every stage of my studies. My friends provided welcome distractions and regular reminders that there is a world outside Isabelle, and their company made the more intense periods of work much easier. Their patience and friendship have meant a great deal to me throughout my time at TUM.

Abstract

An abstract interpreter over-approximates a program’s executions, so any property that holds for the over-approximation holds in every run. Goblint, an abstract interpreter for C, builds a control-flow graph of the program, translates the graph into a system of equations over abstract states, and solves that system with a generic fixpoint solver [1], [2]. Its guarantee therefore depends on four conditions: a control-flow graph that simulates the source program, equations that over-approximate its executions in every calling context, a solver result that is a post-solution of the equations, and reported verdicts that follow from that result. For Goblint’s top-down solver, only the third condition had been verified [3], [4].

Voblint is an Isabelle/HOL formalization of a context-sensitive interprocedural analyzer. Its soundness proof covers all four conditions for VIMP, a small C-like language with global and local integer variables and recursive procedures with parameters and return values. The end-to-end soundness theorem states that every report the analyzer returns covers every store that a finite execution of the program reaches from an initial store, and that every **PROVED** or **REFUTED** verdict at the corresponding program point holds for that store. This is partial correctness, since termination of the solver is not proved. A check reported **DEAD** is also proved unreachable. At a point without an arithmetic warning or error, no divisor is zero, and a divisor reported as an error is zero wherever its point is reached. To the author’s knowledge, no prior mechanized analyzer has a soundness proof that connects a source semantics, through a context-indexed collecting semantics, to a side-effecting constraint system and its verified solver.

The main difficulty is modeling context sensitivity. A context-sensitive analyzer computes a separate abstract state for each program point and calling context. Soundness proofs usually compare them with the collecting semantics, the stores reachable at each program point. This semantics, however, merges the stores of all calls to a procedure, so it cannot say which stores a given context must cover. Voblint therefore introduces activation traces, inspired by the local traces of Schwarz et al. [5]. Each trace follows one procedure activation, and a context policy decides to which contexts it belongs. If the policy assigns at least one context to every call that an execution reaches, every execution belongs to some context, and the per-context sets of stores together equal the collecting semantics.

The proof is modular and follows Goblint’s architecture. Domains, context policies, analyses, and the solver each have their own proof obligations, and one soundness theorem composes them. A numeric domain only has to implement a shared interface and prove its abstract operations sound. The framework builds transfer functions from these operations and proves them sound once for all domains, as in Nipkow and Klein’s generic abstract interpreter [6, §13.5]. This theorem holds for every offered choice of analysis, context policy, and placement of program globals.

The analysis function is exported to OCaml, runs in a command-line tool and a [browser playground](#), and is tested on 386 regression fixtures. Proofs by evaluation show that the theorem is non-vacuous on concrete programs.

The analyzer runs inside unverified handwritten OCaml glue and driver code, including the parser and the rendering. This code, Isabelle’s code generator, and the OCaml and WebAssembly toolchains are trusted and tested conventionally using unit and regression tests. Because the theorem concerns VIMP, VIMP’s differences from C11 [7] are documented.

Use of Generative AI

The Leiden Declaration on Artificial Intelligence and Mathematics recommends that authors transparently disclose the automated tools they use, including large language models [8]. This section follows that recommendation.

Substantial parts of this work were carried out with generative AI tools, primarily Anthropic’s Claude, OpenAI’s ChatGPT and Codex, and Cursor. Under the author’s supervision, these tools contributed a large share of the implementation-level material. The author chose the research direction, selected and revised the designs, and approved the definitions and theorem statements.

In the Isabelle development, the tools generated many proofs, carried out refactorings across theories, and transformed tactic-style proofs and Sledgehammer output into structured Isar. They proposed designs and drafted planning documents for larger changes. For the thesis, they assisted with literature searches, checked citations and claims against cited sources, and drafted and revised the chapter structure, text, and figures.

Separately, Grammarly was used to check spelling and grammar in the thesis text. The Isabelle interfaces of AutoCorrode [9] and the Isabelle PIDE MCP server [10] served as proof-development infrastructure, and the proof work followed the human-guided workflow proposed by Kappelmann et al. [11].

The assistance also extended to design. To formalize calling contexts, the concrete semantics is based on traces instead of states, inspired by the local traces of Schwarz et al. [5]. Language models helped adapt the thread-local setting of that work to procedure activations. The requirements for the activation traces of Section 4.2.1 were evaluated and refined in collaboration with agents: how traces are generated, that a callee trace begins only at a call, that a trace retains its call history, and that calling contexts are derived from traces.

For the Isabelle development of the analysis framework, agents inspected Goblint’s OCaml sources to identify interfaces the formalization could approximate, such as local and global unknowns, unknowns indexed by node and context, and the enter/combine protocol at calls (Section 14.6). The source language and its compilation, the activation traces, the coverage contract, and the corresponding proofs have no direct counterpart in Goblint and were developed for this work.

A major part of the development effort concerned finding suitable definitions, interfaces, invariants, and locale boundaries rather than discharging individual proof steps. During the first weeks, a small prototype mirrored Goblint’s structure while leaving proof obligations as `sorry`. Once the definitions and locale boundaries had stabilized, many of the remaining proof obligations were short and were discharged with substantial agent assistance.

Throughout the project, the author also explored how such tools can be used responsibly and correctly in a formal development of this size. Agent output was treated as a proposal until Isabelle, a test or the author’s review confirmed it. This caution was necessary, since the assistants’ output was not reliable on its own. Agents proposed incorrect lemmas and, in one case, introduced a premise into the end-to-end soundness theorem that no program satisfied, so the theorem was vacuous until the solver run was redesigned. Audits of theorem statements and full batch builds exposed these errors. The non-vacuity instances of Section 13.1 now demonstrate that the premises of the end-to-end soundness theorem and its companions can be satisfied.

Agents also wrote much of the tooling around the formalization, such as the command-line tool, the project site, and the regression corpus with its ports of Goblin regression tests. They also wrote the checks that keep the theories, the generated material, the analyzer, the project site, and this thesis consistent (Appendix C). These checks proved especially useful because they replaced repeated manual inspection with conditions that every change had to satisfy, and they exposed errors that manual reading had missed.

Table 1 quantifies the recorded part of this assistance using session-log counts, following Bryant et al. [12, Section 5]. The counts measure interaction with the tools, not authorship or the proportion of the work attributable to them.

	Claude Code		Codex	
	main	subagent	main	subagent
sessions	110	451	121	47
assistant messages	52,374	22,755	15,063	484
tool calls	56,474	27,952	25,235	2,402
shell	27,525	17,568	—	—
Isabelle batch builds	726	97	361	1
Isabelle server (I/Q, PIDE)	24,149	5,032	18,251	1,247
file edits and writes	850	1,340	600	22

Table 1: Recorded automated-assistant activity during development, counted on 2026-09-25 from the local session logs (2026-08-24 to 2026-09-25 for Claude Code, 2026-06-27 to 2026-09-21 for Codex). No local logs are available for earlier Claude Code sessions, ChatGPT in the browser, or Cursor, so these are not counted. The author sent 2,434 messages to Claude Code and 1,227 to Codex. Shell edits count as shell calls. Interactions through the Isabelle server, including theory edits, count as server calls. Co-author lines in commit messages were not kept consistently and are not used as a measure.

Isabelle/HOL accepts every theorem in the development. Proofs by evaluation, used for some facts about fixed programs, additionally rely on Isabelle’s code generator (Figure 1.4). This establishes the stated propositions relative to their definitions and assumptions, but does not establish that those definitions faithfully capture the intended language or analysis. Definitions and theorem statements were therefore also checked against example programs, the regression corpus, and the documented differences between VIMP and C11, much as a human audit checks the statements of AI-generated formal proofs [13, Sec. “Is Lean reliable?”]. Related-work claims were checked against the cited sources, often with agent assistance and subsequent author review.

Responsibility for the final claims, framing, and treatment of related work lies with the author alone.

Reading This Thesis

The thesis can be read on paper, but it is designed to be read on screen. It closely follows a machine-checked Isabelle/HOL development, and most of its definitions and theorems link to the HTML rendering of the theories. A single [PDF export of the whole formalization](#) is available as well. The project site at manuellerchner.github.io/Voblint-Verification-Master-Thesis/ also hosts an [explainer](#) that animates several of the constructions, a [profiling page](#) with the full benchmark data and CPU profiles, and the [playground](#), which runs the analyzer in the browser on any VIMP program.

Each kind of object the text refers to has its own marker, and most markers are also links. An Isabelle name opens its definition, a program opens the playground, and a fixture opens its file in the repository.

Appendix A explains how to read the Isabelle listings, and Appendix B lists the mathematical notation shared by the theories and the text.

marker	meaning
<code>run_voblint</code>	Monospace text marks an Isabelle name, blue for a constant, green for a theorem, purple for a type, and teal for a locale. Every such name links to its definition in the rendered theories, and the thesis build fails if a link has no target.
<code>run_voblint_source_sound</code>	
<code>analysis_config</code>	
<code>activation_coverage</code>	
<code>INIT</code> , <code>TOTAL</code>	Teal small capitals mark an obligation of the coverage contract (Section 4.4), linked to the locale assumption that states it.
<code>FunctionEntry</code> , <code>while</code>	Purple sans-serif text marks a datatype constructor in notation, and bold maroon text a keyword of the source language VIMP.
Section title <code>thy</code>	A small blue thy box beside a heading marks a section that presents one theory, and opens that theory's page.
<code>(* src/... *)</code> <code>Isabelle</code>	A framed code block with a blue Isabelle tag is an Isabelle listing. Definitions and theorem statements are lifted verbatim from the theories, and every build checks them against their source. The first line names the source file, and the listing links to the declaration.
<code>VIMP</code> ↗	A framed code block with a maroon VIMP tag is a source program. The tag opens the program in the browser playground, with the analysis settings the text uses, so every analyzer run can be repeated and varied.
<code>PROVED</code> <code>REFUTED</code> <code>UNKNOWN</code> <code>DEAD</code>	A coloured monospace word on a tinted background is a verdict as the analyzer reports it (Section 10.5).
<code>intro-answers</code>	An underlined blue name marks a <i>claim</i> , a registered analyzer run. Every verdict and state of Voblint quoted in the text is read from a claim, which the build repeats, and the name opens the run in the playground.
<code>01-unbounded_...</code>	A blue monospace file name marks a fixture of the regression corpus, linked to its file in the repository.
<code>+</code> , <code>[1, 9]</code> , <code>1+2Z</code>	A light purple background marks an abstract value, as opposed to a concrete integer or set of stores.

Contents

Acknowledgements	i
Abstract	ii
Use of Generative AI	iii
Reading This Thesis	v

PART I PROBLEM AND FOUNDATIONS

1. Introduction	2
1.1. Sound static analysis and its verdicts	3
1.2. What existing analyzers and proofs leave open	4
1.3. Contributions	6
1.4. Scope and limits of the guarantee	7
1.5. Thesis outline along the proof chain	8
2. Abstract Interpretation and Solvers	11
2.1. Control-flow graphs and the collecting semantics	11
2.2. Lattices and fixpoints	12
2.3. Concretization and sound transfer functions	13
2.4. Analyses as constraint systems	15
2.5. Widening and narrowing	16
2.6. Context-sensitive analysis of procedures	17
2.7. Side-effecting constraint systems	17
2.8. The verified top-down solver and its guarantee	18

PART II WHAT MUST BE OVER-APPROXIMATED

3. VIMP and Its Control-Flow Graphs	21
3.1. VIMP syntax and well-formed programs	21
3.2. Small-step source semantics	24
3.3. The procedure-aware control-flow graph	26
3.4. Compiling VIMP to a graph	28
3.5. Compilation preserves source executions	30
3.6. Adequacy of VIMP relative to C	33
4. Traces and the Coverage Contract	34
4.1. Why calling contexts need activation traces	34
4.2. From graph executions to activation traces	35
4.3. Assigning calling contexts to traces	39
4.4. The coverage contract and its soundness	41

PART III BUILDING ANALYSES

5. Abstract Domains	44
5.1. Numeric domains and their concretization laws	45
5.2. Abstract states as sets of stores	45
5.3. Pointwise states and the unreachable state	46
5.4. Forward evaluation and truth tests	48
5.5. Answering checks with three-valued queries	48
5.6. Refining states at branch guards	50
5.7. The domain interface and its derived transfers	51
5.8. Carriers without least upper bounds	51
6. The Analysis Interface	53
6.1. Edge transfers and the enter/combine protocol	53
6.2. Local soundness laws of an analysis	54
6.3. Facts shared across program points	55
7. Cooperating Analyses	57
7.1. Queries, answers, and channels	58
7.2. Proving each analysis against true answers	59
7.3. Combining analyses into one sound analysis	60

PART IV COMPUTING SOUND RESULTS

8. Context-Indexed Equations and Routing	63
8.1. Context-indexed unknowns and call equations	63
8.2. Routing that agrees with the context policy	65
8.3. Post-solutions satisfy the coverage contract	67
9. Solving the Equations	68
9.1. Encoding the equations for the solver	68
9.2. Update rules for global unknowns	70
9.3. Solver certificates and partial correctness	71
10. From Solutions to Verdicts	73
10.1. The analyzer function and its report	73
10.2. Computing check verdicts from the report	74
10.3. Four inclusions from source execution to verdict	75
10.4. The end-to-end soundness theorem	76
10.5. What verdicts and diagnostics guarantee	78

PART V INSTANCES, IMPLEMENTATION, AND EVALUATION

11. Numeric and Relational Analyses	80
11.1. Four non-relational numeric domains	80
11.2. Combining the domains in a reduced product	81
11.3. Order constraints as a relational state	83
11.4. Proving order relations between variables	84

12. The Executable Analyzer	85
12.1. Finite representations of abstract states	85
12.2. Exporting the analyzer to OCaml and WebAssembly	86
12.3. Tracing solver steps without changing the proofs	87
12.4. Unverified parsing, rendering, and playground	88
12.5. Testing the unverified components	89
13. Evaluation of Strength and Precision	90
13.1. The soundness theorems are not vacuous	90
13.2. The obligations reject a real Goblint bug	90
13.3. Measuring precision on concrete programs	91
13.4. Comparing precision with Goblint	93
13.5. Running time	98
 PART VI CONTEXT AND CONCLUSION	
14. Related Work	100
14.1. Unverified analyzers for C	100
14.2. Verified analyzers in the CompCert line	100
14.3. Mechanized abstract interpreters	101
14.4. Verified solvers and termination results	103
14.5. Calling contexts and trace partitioning	103
14.6. Goblint and its local-trace semantics	104
14.7. Closest precedents and inherited parts	105
15. Conclusion	106
15.1. Lessons from the formalization	107
15.2. Relevance to Goblint and limitations	108
15.3. Extending Voblint towards C and Goblint	109
Bibliography	110
A. Reading the Isabelle Listings	118
B. Notation	120
C. Repository and Tooling	123
C.1. Repository contents and the vendored solver	123
C.2. OCaml packages on conda-forge	123
C.3. Derived files and checks	124
C.4. Maintaining the formalization: <code>isar-tools</code>	125

PART I

Problem and Foundations

1. Introduction

A static analyzer answers questions about every execution of a program at once: whether a divisor can be zero at some point, whether a program point is reachable at all, or whether a check always holds. Users act on these answers without running the program, so the answers have to be right. The analyzer, however, is itself a program, and its errors produce confident answers that real executions contradict. In Goblint [issue 1156](#), Goblint claimed that $c \% 2 == 1$ holds for $c \in \{-5, -7\}$, although the truncating remainder of C11 [7, §6.5.5p6] gives -1 for both values. The congruence domain had ignored the sign of the dividend, and [pull request 1161](#) fixed it [14]. An error of this kind is worse than imprecision. An imprecise answer leaves the check undecided, whereas a wrong answer gives a user who trusts it no reason to look for the error.

Fixing one such bug does not rule out the next. Ruling out all of them requires a proof that the analyzer’s answers hold for every execution of every program, and that proof must itself be checked. Formal verification states such a claim as a proposition over explicit definitions, and a proof assistant checks its proof [6]. Isabelle/HOL, one such proof assistant, constructs every theorem through a small trusted inference kernel, following the LCF approach [15]. Trusting a proof therefore means trusting this kernel and the system it runs on, not the tactics that produced the proof.

The kernel, however, checks a proof only against the definitions it is given. Whether those definitions model the intended objects, and whether the theorem expresses the intended claim, is left to human review. For example, a review of parts of an Isabelle/HOL formalization of Munkres’ general topology found definitions that are logically weaker than the textbook’s. They are harmless only because each theorem assumes the missing constraints again [12, §8.1]. Verification projects therefore state explicitly what lies outside the proof. The seL4 proof, for example, relates the kernel’s C implementation to an abstract specification in Isabelle/HOL and names the compiler, assembly code, boot code, cache management, and hardware as assumptions. It also assumes that its formal model of C reflects the C standard [16].

This thesis builds Voblint, an executable analyzer whose answers are proved to hold for every execution of the analyzed program. Like these projects, it names the assumptions under which its proof holds. Voblint draws its boundary at the source language. Its end-to-end soundness theorem starts from *source executions*, the executions of a program under the small-step semantics [pstep](#) of Voblint’s input language VIMP. Every later representation, from the control-flow graph to the solver’s result, is connected to source executions by proof. Of the semantics, only [pstep](#) therefore needs an adequacy argument instead of a proof. That argument states which fragment of C it models, where it departs from C11 (Section 3.6), and why no existing verified semantics can be used instead.

1.1. Sound static analysis and its verdicts

Software is usually checked by testing. A failing test demonstrates a bug, but a passing test suite has observed only finitely many executions [17, §1.4.1]. A static analysis reasons about all executions instead. At each program point, it computes an *abstract state*. An abstract state is a finite representation of a set of program states, for instance, an interval that contains every value a variable takes whenever execution reaches the point. The analysis is *sound* if the abstract state contains every state that occurs in a real run, although it may contain more. Such a result is an *over-approximation*. If a sound abstract state contains no state that violates a check, no execution violates it, tested or not (Figure 1.1). The converse does not hold. An abstract state can contain violating states that no execution reaches, and the analysis then cannot decide the check, even when every execution satisfies it.

Suppose the analysis computes $x \in [43, +\infty)$ at a point that divides by x . Every execution that reaches this point has $x \geq 43$, so the division is safe. If it computes only $x \in [0, 100]$, the abstract state contains $x = 0$, and the analysis cannot prove the division safe. However, every real execution may still have $x \geq 43$, for instance, because of a relation to another variable that an interval cannot express.

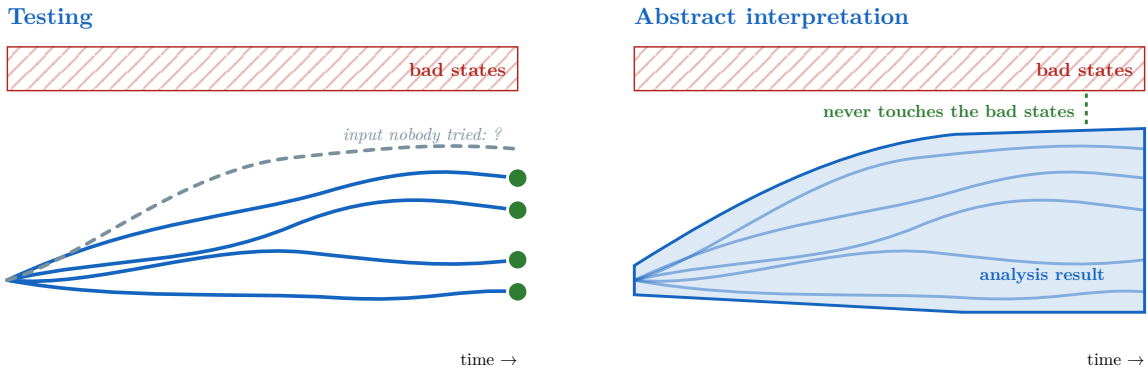


Figure 1.1: Testing and abstract interpretation, schematically. Each curve is one execution, the horizontal axis is time, and the vertical axis the program state. Left: tests observe only the runs they execute, and a run on an input nobody tried (dashed) stays unknown. Right: a sound analysis result (shaded) contains every run, tried or not, and may also contain states that no run reaches. Because it does not overlap the bad states, no execution can reach them.

For a general-purpose language, this incompleteness is unavoidable. By Rice’s theorem, no algorithm decides a nontrivial semantic property for every program of a Turing-complete language [17, §1.3.3], [18]. A sound analysis may therefore raise false alarms and leave checks undecided, but it never makes a false claim. It is useful as long as it decides the checks of interest often enough [17, §1.3.5]. Since every sound analysis over-approximates the same executions, analyses can specialize. One tracks intervals, another parities, and another relations between variables. If any one of them, or their combination, proves a check, the check holds in every run that reaches it. An analysis can also separate the activations of a procedure by their *calling context*, an abstraction of how the call was reached, such as the call site or the abstract state at entry. It then keeps one abstract state per program point and context, and it may analyze one call in several contexts, for

instance, one per abstract state that reaches the call. A richer domain or more contexts can decide more checks, but they also make the analysis more expensive. If there are many contexts, the analysis may not finish in practice.

Voblint answers every check that the user writes into the source program as `__voblint_check(e)`. It also warns at every division and remainder where it cannot exclude a zero divisor. Figure 1.2 shows a run in the [browser playground](#) together with what each answer guarantees. **PROVED** and **REFUTED** are the *definite* verdicts, and **UNKNOWN** claims nothing. Only **DEAD** makes a claim about reachability, and **REFUTED** is not a verified counterexample. Voblint offers several abstract domains and context policies (Chapter 11), and the run shown uses intervals and the unit context policy.

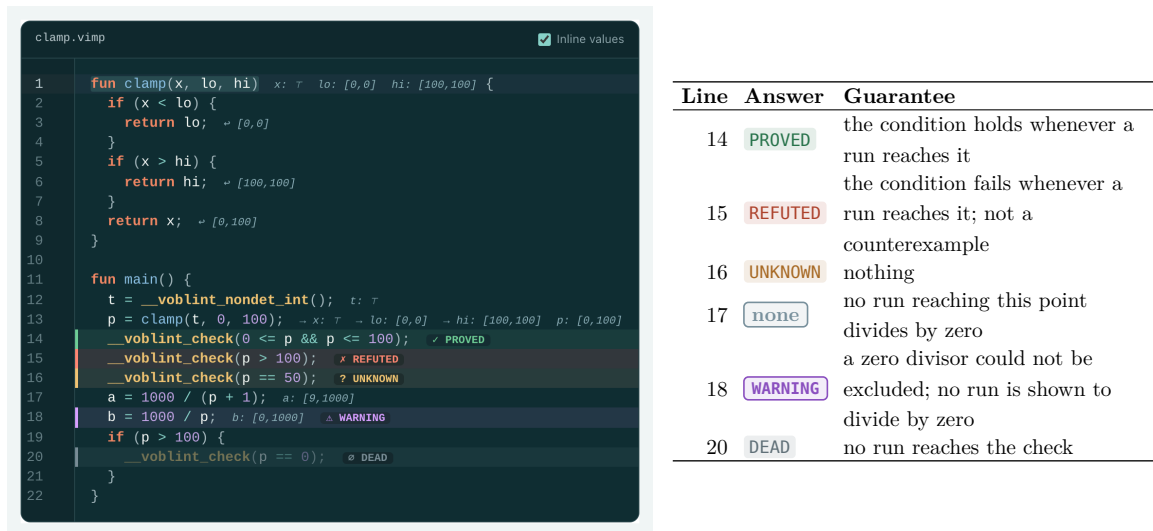


Figure 1.2: One analyzer run on a clamp function, as the browser playground shows it, with the interval domain and the unit context policy. The analysis covers every value of `t` at once. The screenshot opens the run in the playground (claim [intro-answers](#), fixture [24-site-figures/precision/48-clamp every answer.vimp](#)).

1.2. What existing analyzers and proofs leave open

Most abstract interpreters in practical use have no machine-checked soundness proof and rely on testing instead. Livshits et al. know of no realistic whole-program analysis tool that does not purposely make unsound choices, and they observe that virtually all published whole-program analyses are unsound [19, p. 44]. Most of these analyzers have a sound core that over-approximates most language features and models all their possible behaviors. It instead under-approximates a few features that experts in the area know well, such as reflection in Java, `eval` in JavaScript, or `setjmp/longjmp` in C. The analysis then pretends that some possible behaviors of these features cannot happen, for instance, that `eval` does nothing, so its result can miss executions. The authors call such analyses *soundy*. They treat this as an engineering choice for precision and scalability, and they ask the authors of such analyses to state explicitly the nature and extent of the unsoundness [19, p. 45]. Goblint, for instance, documents such assumptions. It assumes that pointer arithmetic does not overflow and, by default, that array reads stay in bounds, and it notes that this list is likely incomplete [20].

A documented assumption tells the user where a soundness claim ends. Within that boundary, however, the claim still rests on an implementation that no one has proved correct. Even the analyzers in practical use that are designed to be sound are tested rather than proved (Section 14.1), and testing an analyzer is incomplete in the same way as testing any other program (Section 1.1). Since its first participation in SV-COMP in 2021, a competition on benchmark programs with known verdicts, Goblint has never produced a false negative on the competition’s benchmark suite [2]. Still, its issue tracker lists more than a hundred [issues labeled as unsound](#) that have been opened since 2021. The two observations do not contradict each other. A test finds an error only when someone thinks of exactly the scenario that triggers it, and every other error waits until it misleads a user who then reports it. The Goblint defect at the start of this chapter needed a negative dividend in the congruence domain and was reported only after Goblint had claimed a false result. A proof does not depend on the right scenario. If every abstract operation is proved sound for every input, a congruence remainder that ignores the sign of the dividend cannot pass, and errors of this kind are ruled out completely.

A proof about single operations, on paper or in a proof assistant, however, protects the user only if a theorem connects these operations to the tool the user runs. Between the operations and the reported verdict, the analyzer compiles the program, assigns calling contexts, and solves equations, and each of these steps could lose executions. CompCert shows that such a theorem is possible for a realistic tool. It proves that its compiled code behaves as the source semantics specifies, and it lists the components that remain trusted [21]. Verasco carries the approach to a static analyzer for C [22], and several other abstract interpreters have machine-checked soundness proofs as well. None of the compared works, however, combines recursive procedures, calling contexts, and a verified solver (Section 14.3). Goblint’s architecture uses the first two and a generic solver, whose top-down algorithm has since been verified. It analyzes recursive programs and keeps one abstract state per program point and calling context.

Goblint defines its analyses independently of the solvers that compute their results, and Voblint follows this architecture [1], [2]. The interface between the two is a side-effecting constraint system [2], [23], [24]. Such a system relates unknown abstract states by inequalities, and the inequality of one unknown may also contribute to other unknowns. Because this architecture is modular, it also suits a proof. The soundness argument about programs needs only one fact about the solver, namely that its result satisfies the inequalities. The argument therefore holds for every solver that guarantees this fact. Side effects let an analysis keep flow-insensitive information, such as the values of global variables, in one unknown to which every program point contributes. They also serve calling contexts. Under some context policies, a callee’s contexts become known only while the system is solved, so each call contributes its entry state to the callee instead of the callee listing its callers in advance (Section 2.7). For the solver itself, a proof already exists. The top-down (TD) solver that Goblint uses, including its extension to side effects, has been formalized and proved partially correct in Isabelle/HOL [3], [4]. When it terminates, it returns values that satisfy the inequalities of the queried unknown and of every unknown it transitively reads ([part_post_solution](#)). Whether the equation system describes the program, and whether the verdicts read off its solution hold, lies outside that theorem.

The solver theorem, however, says nothing about programs. To connect a program to its equation system, every abstract state must be compared with the concrete stores it

describes. Calling contexts make this comparison hardest, since the per-context abstract states of an analyzer have no counterpart in an ordinary concrete semantics. Suppose `main` calls `bump(5)` and `bump(4)`, and `bump` returns its argument plus one. An analysis whose contexts record the argument at entry claims the result 6 in the first context and 5 in the second. The collecting semantics, the set of stores that executions reach at each program point, holds both results at the point where `bump` returns. It forgets how each store was reached, so it does not say which store belongs to which context, and the claim for one context cannot be checked against it. To state such claims, the concrete semantics must be strengthened so that it records the contexts of every *activation*, one execution of a procedure from its call to its return. Voblint’s activation traces record each activation together with the activation that called it, and a context policy reads the contexts off this chain. With such a semantics, every step from source executions to the verdicts becomes an inclusion between sets of concrete stores, and a proof can check each of them.

1.3. Contributions

The contribution of this thesis is the Isabelle/HOL formalization of Voblint and its machine-checked soundness proof, available at github.com/ManuelLerchner/Voblint-Verification-Master-Thesis and archived on Zenodo [25]. The proof closes the gaps of Section 1.2 with a chain of inclusions at every program point, from source executions to the reported verdicts (Figure 1.3). Each step may add stores that no execution reaches, which costs precision, but it must not lose a reachable store. Errors such as a compiler that drops an edge or a return to the wrong caller would lose such a store, and the proof excludes them.

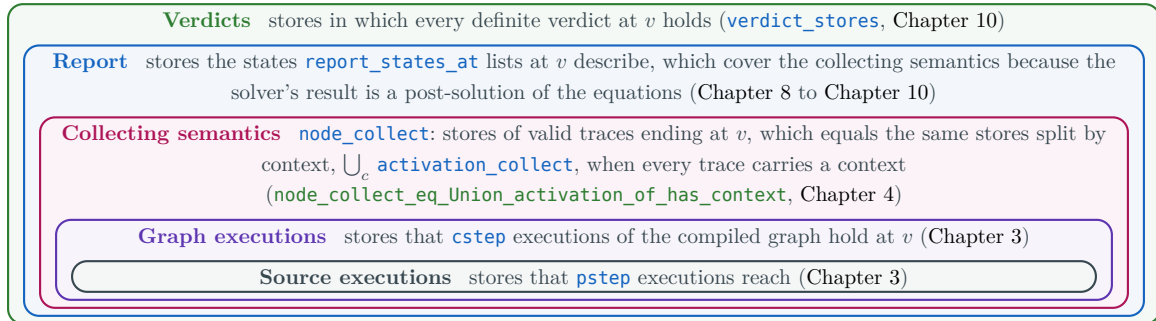


Figure 1.3: Soundness at a program point v as nested sets. A valid trace records, besides the stores of a graph execution, which call created each activation, and every graph execution is represented by one. Splitting the node collecting semantics by context loses no store. The end-to-end soundness theorem chains these inclusions for every report the analyzer returns (`run_voblint_source_sound`, Chapter 10).

Voblint closes these gaps by combining existing ideas, some of them already formalized and others so far only described on paper or implemented. Its overall design mirrors Goblint’s architecture inside Isabelle/HOL, without formalizing Goblint’s implementation or C (Section 14.6). To the author’s knowledge, no prior mechanized analyzer connects a source semantics, through an activation collecting semantics, indexed by context, to a side-effecting constraint system and its verified solver (Section 14.7). Within this design, the formalization makes five contributions.

- *End-to-end soundness.* The theorem connects the obligations of single operations to the tool the user runs. Whenever the analysis function `run_voblint` returns a report, its definite verdicts (`PROVED` and `REFUTED`) hold for every source execution from an initial store with zeroed globals that reaches the corresponding program point, under every analysis configuration the function offers (`run_voblint_source_sound`, Section 10.4). Companion theorems justify `DEAD` and the absence of arithmetic warnings (Chapter 10).
- *A concrete semantics of calling contexts.* Activation traces give the per-context states of an analysis a concrete counterpart. Each trace follows one procedure activation and records the activation that called it, so the contexts of a call can be read off the execution. Splitting the concrete stores by context loses no execution (Section 4.4), and every report covers each context separately (`run_voblint_context_covers`). One call may have several contexts, which lets an analysis split it into cases, for instance one per path-sensitive state.
- *Separately verified components.* Each component discharges local soundness obligations, and generic theorems compose them into the end-to-end guarantee for every configuration. Domains prove their primitive operations sound, context policies prove that their contexts agree with the concrete semantics, cooperating analyses prove conditions on the answers they exchange (Chapter 7), and the solver enters the proof only through proved facts about its result (Section 9.3).
- *An executable analyzer.* Isabelle’s code generator [26] exports `run_voblint`, the constant the soundness theorem is about, to OCaml. A command-line tool and a browser playground run it on VIMP programs. The playground also draws the control-flow graph and replays the solver’s steps.
- *Needed assumptions and nonempty results.* The kernel cannot tell whether a theorem assumes too much or concludes too little. Further theorems therefore show that the analyzer gives definite verdicts on concrete programs (Section 13.1). Dropping the obligation `TOTAL` admits answers that concrete executions contradict (Section 4.4), and one domain obligation rejects a defect that occurred in Goblint (Section 13.2). Concrete programs show what each mechanism gains (Section 13.3).

1.4. Scope and limits of the guarantee

Not every part of the delivered analyzer is covered by the proof (Figure 1.4). *End-to-end* means that the proof reaches from source executions under `pstep` to the result of the analysis function, while the parser, the handwritten OCaml for printing, and the trusted foundation lie outside it. Nor does the proof cover Goblint itself, only the parts of its architecture that Voblint isolates, over a small language with recursive procedures.

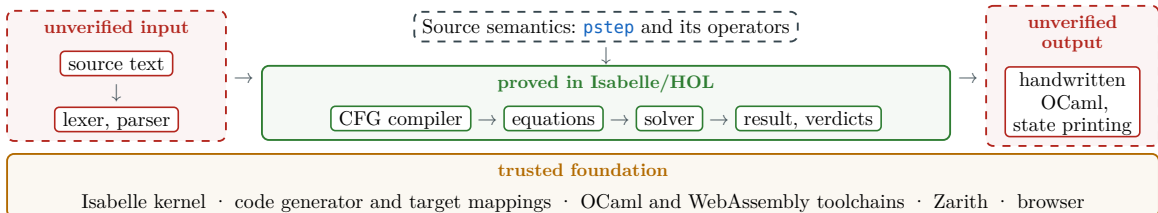


Figure 1.4: Where the proof starts and stops, for accepted programs whose solve terminates. The adequacy of `pstep` is argued (Section 3.6).

The theorem has the following limits.

- *Language and properties.* VIMP has no pointers, heap, or threads. Its integers are unbounded, division by zero is defined, and a callee’s locals start at 0. Therefore, a verdict does not always hold for a C program with the same text (Section 3.6). Voblint covers checks (`__voblint_check`) and division by zero, while Goblint also checks overflows, data races, memory safety, memory leaks, and program termination [27, §3.3].
- *Adequacy and trusted components.* It is argued, but not proved, that `pstep` models the intended language (Section 3.6). The parser, the code generator, and the target toolchains lie outside the proof (Figure 1.4).
- *Partial correctness.* The theorem covers every report the analyzer returns. Termination of the solve is not proved for every program (Section 9.3).
- *Precision and cost.* No completeness or general precision ordering is proved (Section 13.3). The executable uses the same data representations as the proofs and was not optimized beyond one fixed performance defect (Section 12.2). Running time is measured only to show that the analyzer finishes in acceptable time on the corpus (Section 13.5).
- *Goblint and evidence.* The correspondence to Goblint is architectural, and no theorem transfers to Goblint’s implementation (Section 14.6). The regression corpus is small and was written or adapted for this work (Section 12.5).

1.5. Thesis outline along the proof chain

The nested sets of Figure 1.3 say what the theorem claims. Figure 1.5 shows how the proof establishes it. Each line groups the results of one layer of the development, and each interchange is a theorem that uses the results of the lines that meet there. The labels name the theorems, and the caption gives the chapter that presents each line.

The proof map selects the results that connect the semantic layers. Figure 1.6 shows the surrounding formal development as a treemap of exported Isabelle entities. It groups the declarations by session and theory, including facts generated by Isabelle’s definition packages. The area of a block therefore measures its share of the exported corpus, not the difficulty of its proofs. The interactive view links each declaration to its documentation.

Part I consists of this introduction and Chapter 2, which recalls abstract interpretation, constraint systems, calling contexts, side-effecting constraint systems, and the verified TD solver.

Part II fixes what an analysis must over-approximate: source programs and their graphs (Chapter 3) and activation traces with the coverage contract (Chapter 4). The contract reduces soundness to five local obligations about a *cover*, one set of stores per node and context, and every later part works toward a cover that meets them.

Part III states what domains, analyses, and cooperating analyses must prove (Chapter 5 to Chapter 7). A domain proves its primitives sound once, an analysis supplies abstract transfer functions with the analysis contract, and cooperating analyses prove conditions on the answers they exchange.

Part IV builds the equations, solves them, and composes the end-to-end soundness theorem (Chapter 8 to Chapter 10). Every post-solution of the equations induces a cover that meets the contract (Chapter 8), the verified TD solver returns such a post-solution whenever it terminates (Chapter 9), and the report read off it is sound for every source execution (Chapter 10).

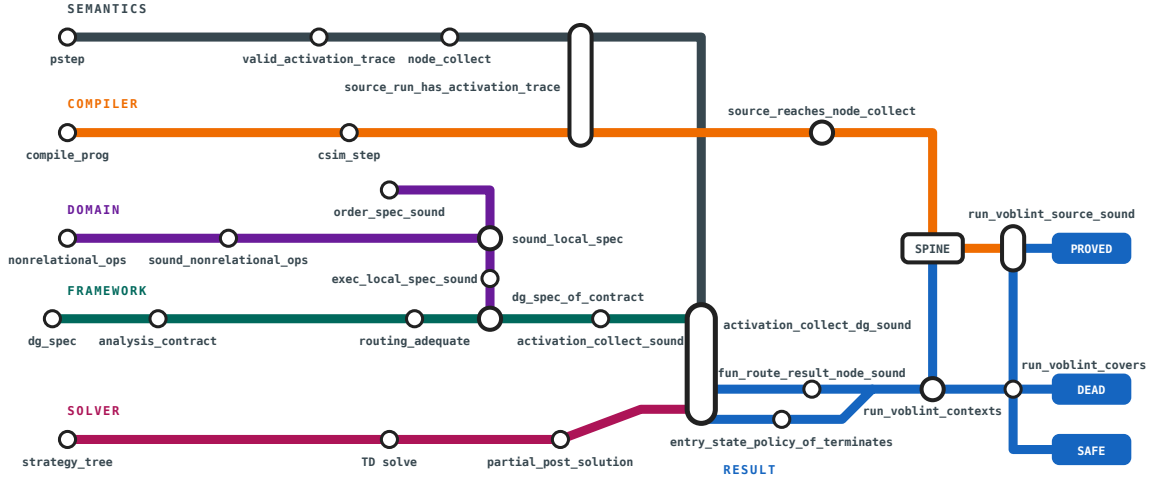


Figure 1.5: A map of the proof with its interior definitions omitted. The [project site](#) shows it interactively, with every station linked to its theory. Each line groups the results of one layer: **Semantics**, the source and trace semantics (Part II, Chapter 3 and Chapter 4), **Compiler**, the compiler and its simulation (Part II, Chapter 3), **Domain**, what a domain proves and how the analyses cooperate (Part III, Chapter 5 to Chapter 7, and the order analysis of Chapter 11), **Framework**, the analysis contract, routing and coverage (Chapter 4, Chapter 6 and Chapter 8), **Solver**, the verified TD solver (Part IV, Chapter 9), and **Result**, from the solved equations to the verdicts (Part IV, Chapter 8 and Chapter 10). Wide markers are interchanges, where a theorem uses the results of the lines that meet. The result line leaves the central interchange, splits for functional and entry-state context policies, and rejoins at one common theorem. From there the context-free coverage leads directly to the terminals for **DEAD** (`run_voblnt_dead_check_unreached`) and for **SAFE**, which says that a division without a **WARNING** never divides by zero (`run_voblnt_arithmetic_safe`). The terminal for **PROVED** (`run_voblnt_check_sound`) goes through the end-to-end soundness theorem `run_voblnt_source_sound`, which also uses the compiler line to place a source execution at a graph node. A second branch uses the compiler line in the same way for the chain that keeps the contexts.

Part V instantiates the theorem (Chapter 11), follows `run_voblnt` to the delivered tools and their trust boundary (Chapter 12), and evaluates it (Chapter 13).

Part VI compares Voblnt with prior work (Chapter 14) and concludes (Chapter 15). Appendix A explains the Isabelle syntax of the listings, Appendix B lists the notation, and Appendix C describes the repository, its checks, and the source tooling built for this work.

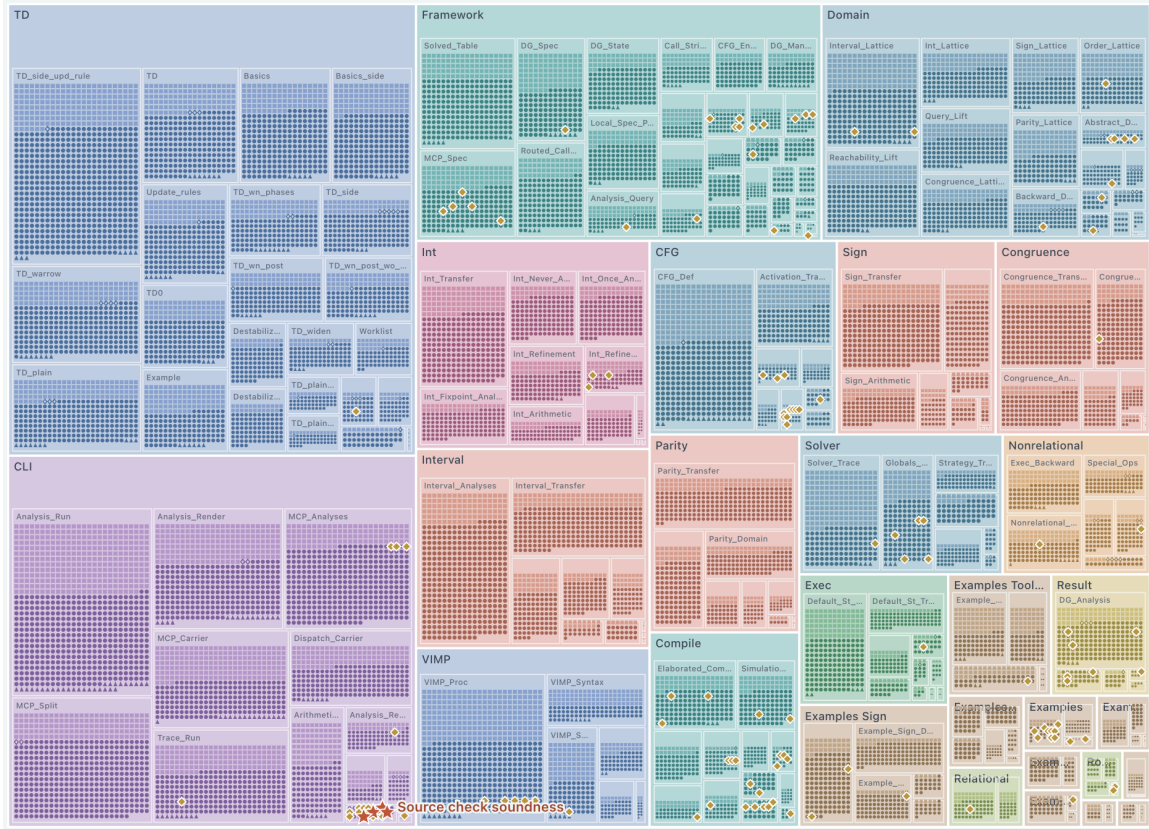


Figure 1.6: The formalization’s exported declarations, captured from the [interactive treemap](#). Blocks group sessions and theories; their areas follow entity counts. Colors identify session dependency layers. Gold diamonds mark thesis citations, and red stars mark selected source-level soundness results. This is a corpus overview; it does not depict proof dependencies.

2. Abstract Interpretation and Solvers

Section 1.2 ended with a plan. Each step from source executions to the verdicts should relate two sets of concrete stores, so that a proof can check that the step loses nothing. This chapter introduces the ideas needed to state these steps. It first describes what a program can actually do, then how an analysis describes the same behavior more coarsely, and finally how a solver computes this description. A loop that counts from 0 to 5 is the running example for all three.

2.1. Control-flow graphs and the collecting semantics

An analysis describes the states a program can be in at each of its points, so the points are made explicit first. A *control-flow graph* (CFG) has the program points as nodes and an edge for every way control can pass from one point to the next, labeled with the assignment or condition executed on the way. Figure 2.1 shows the graph of a loop that counts i from 0 to 5, the running example. The loop becomes a cycle through h , b , and t , and its *loop head* h is the node through which control enters and exits the cycle.



Figure 2.1: The counting loop and its control-flow graph (schematic). Nodes are program points: h is the loop head, b the start of the body, t the point after the increment, and e the point after the loop. Edges carry the assignment or the condition that leads from one point to the next. The edge from t back to h has no effect.

The state of an execution is a *store*, which maps each variable to its value, here the value of i . An execution starts at the entry of the graph in an initial store and follows a path, updating the store at every edge. The counting loop reaches h once before the first iteration and again after each run of the body, so i takes exactly the values $0, 1, \dots, 5$ there. For every program point, collecting the stores of all executions that reach it gives the *collecting semantics* $\mathcal{C}$ [28, §4], [29, §3.4], which maps every program point v to the set $\mathcal{C}(v)$ of stores reached there. For programs with procedures, Chapter 4 defines Voblint’s version over activation traces ([node_collect](#)).

The collecting semantics is the exact answer an analysis approximates. An analysis is *sound* if, at every program point, its result describes every store that the collecting semantics gives there. It may describe more stores, which costs precision, but never fewer.

2.2. Lattices and fixpoints

At h , the intervals $[0, 5]$ and $[0, 10]$ both contain every value i takes there (the collecting semantics $\{0, \dots, 5\}$), but $[0, 10]$ also admits $i = 7$, which no execution reaches. A description is *more precise* than another if it admits fewer states. Ordered sets make this comparison formal. In an ordered set $(\mathbb{D}, \sqsubseteq)$ (**order**), $a \sqsubseteq b$ means that a is at least as precise as b [6, Def. 10.23]. For sets of concrete states, the order is inclusion. Isabelle writes the order as $\leq$, and the listings taken from the theories keep this symbol.

Two descriptions reaching a point along different paths must be joined into one that covers both. Their *join* $a \sqcup b$ is the least upper bound, the most precise element above both (**semilattice_sup**). At the loop head, the value before the loop and the value after the body are joined this way. Dually, the *meet* $a \sqcap b$ is the greatest lower bound. An analysis uses it at a branch to restrict the incoming value to the values the condition admits. If the analysis knew only $[0, 10]$ at h , entering the body would restrict it to $[0, 10] \sqcap [-\infty, 4] = [0, 4]$. Soundness does not need this refinement, since $[0, 10]$ itself is sound at b , but it makes the result more precise, and Goblint and Voblint both refine at branches. The least element $\perp$ lies below every element, and a top element $\top$, if one exists, lies above every element. In abstract interpretation, $\perp$ describes no states and $\top$ describes all of them. A *complete lattice* (**complete_lattice**) has a join $\bigsqcup X$ and a meet $\bigsqcap X$ for every subset X [6, Def. 13.1], [29, Def. 2.5].

Voblint does not need a complete lattice. Its solver requires only a join semilattice with a least element (**bounded_semilattice_sup_bot**) that carries the widening and narrowing operators of Section 2.5. The solver joins two values at a time, whenever a new value reaches an unknown, so joins need to exist only for pairs. Joins of infinite sets would not help an executable analyzer anyway, since it computes only with finite representations.

A numeric domain adds two operations to this structure. Its top element describes a variable about which nothing is known, for instance, a value read from input. Refinement at a branch adds an intersection, which does not have to be the exact meet. It must keep every value that both operands admit, and its result must lie below both operands (**sound_intersection**, Section 5.6). An intersection that admits more values than both operands share loses precision, but not soundness.

An analysis computes the description at a point from the descriptions at its predecessors, by a function f on $\mathbb{D}$, and its result must be stable under f . A *fixpoint* of f is an element with $f(d) = d$, and a *post-fixpoint* is an element with $f(d) \sqsubseteq d$, so applying f once more stays below d . The classical theory of least fixpoints requires f to be *monotone* (**mono**): $a \sqsubseteq b$ implies $f(a) \sqsubseteq f(b)$ [6, Def. 10.26], [29, Def. 2.9], so more precise input never yields less precise output. By the Knaster–Tarski theorem, a monotone f on a complete lattice has a least fixpoint $\text{lfp } f$, the meet of all post-fixpoints [30, Thm. 1] (**lfp**, **lfp_unfold**). On an order of finite height, the *Kleene iterates* $\perp, f(\perp), f(f(\perp)), \dots$ stabilize at $\text{lfp } f$ after finitely many steps [31, §1.5] (**lfp_kleene_iter**). An analysis, however, does not have to compute $\text{lfp } f$. As long as f is sound, every post-fixpoint that includes the initial states is already a sound result (Section 2.3). The verified solver that Voblint reuses therefore does not assume monotonicity (Section 2.5).

2.3. Concretization and sound transfer functions

The correctness of an abstract result needs a meaning for its elements. Let U be a set of concrete objects, such as integers or stores (**store**), and A an *abstract domain*, an ordered set of *domain elements* whose underlying set is its *carrier*. A domain element is an *abstract value* when it describes the values of one variable, such as an interval for one integer, and an *abstract state* when it describes stores. A concretization $\gamma : A \rightarrow \mathcal{P}(U)$ gives each domain element its meaning. It is monotone, so a less precise element admits at least the same objects: $a \sqsubseteq b$ implies $\gamma(a) \subseteq \gamma(b)$ (**gamma_mono**). With a concretization, the informal notion of soundness from Section 2.1 becomes a definition. An analysis result a at a program point v is *sound* if it describes every store that executions reach at v , that is, if

$$\mathcal{C}(v) \subseteq \gamma(a).$$

This inclusion is the consistency requirement between an abstract and a concrete interpretation [28, §6], [29, Def. 2.11].

The running example uses the *interval domain* [28, §9.2], [29, §4.5]. Its elements are $\perp$ and the intervals $[l, u]$ with $l \in \mathbb{Z} \cup \{-\infty\}$, $u \in \mathbb{Z} \cup \{+\infty\}$ and $l \leq u$, and an interval denotes the integers between its bounds:

$$\gamma([l, u]) = \{n \in \mathbb{Z} \mid l \leq n \leq u\}, \quad \gamma(\perp) = \emptyset.$$

The order is inclusion of these sets, so $[l, u] \sqsubseteq [l', u']$ holds exactly when $l' \leq l$ and $u \leq u'$. The join $[l, u] \sqcup [l', u'] = [\min(l, l'), \max(u, u')]$ is the smallest interval containing both. An interval cannot have holes, so the join may contain values that neither operand contains: $[0, 0] \sqcup [5, 5] = [0, 5]$. The meet intersects the bounds. This is the textbook domain. Voblint's representation **ivl** admits every pair of bounds, so $\perp$ is one of several empty pairs (Section 5.1).

At the loop head, both $[0, 5]$ and $[0, 10]$ are sound (Figure 2.2), as is every interval that contains $[0, 5]$. An analysis aims for a precise one, because a more precise element admits fewer values and so decides more checks: $[0, 5]$ proves $i \leq 5$ at the loop head, and $[0, 10]$ does not.

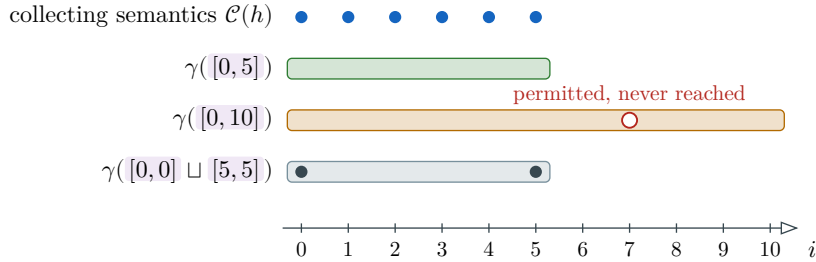


Figure 2.2: Concretization of intervals at the loop head. Both $[0, 5]$ and $[0, 10]$ contain the collecting semantics $\mathcal{C}(h)$, and $[0, 10]$ also permits $i = 7$. The join of $[0, 0]$ and $[5, 5]$ adds the values in between.

Much of abstract interpretation also assumes an abstraction function $\alpha : \mathcal{P}(U) \rightarrow A$ that forms a *Galois connection* with γ and gives every set of states a best abstraction [32, §5.3], the most precise domain element whose concretization contains the set. Not every useful domain has such best abstractions, and Cousot and Cousot treat a Galois connection as the ideal case that efficient representations do not always meet [33, §5]. Voblint uses γ

alone, like the verified analyzers of Jourdan et al. and of Nipkow [22, §2], [34, §5.2], since its soundness theorem claims coverage and nothing about optimal precision.

The analysis replaces each concrete operation by an abstract one, a *transfer function* $f^\# : A \rightarrow A$, which must not lose any state that the concrete operation can produce. In general, a program's steps form a relation $\rightarrow$ on U , and one state may have several successors, even infinitely many, for example, when the next value is read from user input. The *post-image* $\text{post}(X) = \{s' \mid \exists s \in X. s \rightarrow s'\}$ collects the successors of all states in X . The transfer function is sound if, for every a ,

$$\text{post}(\gamma(a)) \subseteq \gamma(f^\#(a)),$$

so stepping from any state that a describes leads to a state that $f^\#(a)$ describes [17, §4.2.2], [28, §6], [29, Def. 2.15]. The square of Figure 2.3 thus commutes only up to inclusion. In the counting loop, the edge from b to t executes $i = i + 1$. Over $\mathbb{Z}$, the interval transfer $[l, u] \mapsto [l + 1, u + 1]$ is sound for it, and applied to $[0, 4]$ it yields $[1, 5]$, which contains every successor $i = 1, \dots, 5$. Soundness composes along paths [29, Thm. 2.6], and at control-flow joins, the upper-bound laws of the join keep the states from both predecessors (`gamma_sup_ub1`, `gamma_sup_ub2`).

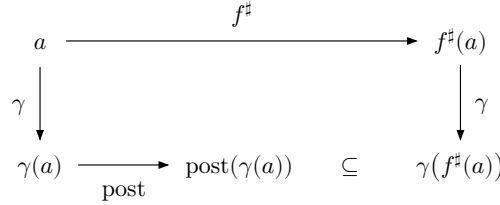


Figure 2.3: Soundness of a transfer function $f^\#$: the square commutes up to inclusion.

Sound transfer functions make every post-fixpoint that contains the initial configurations a sound result [29, Thm. 2.8]. The standard argument works with *configurations*, each a program point together with a store, so that one relation $\rightarrow$ describes every step of the program. With the set I of initial configurations, the reachable configurations form the least fixpoint of

$$F(X) = I \cup \text{post}(X)$$

[17, §4.1.1, Thm. 4.1]. One application of F keeps the initial configurations and adds every successor of a configuration already in X . A set with $F(X) \subseteq X$ therefore contains the initial configurations and is closed under steps, so no execution can leave it. The least such set, $\text{lfp } F$, contains exactly the configurations that some execution reaches. In the counting loop, executions begin at `start` with any value of i , and $\text{lfp } F$ contains (h, n) exactly for $n \in \{0, \dots, 5\}$, which is $\mathcal{C}(h)$.

Now take a domain element a with $I \subseteq \gamma(a)$ and $f^\#(a) \sqsubseteq a$, for a sound $f^\#$. Soundness of $f^\#$ and monotonicity of γ give $F(\gamma(a)) \subseteq I \cup \gamma(f^\#(a)) \subseteq \gamma(a)$. So $\gamma(a)$ is a post-fixpoint of F and, by Knaster–Tarski, contains $\text{lfp } F$, every reachable configuration.

This argument uses a single domain element a for the whole program. Analyzers, however, typically keep one abstract state per program point [17, §4.2.1], or per point and context (Section 2.6). The element a then becomes a family of abstract states, and the condition $f^\#(a) \sqsubseteq a$ becomes one inequality for each of these states, as the next section describes. Voblint’s proof uses the same principle on a set of activation traces defined inductively as a least fixpoint, and proceeds by induction over its rules.

2.4. Analyses as constraint systems

An analysis needs an abstract state at every program point, and the state at one point depends on the states at its predecessors. The analysis therefore states its result as a system of constraints. Each *unknown* of the system stands for the abstract state at one program point, and each constraint says what the unknown must cover.

The counting loop has one variable, so an abstract state is an interval for i . (With several variables, a *non-relational* state maps each variable to its own abstract value.) With one unknown per program point of Figure 2.1, each unknown must cover what the edges entering its point produce:

$$\begin{aligned} h &\sqsupseteq [0, 0] \sqcup t, & b &\sqsupseteq h \sqcap [-\infty, 4], \\ t &\sqsupseteq b +^\# [1, 1], & e &\sqsupseteq h \sqcap [5, +\infty]. \end{aligned}$$

The assignment $i = 0$ yields $[0, 0]$ at h , the condition $i < 5$ restricts the branch into the body, and its negation restricts the exit. Substituting b and t into the first inequality gives one for the loop head, $h \sqsupseteq f(h)$, with

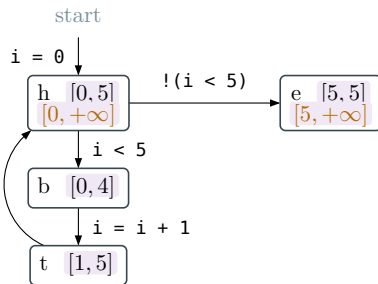
$$f(d) = [0, 0] \sqcup ((d \sqcap [-\infty, 4]) +^\# [1, 1]),$$

whose post-fixpoints appear in Figure 2.4b. A solution assigns an interval to every unknown such that all four inequalities hold, and Figure 2.4a shows two. Both contain the collecting semantics ($\{0, \dots, 5\}$ at h), and the second is less precise at h and e , so not every solution is the least one.

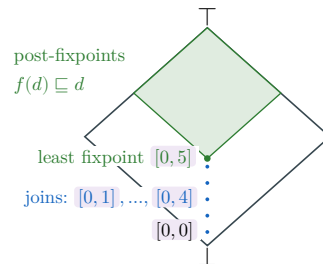
In general, let $\mathcal{X}$ be a set of unknowns and $\sigma : \mathcal{X} \rightarrow A$ a *valuation*, which assigns a domain element to every unknown. Valuations are ordered pointwise: $\sigma \sqsubseteq \sigma'$ if $\sigma(x) \sqsubseteq \sigma'(x)$ for every unknown x . Each unknown x has a *right-hand side* rhs_x , a function of the valuation, and σ is a *post-solution* if

$$rhs_x(\sigma) \sqsubseteq \sigma(x) \quad \text{for every unknown } x$$

([post_solution](#)), what the constraint-system literature calls a solution [27, §2.4.4]. For the loop head, $rhs_h(\sigma) = [0, 0] \sqcup \sigma(t)$. A post-solution is a post-fixpoint of all right-hand sides at once, so the argument of Section 2.3 applies at every program point, and a post-solution bounds the reachable states everywhere. The term *constraint system* is used for the general concept and the term *equation system* for the system that Voblint generates and the solver receives, although soundness only requires inequalities.



(a) two solutions on the graph



(b) post-fixpoints of $h \sqsupseteq f(h)$

Figure 2.4: The counting loop’s inequalities, solved by hand. (a) The least solution at each node of Figure 2.1 and, in orange where it differs, a less precise solution. (b) Post-fixpoints of the loop-head constraint (schematic, shaded), its least fixpoint $[0, 5]$, and the Kleene iterates.

2.5. Widening and narrowing

An analysis should terminate in a reasonable time, even on a program whose executions do not terminate. With joins alone, successive abstract iterates can mirror successive loop iterations. At the example's loop head, joins take five increases after $[0, 0]$. For $i = 0$; **while** (**true**) { $i = i + 1$; }, the iteration over intervals with bounds in $\mathbb{Z}$ never stabilizes, because these intervals have infinite ascending chains such as

$$[0, 0] \sqsubset [0, 1] \sqsubset [0, 2] \sqsubset \dots$$

A *widening* $a \nabla b$ replaces the update from the old value a to the new value b by a guess that lies above both, which keeps the result sound (**widening**). The standard interval widening replaces a growing bound by $+\infty$ [28, §9.2], [29, §4.5.2], so widening at h turns the first change there, from $[0, 0]$ to $[0, 1]$, into $[0, +\infty]$ at once. Propagated through the loop, this value yields the less precise solution of Figure 2.4a, which loses the bound 5 of the loop condition $i < 5$.

The branch into the body restricts $[0, +\infty]$ to $[0, 4]$, the body yields $[1, 5]$, and evaluating the head again gives $[0, 0] \sqcup [1, 5] = [0, 5]$. A widened iteration only grows and would not take this smaller value. A *narrowing* lets it descend. It maps the current value a and a candidate $b \sqsubseteq a$ to a value between the two [28, §9.3.4], [33, §4]:

$$b \sqsubseteq a \Delta b \sqsubseteq a.$$

In the counting loop, the interval narrowing reaches $[0, 5]$, but in general, narrowing does not always recover the least fixpoint. Both steps appear in Figure 2.5a. The verified TD solver applies both operators in a single update, *warrowing* (**warrow**), shown in Figure 2.5b. It narrows when the candidate lies below the current value and widens otherwise [35], [36, §3], [37].

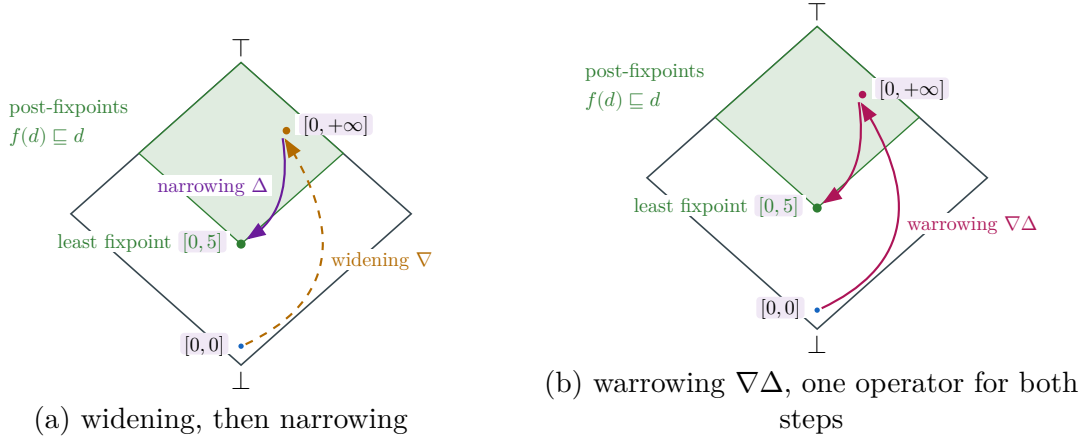


Figure 2.5: Extrapolation on the lattice of Figure 2.4b. (a) Widening jumps from $[0, 0]$ to the post-fixpoint $[0, +\infty]$, and narrowing then recovers the least fixpoint $[0, 5]$. (b) Warrowing takes the same two steps with one operator, which narrows when the candidate lies below the current value and widens otherwise. Here both reach $[0, 5]$.

Soundness, termination, and precision depend on different properties of these operators. The solver's classes **widening** and **narrowing** state the upper-bound law of widening and the *bracket laws* of narrowing above. Soundness needs only the upper-bound law and the lower bracket law $b \sqsubseteq a \Delta b$. With them, each warrowing update lies above the candidate (**warrowing_properties**), so an unknown whose value no longer changes satisfies its

inequality. The solver’s correctness theorem uses only this and holds whether the right-hand sides are monotone (Section 2.8). Monotonicity is needed for the classical theory of least fixpoints, but not to certify that a returned valuation is a post-solution. Termination needs more. A classical widening is chosen so that repeated widening stabilizes after finitely many steps [28, §9.1.3], [33, §4], and a classical narrowing must also make repeated narrowing stabilize. The solver’s classes omit these stabilization laws, since termination of the side-effecting TD that Voblint uses is not proved, so the laws they state do not make the solver terminate. Within these laws, the choice of widening and narrowing affects only precision and termination.

2.6. Context-sensitive analysis of procedures

The counting loop has no procedures. With procedures, the analysis must decide which abstract state a callee’s body starts from. Take the calls `bump(5)` and `bump(4)` of Chapter 1, where `bump` returns its argument plus one. With one unknown per program point, the entry of `bump` must cover both arguments, so, over intervals, its result is at best $[5, 6]$, and the analysis cannot show that the first call returns 6. A context-sensitive analysis instead separates the *activations* of a procedure, each execution from its call to its return, by a *calling context*, an abstraction of how the activation was reached [38, §7.2.1]. Its unknowns become pairs $[v, c]$ of a program point v and a context c [24, §3], so every context of a procedure gets its own copy of the procedure’s unknowns.

Sharir and Pnueli describe two classical choices of what a context records [39]. The call-string approach uses the most recent call sites as the context. Since `bump(5)` and `bump(4)` are called from different sites, each gets its own copy. The functional approach instead summarizes a procedure by its effect on each entry state, computed only for the entry states that occur. With the entry state as context, `bump` gets one copy for $[5, 5]$ and one for $[4, 4]$, with the results 6 and 5. Neither choice subsumes the other. Only entry states separate two calls from one site with different arguments, and only call strings separate calls from two sites with the same argument. Apinis et al. also distinguish calls by only part of the entry state [24, §4], for instance by the argument of `bump` alone, ignoring global variables it never reads. The context is therefore a choice of abstraction, and the trivial choice, a single context, gives back the analysis with one unknown per program point.

A context-sensitive analysis handles a call with two abstract operations [24, §3]. The first, `enter#`, maps the caller’s state before the call to the callee’s entry state. The callee’s context is chosen at the same point, for example as the extended call string or as this entry state itself. The second, `comb#`, computes the caller’s state after the call. It needs the callee’s exit state, which holds the return value, and the caller’s state before the call, which holds the caller’s local variables that the callee does not see. Goblint’s [constraint generator](#) implements them as `enter` and `combine` (Section 6.1).

2.7. Side-effecting constraint systems

The analyses so far are *flow-sensitive*: they respect the order in which statements execute and therefore compute a separate abstract state for every program point. For some facts, one value for the whole program is enough. A *flow-insensitive* analysis ignores execution order and keeps one abstract value for an aspect of the state across the program, for

example, one range for a global variable, and mixed analyses choose between the two per aspect [2].

A flow-insensitive fact needs an unknown of its own that belongs to no program point, and every point that changes the fact has to feed its new value into that unknown. An ordinary right-hand side for the unknown would have to list all these points in advance, which is not always possible. In a context-sensitive analysis, for example, an unknown that writes a global is a pair of a program point and a calling context, and the contexts are discovered only while the system is solved.

Every context-sensitive analysis meets the same problem at procedure entries, even without flow-insensitive facts. The entry of a callee in context c must cover the entry states of all calls that enter it in c (Section 2.6). When contexts are computed from abstract entry states, as for `bump(5)` and `bump(4)`, the calls that enter a context are known only once the states of their callers are known. The entry's right-hand side therefore cannot list them either. Voblint meets this at every call (Section 8.1).

Side-effecting constraint systems avoid that list. Seidl, Vene, and Müller-Olm introduced them for multithreaded programs [23, §3], and Apinis et al. use them for context-sensitive analysis [24]. While the right-hand side of one unknown is evaluated, it may *publish* a *contribution* to another unknown as a *side effect*. As in the verified solver, side effects target only *global unknowns* $\mathcal{X}_G$, which have no right-hand side and receive only contributions. From here on, $\mathcal{X}$ denotes the *local* unknowns, which keep their right-hand sides, and a valuation σ assigns values to both kinds. A global unknown does not have to correspond to a global variable of the program. It can hold any fact to which several right-hand sides contribute. In Voblint, for instance, the entry state of a callee in one context is a global unknown, and every call that enters this context publishes its entry state to it (Section 8.1). An analysis may also keep facts of its own in global unknowns that stand for no program variable at all (Chapter 6). A right-hand side then returns a pair [27, §2.4.5], $rhs_x : (\mathcal{X} \cup \mathcal{X}_G \rightarrow A) \rightarrow (\mathcal{X}_G \rightarrow A) \times A$, of a partial map ρ of the contributions it publishes, joined per global unknown, and its own value d . If $rhs_x(\sigma) = (\rho, d)$, a post-solution must bound both, $d \sqsubseteq \sigma(x)$ and $\rho(g) \sqsubseteq \sigma(g)$ for every $g \in \text{dom}(\rho)$. If the assignments `g = 5` and `g = 4` publish `[5, 5]` and `[4, 4]` to the unknown of a flow-insensitive global `g`, that unknown must bound their join `[4, 5]`. This describes what a right-hand side computes. The verified solver represents its evaluation by a strategy tree (Section 2.8), so that it discovers reads and side effects while it solves.

With contexts as indices, a system can have infinitely many unknowns, of which only those that influence the *query*, the unknown whose value is asked for, matter. A *local* solver therefore explores the system from the query and solves only the unknowns that it meets [40]. Its result comes with a *certificate*, the property `part_post_solution`, which requires the bounds above only on a set of unknowns that contains the query and is closed under the local reads of its right-hand sides.

2.8. The verified top-down solver and its guarantee

Plain Kleene iteration recomputes every unknown in every round, although most values no longer change, so practical solvers update one unknown at a time. They work in rounds or with a worklist of unknowns whose inequalities may be violated [31, §§1.5, 1.12], in orders

chosen for widening [41], or locally, on demand from a query, as the solver RLD does [42]. Voblint reuses the verified local top-down solver (TD) of [3], [4].

TD starts from the query and evaluates right-hand sides on demand. When the right-hand side of x reads an unknown y , TD first solves y and records that x depends on y . An unknown is *stable* if it has been evaluated and no unknown that it read has changed since then. When the value of y changes later, every unknown that read it is *destabilized* and evaluated again. The unknowns at which a read closes a cycle are the solver's *widening points* [40]. TD detects them while it solves [36, §4], so they can differ from the loop heads of the control-flow graph. At a widening point, it updates the old value with the new one by warrowing. The playground replays this run on the counting loop step by step. Contributions published to a global unknown are merged into its value by an *update rule* (`update_rule`), which keeps one record per *origin*, the unknown whose right-hand side published the contribution (Section 9.2). The run ends once every unknown the query transitively reads is stable, and it returns this *stable set* V together with the valuation σ .

To solve on demand, TD has to discover which unknowns a right-hand side reads. In the counting loop, these reads are fixed. In general, a right-hand side may choose the next unknown from a value it has already read. At a procedure call, for instance, the callee context to read is known only once the caller's value has been read. The verified TD solver therefore receives every right-hand side as a *strategy tree* (`strategy_tree`), a small program that reads one unknown at a time and makes every read and every side effect explicit:

$$\tau ::= \text{Answer}(d) \mid \text{QueryL}(y, k) \mid \text{QueryG}(g, k) \mid \text{Side}(g, d, \tau)$$

with a local unknown $y \in \mathcal{X}$, a global unknown $g \in \mathcal{X}_G$, a domain element $d \in A$ and a continuation $k : A \rightarrow \tau$ that receives the value read. `Answer`(d) returns d as the value of the right-hand side, `QueryL` and `QueryG` read a local or a global unknown, and `Side`(g, d, τ) publishes d to g before continuing with τ . Figure 2.6 shows the trees of the counting loop and of a flow-insensitive assignment.

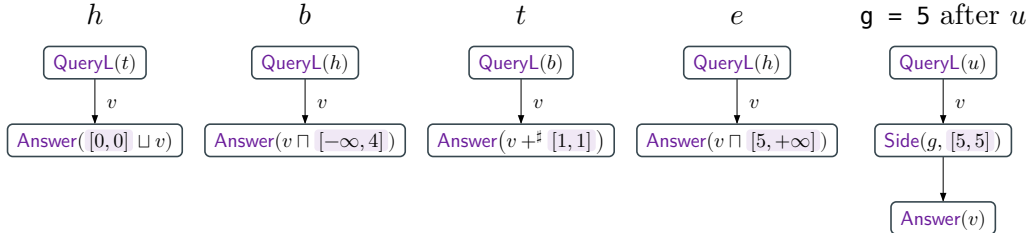


Figure 2.6: Right-hand sides as strategy trees. The first four are the right-hand sides of the inequalities of the counting loop (Section 2.4): each reads one unknown, passes its value v on, and answers. The last publishes the value of a flow-insensitive global g (Section 2.7) before answering.

The solver's main theorem, `partial_post_solution`, establishes the certificate `part_post_solution` of Section 2.7 and states partial correctness. If the run terminates on the query x and returns its stable set V and the valuation σ , then V contains x , is closed under the local reads of its right-hand sides, and, on V , the valuation is a post-solution, published contributions included. It says nothing about unknowns outside V , and V is called the *solved set*. Unlike some TD variants without side effects [43], this solver has no termination proof (Appendix A explains how Isabelle defines such a function).

PART II

What Must Be Over-Approximated

3. VIMP and Its Control-Flow Graphs

Chapter 2 analyzed a single loop through its control-flow graph. The soundness theorem, however, should talk about the programs a user writes, and these have procedures and recursion. Such a program has a meaning of its own, given by a source semantics that executes the program text directly (Section 3.2). To generate its equations, however, an analysis needs a fixed, finite set of program points, because its constraint system attaches one abstract state to each of them. The source semantics provides no such set. It locates an execution only by the command that still has to run, and under recursion these commands nest arbitrarily deep (Section 3.3.1). Voblint therefore compiles a program to a procedure-aware control-flow graph, whose nodes serve as the program points, and generates its equations from the graph (Figure 3.1). The compilation step must then be part of the proof. If soundness held only for the graph, the compiler would stay outside the theorem, and a verdict would say nothing about the source program. This chapter closes the gap with the first link of the soundness chain. Every source execution of a well-formed program is matched by a graph execution that holds the same store (Section 3.5). The running example is the program of Figure 3.1, in which `main` calls `bump(5)` and `bump(4)`.

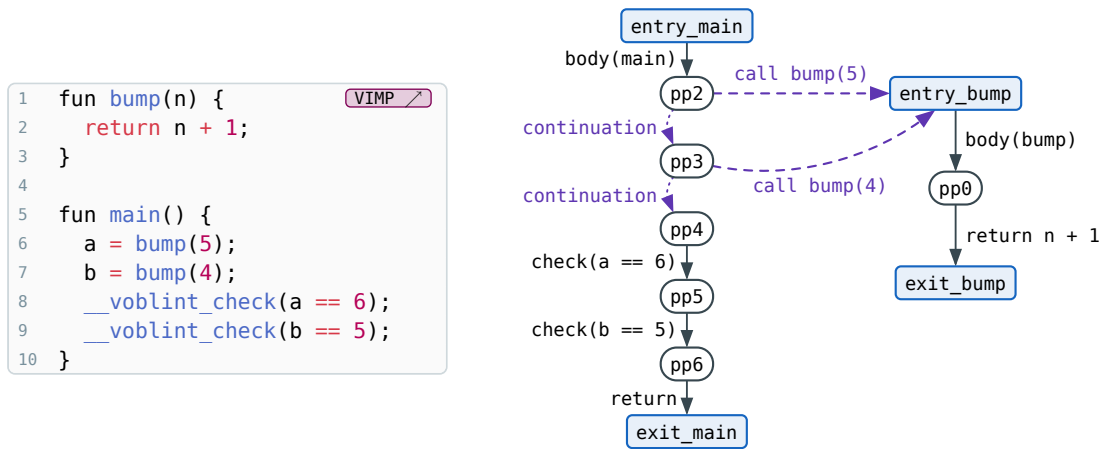


Figure 3.1: The running example and its procedure-aware graph, with node and edge labels as `voblint --dot` prints them. Solid arrows are local edges and dashed arrows are call edges. Dotted connectors are not edges. They show the continuation of a call edge, that is, the node where the caller resumes. Figure 8.1 shows the unknowns that Chapter 8 generates around the first call.

3.1. VIMP syntax and well-formed programs

VIMP (Figure 3.2) is a small imperative language with C-like syntax that the formalization analyzes. It is kept small because every construct needs a semantics, compiler clauses, and a transfer function proved sound in every abstract domain. It has unbounded integer

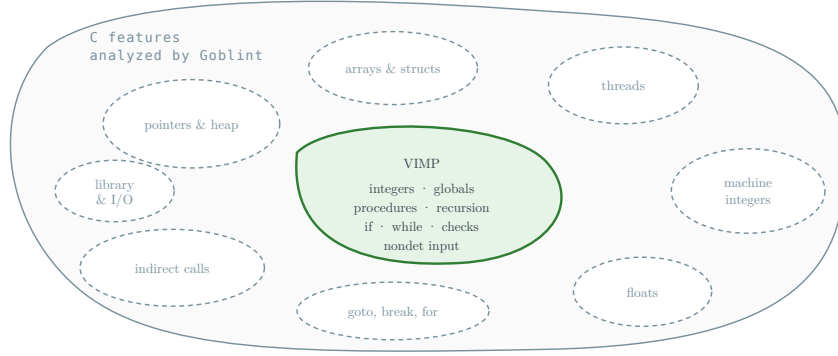


Figure 3.2: Constructs represented in VIMP relative to Goblint’s C input, taken from the explainer page. Dashed regions are constructs that VIMP omits. Sizes and positions are schematic. The figure compares only syntax. Section 3.6 lists where shared constructs have different meanings.

variables, locals and globals, procedures with value parameters, direct calls with an optional result, recursion, assignment, sequencing, conditionals, `while` loops, nondeterministic input, and checks. It omits machine integers, pointers and the heap, arrays and structs, indirect calls, threads, floating point, `goto`, `break`, `continue`, and exceptions. Each of these would need its own preservation argument at every layer (Chapter 15 discusses the heap). Section 3.6 compares the fragment with C and with Goblint’s input.

One of these omissions matters in later chapters. Without pointers, concrete memory is a store from variable names to integers. The non-relational analyses of Chapter 5 lift it pointwise to one abstract value per variable. Section 11.3 gives a carrier that is not pointwise.

Expressions are integer-valued. As in C11, comparisons and logical operators yield 0 or 1 ([§6.5.8p6](#), [§6.5.9p3](#), [§6.5.3.3p5](#), [§6.5.13p3](#), [§6.5.14p3](#)), and a condition holds when it is nonzero ([truthy](#), [§6.8.4.1p2](#), [§6.8.5p4](#)).

Definition 3.1 (Expressions, datatype `exp`).

$$\begin{aligned}
 e ::= & n \mid x \mid e + e \mid e - e \mid e * e \mid e / e \mid e \% e \\
 & \mid e < e \mid e \leq e \mid e > e \mid e \geq e \mid e != e \mid e == e \\
 & \mid !e \mid e \&\& e \mid e || e
 \end{aligned}$$

Each operator has its own constructor, because an abstract domain chooses its transformer by the operator, and the executable analyzer has to make this choice by inspecting the syntax. The two AFP languages with procedures that were considered, IMP2 and Simpl, do not allow this. IMP2 [44] represents a binary operator by a HOL function of type `int ⇒ int ⇒ int`. Concrete evaluation can apply such a function, but executable code cannot compare it with another one, so it cannot tell `+` from `*`. IMP2 also matches Voblint’s programs poorly in other respects. Its semantics is deterministic, so it has no input, and its call command takes no arguments and returns no value, so parameters and results pass through variables set by generated code (Chapter 14). Schirmer’s Simpl [45] is more expressive, with nondeterministic specifications and a derived call command that passes parameters, but its basic commands and conditions are likewise HOL functions and sets of states. Both languages target program verification, where semantic objects are convenient,

whereas an analyzer needs syntax that it can inspect. VIMP is therefore defined from scratch, with such a syntax, which code generation can also export.

An expression is evaluated in a *store* s of type `store`, a total function $\text{vname} \Rightarrow \text{int}$ from variable names to integers. Evaluation `aval`, written $\llbracket e \rrbracket_e s$, is a function defined by one equation per constructor, for example

$$\begin{aligned} \llbracket n \rrbracket_e s &= n, & \llbracket x \rrbracket_e s &= s(x), & \llbracket e_1 / e_2 \rrbracket_e s &= \text{c_div} (\llbracket e_1 \rrbracket_e s, \llbracket e_2 \rrbracket_e s), \\ \llbracket e_1 < e_2 \rrbracket_e s &= \begin{cases} 1 & \text{if } \llbracket e_1 \rrbracket_e s < \llbracket e_2 \rrbracket_e s \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

Every expression has a value, because VIMP fixes the results of division and remainder by zero (`c_div`, `c_mod`, Section 3.6). It also has no effects, so a transfer function may evaluate a guard again, as the backward filtering of Chapter 5 does. Input enters only through the statement `x = __voblnt_nondet_int()`; and never inside an expression. This keeps `aval` a function.

Commands are built from expressions. Besides the structured statements, there are calls, which may store a result, and two markers that appear only during execution.

Definition 3.2 (Commands, datatype `gcom`).

$$\begin{aligned} c ::= & \text{skip} \mid x := e \mid \text{check } (e) \mid c; c \mid \text{if } (e)\{c\} \text{ else } \{c\} \mid \text{while } (e)\{c\} \\ & \mid [x :=] p(e, \dots, e) \mid \text{return } e? \mid \text{Restore} \mid \text{Unwind} \end{aligned}$$

Every constructor except sequencing also carries an *annotation*, whose type is a parameter of `gcom`. The commands of a parsed program carry the unit annotation and have type `com`. The grammar above omits annotations, which the rules of Section 3.2 write as i . Section 3.5.2 explains why they are needed.

In concrete syntax, an assignment is written `x = e;` and a check `__voblnt_check(e);`. The parser lowers `true`, `false`, and unary minus to the expression grammar above. A check states a condition for the analyzer to decide (Section 10.2) and does not influence the execution. It behaves like `skip` whatever value e has, so it neither stops the execution nor refines the store. Its constructor also carries the source position, which only the report uses, to name the check.

Calls come in two kinds. A call of a declared procedure starts an *activation*, one execution of the procedure body from its call to its return. VIMP also has three built-in *library calls*, `__voblnt_nondet_int`, `min`, and `max`, which a program calls but does not declare. The fixed table `special_table` recognizes their names, and `special_result` gives the values they may return: any integer for `__voblnt_nondet_int`, and the minimum or maximum of the arguments for `min` and `max`. A library call finishes in one step and stores its value, without an activation or a frame (Section 3.2). The remaining constructors, `Restore` and `Unwind`, are not source syntax (`source_com` excludes them). They appear only during execution, where Section 3.2 uses them for calls and returns.

A program (`imp_prog`) is a list of procedure declarations (`proc_rep`), each with formal parameters and a body, plus a list of global variables (`declared_global_vars`). Its procedure table (`proc_table`) looks up a name in that list, and the classifier `declared_global` tests membership in the globals. The entry procedure is the declaration named `main`, and `main_body` is its body. Figure 3.3 shows the running example in this form. The formal

the store and leaves **skip**, and a check leaves the store unchanged. A sequence steps its first command until it is **skip**, a conditional selects a branch by **truthy**, and a loop unfolds into a conditional that runs the body and then the loop again. After each step, the remaining command describes the remainder of the execution [46, §3.2], here called a *residual*.

Calls raise two questions. While a callee runs, the caller's remaining work has to wait somewhere, and a **return** has to skip the rest of the callee's body. VIMP keeps the waiting work inside the residual command and saves only data on a stack. A call continues with the callee's body followed by **Restore**, a marker that exists only at run time and hands control back to the caller once the body falls through. A **return**, possibly before the end of the body, becomes the marker **Unwind**, which discards the commands after it until it meets the **Restore** of its call. Each suspended activation keeps its data in a *frame*.

Definition 3.3 (Source execution, inductive **pstep**). A source configuration is a triple (c, s, frs) of the command that remains to run, a store s , and a list frs of **frame** values for the suspended activations. Each frame records the caller's store and the variable, if any, that receives the result. The step relation **pstep** relates a configuration to its successor.

```

1 (* src/Program_Model/VIMP/VIMP_Proc.thy *)
2 inductive
3   pstep :: "(vname ⇒ bool) ⇒ 'a ptable ⇒ 'a acom × store × frame list
4             ⇒ 'a acom × store × frame list ⇒ bool"
5   ("(_ , _ ⊢ / _ →p / _)" [51, 51, 51, 51] 50)
6   for G :: "vname ⇒ bool" and Π :: "'a ptable"
7   where
8     Assign: "G, Π ⊢ (Assign i x a, s, frs) →p (SKIP i, s(x := [a]e s), frs)"
9   | Check: "G, Π ⊢ (Check i l c, s, frs) →p (SKIP i, s, frs)"
10  | Seq1: "G, Π ⊢ (Seq (SKIP i) c2, s, frs) →p (c2, s, frs)"
11  | Seq2: "G, Π ⊢ (c1, s, frs) →p (c1', s', frs')
12          ⇒ G, Π ⊢ (Seq c1 c2, s, frs) →p (Seq c1' c2, s', frs')"
13  | IfTrue: "truthy ([b]e s) ⇒ G, Π ⊢ (If i b c1 c2, s, frs) →p (c1, s, frs)"
14  | IfFalse: "¬ truthy ([b]e s) ⇒ G, Π ⊢ (If i b c1 c2, s, frs) →p (c2, s, frs)"
15  | While: "G, Π ⊢ (While i b c, s, frs)
16            →p (If i b (Seq c (While i b c)) (SKIP i), s, frs)"
17  | Call: "Π p = Some decl
18            ⇒ length actuals = length (formals decl)
19            ⇒ distinct (formals decl)
20            ⇒ vals = map (λe. [e]e s) actuals
21            ⇒ callee = bind_formals (formals decl) vals (enter_state G s)
22            ⇒ G, Π ⊢ (Call i dst p actuals, s, frs)
23                  →p (Seq (body decl) (Restore i),
24                      callee,
25                      Frame s dst # frs)"
26  | Special: "special_table p = Some desc
27              ⇒ classify_special desc actuals = Some sc
28              ⇒ special_result sc s v
29              ⇒ G, Π ⊢ (Call i (Some x) p actuals, s, frs) →p (SKIP i, s(x := v), frs)"
30  | RestoreStep:
31    "G, Π ⊢ (Restore i, s, Frame fr dst # frs)
32      →p (SKIP i, combine_assign dst (s ret_var) (combine_env G fr s), frs)"
33  | ReturnSome:
34    "G, Π ⊢ (Return i (Some e), s, frs)
35      →p (Unwind i, s(ret_var := [e]e s), frs)"
36  | ReturnNone:
37    "G, Π ⊢ (Return i None, s, frs) →p (Unwind i, s, frs)"
38  | UnwindDead:
39    "¬ is_Restore c2
40      ⇒ G, Π ⊢ (Seq (Unwind i) c2, s, frs) →p (Unwind i, s, frs)"
41  | UnwindAct:
42    "G, Π ⊢ (Seq (Unwind j) (Restore i), s, Frame fr dst # frs)
43      →p (SKIP i, combine_assign dst (s ret_var) (combine_env G fr s), frs)"

```

A classifier $G :: vname \Rightarrow bool$, separate from the store, says which names are global, because a call resets only the locals. For a program, it is **declared_global**. The relation is written $G, \Pi \vdash \kappa \rightarrow_p \kappa'$ for configurations κ, κ' , and a procedure table Π . Its closure

is $\rightarrow_p^*$, and its rules are listed below. No rule inspects an annotation. The rules only copy annotations, from the rewritten command or from the procedure table. The [explainer page](#) animates these steps.

A call evaluates the actuals in the caller's store, resets the locals, binds the formals (`enter_state`, `bind_formals`), and pushes a frame. A `return e` writes the value of e to `ret_var`, and popping the frame merges the caller's locals, the callee's globals, and the result. These operations select by name and are polymorphic in the value type (`enter_binding`, `combine_env`, `combine_assign`). Therefore, the non-relational analyses reuse the concrete definitions at their abstract state types, with $\top$ in place of the reset value 0.

The root activation is modeled without a `Restore`, since `main` has no caller to return to, and a bare `Unwind` has no step (`pstep_Unwind_stuck`). The source contract therefore rejects `return` in `main` (`no_return`). A source execution starts from the configuration (`main_body` Π , s_0 , $[]$), with an initial store s_0 in `cinit_stores` (Section 3.6).

Just as Nipkow and Klein show that well-typed programs do not get stuck [6, §9.1.2], Voblint proves that well-formed VIMP programs do not. Otherwise, claims about the points after a stuck execution would hold vacuously. For a program satisfying `wf_source_program`, every reachable configuration has either finished, as `skip` with an empty frame stack, or can step:

```
1 (* src/Program_Model/Compile/Simulation/Source_Progress.thy *)
2 theorem source_progress:
3   assumes wf: "wf_source_program  $\mathcal{G}$   $\Pi$ "
4   and run: " $\mathcal{G}$ ,  $\Pi \vdash (\text{main\_body } \Pi, s_0, []) \rightarrow_p^* (c, s, \text{frs})$ "
5   shows "(is_SKIP  $c \wedge \text{frs} = [] \vee (\exists \text{cfg}'. \mathcal{G}, \Pi \vdash (c, s, \text{frs}) \rightarrow_p \text{cfg}')$ )"
```

The source contract rules out every reason a call could fail to step, and every library call has a result (`special_result_ex`).

3.3. The procedure-aware control-flow graph

The source semantics executes a program by rewriting the command that remains to run. The analysis instead works on a procedure-aware control-flow graph, whose nodes are the program points that carry abstract states and whose local edges and call edges carry the transfers.

3.3.1. Why the analysis needs program points

An analysis attaches abstract states to program points and transfer functions to transitions. `pstep` offers neither in a usable form. Its only notion of location is the command that remains to run. Without recursion, only finitely many distinct residual commands arise, because a loop may run arbitrarily often, but its unfolding revisits the same command forms. Recursion is different, because a recursive call can nest another `Restore` marker at every call depth. Residual commands therefore do not form the finite, fixed set of locations that Voblint needs. An assignment is also not a transition between fixed program points. Inside a sequence, it steps through the rule `pstep.Seq2`, which rebuilds the surrounding residual command. So the same assignment is executed by a different derivation inside every surrounding command. Finally, nothing names the entry of a procedure, its result, or

the point where a caller resumes. The schematic control-flow graph of Figure 2.1 solves the first two problems. It has finitely many nodes and one transfer per edge. The procedure-aware graph then solves the third. It adds an entry node and a result node per procedure, and call edges that record their continuations. Each gives the analysis a place for one fact about a call. The entry node holds the state a callee starts from, the result node the state it returns with, and the continuation names the node where the caller resumes. A call can therefore be analyzed by connecting the call site, the callee’s entry and result nodes, and the continuation, instead of by following the callee’s body (Chapter 8).

Introducing a graph is a design choice. A *syntax-directed* analyzer, such as that of Nipkow and Klein [6, Ch. 13], computes its result by recursion over the command structure and needs no graph. Voblint instead follows Goblint and is *constraint-based* [24]: it turns the program into a system of inequalities over unknowns indexed by the graph’s nodes. A generic solver that knows nothing about programs then solves this system. The solver needs no knowledge of calls, because a call contributes right-hand sides and side effects to the same system (Chapter 8). The price is two extra proof layers. This chapter must prove by a simulation that the compiler preserves the program’s executions, and Chapter 4 must say what the inequalities mean for those executions. In return, the argument about programs ends at a solver-independent certificate, a post-solution on the solved set (Section 9.3).

3.3.2. Nodes, local edges, and call edges thy

Definition 3.4 (Control-flow graph, record `cfg`). A graph g is a record of type `cfg`:

```
1 (* src/Program_Model/CFG/CFG_Def.thy *)
2 record cfg =
3   intra      :: "(cfg_node × edge_action × cfg_node) set"
4   calls      :: "(cfg_node × call_action × cfg_node × cfg_node) set"
5   cfg_entry  :: cfg_node
```

The field `intra` holds the *local edges* (u, a, v) , each labeled by an `edge_action` a . The field `calls` holds the *call edges* $(u, ca, \text{FunctionEntry } q, k)$. Such an edge consists of the call site u , the call information ca , the entry node of the callee (by `wf_cfg`), and the node k where the caller resumes. Nodes (`cfg_node`) are `Statement` n , `FunctionEntry` p , or `FunctionResult` p . The field `cfg_entry` holds the entry node.

The call information `CallEdge` dst *formals args* (`call_action`) copies the callee’s formals onto the edge, so the graph semantics binds actuals to formals without looking up the procedure table. VIMP has no function pointers (Section 3.1), so every call names its callee, and the compiler can fix the target of each call edge. Supporting indirect calls would require discovering call targets during the analysis, which is left for future work. The node kinds match Goblint’s (node0.ml), which also has statement and function-entry nodes and calls Voblint’s `FunctionResult` node `Function`.

The graph keeps local edges and call edges apart, in its fields `intra` and `calls`, because they execute differently. A local edge changes the store within one activation, while a call edge starts a new one. No edge action denotes a call, so a call can never be taken as a local step, and each rule of the trace semantics in Chapter 4 reads at most one of the two relations.

Returns, in contrast, are ordinary local edges. A `return` in p leads to the result node `FunctionResult` p , which all callers of p share, so recursion needs no copies of nodes. A shared result node cannot know where to continue, and in a compiled graph, no edge leaves it.

The call edge records that place instead, as its continuation k . After a return, execution resumes at the continuation saved in the frame, and an analysis connects the callee's exit state to the correct caller through the call edge. The matching caller is explicit in the graph, so no separate return-to-call matching is needed. A completed program ends at `FunctionResult` `main (cfg_exit)`, so there is no separate global exit node.

3.4. Compiling VIMP to a graph

The compiler turns a VIMP program into a procedure-aware graph. Its design decides which program points exist, and the simulation of Section 3.5 relies on how it lays out the code of each command.

3.4.1. Compiling commands with continuations thy

To emit the edges of a command, a compiler has to know where control goes after it. Compilers commonly pass this target down, so that each statement receives the label of the code that follows it, the inherited attribute $S.next$ of the dragon book [38, §6.6.3]. Voblint's compiler works the same way and is therefore called *continuation-passing*. Besides the command, it takes the node k at which control continues afterward, and it emits the command's edges so that they end at k (`compile` for commands, `compile_proc` and `compile_prog` for procedures and programs). Schematically, writing `compile` (c, k) for the edges of c against the continuation k , and n and h for fresh nodes,

$$\begin{aligned}
\text{compile } (x = e, k) &= \left\{ n \xrightarrow{x=e} k \right\}, \\
\text{compile } (c_1; c_2, k) &= \text{compile } (c_1, \text{entry}(c_2)) \cup \text{compile } (c_2, k), \\
\text{compile } (\text{if } (e)\{c_1\} \text{ else } \{c_2\}, k) &= \left\{ n \xrightarrow{e} \text{entry}(c_1), n \xrightarrow{\neg e} \text{entry}(c_2) \right\} \\
&\quad \cup \text{compile } (c_1, k) \cup \text{compile } (c_2, k), \\
\text{compile } (\text{while } (e)\{c\}, k) &= \left\{ h \xrightarrow{e} \text{entry}(c), h \xrightarrow{\neg e} k \right\} \cup \text{compile } (c, h), \\
\text{compile } (x = q(\bar{e}), k) &= \{(n, x = q(\bar{e}), \text{FunctionEntry } q, k)\}, \\
\text{compile } (\text{return } e, k) &= \left\{ n \xrightarrow{\text{return } e} \text{FunctionResult } p \right\}.
\end{aligned}$$

A **skip** and a check compile like the assignment, to one edge from a fresh node to k . The alternative would give every fragment an exit node of its own, and sequencing two fragments would then need a no-op edge between two nodes that denote the same program point. With continuations, a node is a program point between two transfers, as in Goblint's CFG. A conditional needs no join node, because both branches end at the same k . A loop compiles its body against its own head, so the compiled counting loop of Figure 2.1 has no node t and no edge $t \rightarrow h$. A **skip** branch adds no node, and its guard edge leads to k .

Calls and returns leave the current fragment. A call $x = q(\bar{e})$ of a declared procedure q adds no local edge. It becomes a single call edge in `calls`, from the call site n to the callee's entry, with k as its continuation. A library call finishes in one step (Section 3.1) and compiles to a local edge to k instead. A **return** becomes an `EA_Ret` edge to the result node of its procedure, and a body that can reach its end without a **return** needs such

an edge as well. `compile_proc` therefore compiles the body c of a procedure p against an extra node r and connects both to the procedure's own nodes,

$$\text{compile_proc}(p) = \left\{ \text{FunctionEntry } p \xrightarrow{\text{body}} \text{entry}(c) \right\} \cup \text{compile}(c, r) \\ \cup \left\{ r \xrightarrow{\text{return}} \text{FunctionResult } p \right\}.$$

The `EA_Body` edge marks the start of an activation and changes no store. The return edge from r carries no value. It is emitted for every body, so in a body that always returns, r and its edge are dead code. Section 9.3 explains why the compiler keeps them. Every edge into `FunctionResult` p is therefore a return edge.

These rules give every compiled graph a fixed shape. Call edges always enter an entry node, no local edge enters one, and every return edge ends at the result node of its own procedure. The structural contract `wf_cfg` collects these properties, and every compiled graph satisfies it without premises (`compile_prog_wf`). The graph's entry is `FunctionEntry` `main` (`cfg_entry_compile_prog`), and its edge sets are finite (`compile_prog_finite`). Later chapters rely on this shape to route calls (Chapter 8) and to enumerate edges in the executable analyzer. Figure 3.4 shows the graph compiled from a recursive program.

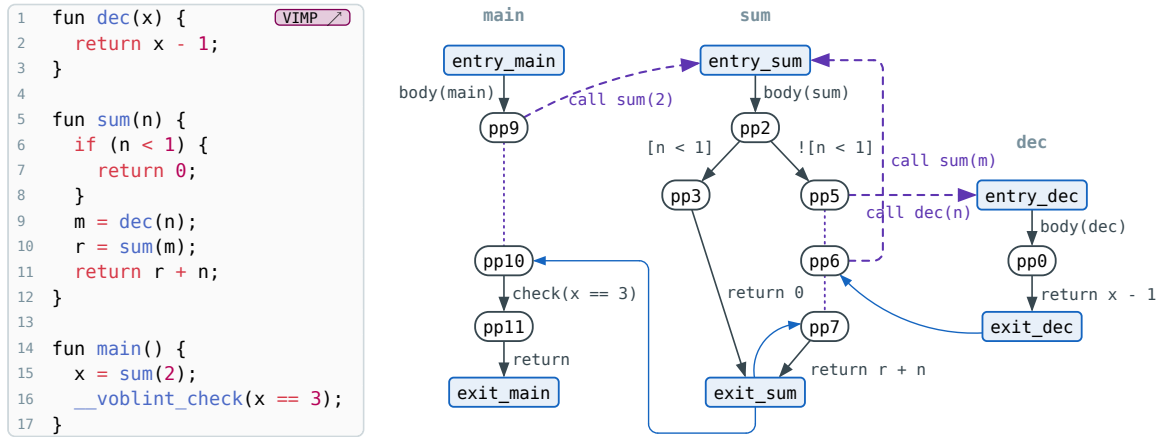


Figure 3.4: A recursive program and its compiled graph, labeled as `voblnt --dot` prints it. Here `pp $n$` is `Statement $n$` , and `entry $_p$` , `exit $_p$` are `FunctionEntry $p$` , `FunctionResult $p$` . As in Figure 3.1, solid gray arrows are `intra` edges, and dashed purple arrows are `calls` edges. Dotted connectors and thin blue arrows are not edges. Dotted connectors show the continuation of a call edge. Thin blue arrows show where execution continues after a result node. Each procedure has one copy of its nodes, whatever the recursion depth. Reserved node numbers without edges, such as `pp4`, are not nodes of the graph.

A whole program is compiled one procedure after another, each over a fresh block of node numbers, with `main` last (`compile_prog`). The compiler therefore needs the procedures in some order. The procedure table Π cannot provide one, because it is a function from names to declarations, and neither the logic nor generated code can enumerate its domain. Instead, `compile_prog` takes, besides Π , a list ps of the procedures to compile, while `main` is compiled separately from `main_body`. The executable analyzer reads this list off the parsed declarations (`prog_procs`). The list must agree with the table. If it misses a declared procedure, a call to it reaches an entry node that no edge leaves, and the graph execution stops where

the source execution continues. If it names a procedure twice, the procedure is compiled into two blocks that share its entry node, and the node blocks of different procedures are no longer disjoint, which later proofs rely on (`compile_procs_member_frag`). The source contract of the compiler, `wf_compile_input` $\mathcal{G} \models ps$, therefore adds to `wf_source_program` that ps lists every declared procedure except `main` exactly once.

3.4.2. Executing a graph with a frame stack thy

The compiler produces a static object. To say that it preserves the behavior of a program, the graph needs an execution of its own, which Section 3.5 then compares with `pstep`. Within one procedure, an execution follows edges, so a node and a store would describe its state. Calls break this picture. A call edge leads to the callee's entry, but the callee ends at its result node, which all callers share and which no edge leaves (Section 3.3.2). The graph alone thus cannot say where execution continues after a return. Graph execution therefore keeps a stack of suspended callers, like the call stack of a machine. A call pushes a frame that remembers the continuation k of its call edge, the variable that receives the result, and the caller's store, whose local variables the return restores. A return pops the frame and continues at k .

Definition 3.5 (Graph execution, inductive `cstep`). A graph configuration (`cconf`) is a triple of a node, a store, and a stack of frames (`cframe`), each a triple of a continuation node, an optional destination, and the caller's store. The step relation `cstep` is written $\mathcal{G}, g \vdash \kappa \rightarrow_c \kappa'$, and $\rightarrow_c^*$ is its closure.

```

1 (* src/Program_Model/CFG/CFG_Exec.thy *)
2 inductive cstep :: "(vname ⇒ bool) ⇒ cfg ⇒ cconf ⇒ cconf ⇒ bool"
3   ("(_ , _ ⊢ _ →c _)" [51, 51, 51, 51] 50) for  $\mathcal{G}$  and  $g$  where
4   Intra [intro]:
5     "(u, a, v) ∈ intra  $\mathcal{G} \implies s' \in \text{edge\_step } a \ s \implies$ 
6      $\mathcal{G}, g \vdash (u, s, \text{stk}) \rightarrow_c (v, s', \text{stk})"$ 
7   | Call [intro]:
8     "(u, CallEdge dst pars actuals, FunctionEntry q, cont) ∈ calls  $\mathcal{G} \implies$ 
9      $\mathcal{G}, g \vdash (u, s, \text{stk})$ 
10     $\rightarrow_c (\text{FunctionEntry } q, \text{call\_enter } \mathcal{G} (\text{CallEdge } \text{dst } \text{pars } \text{actuals}) \ s, (\text{cont}, \text{dst}, s) \# \text{stk})"$ 
11   | Return [intro]:
12     " $\mathcal{G}, g \vdash (\text{FunctionResult } q, t, (\text{cont}, \text{dst}, \text{caller}) \# \text{stk})$ 
13     $\rightarrow_c (\text{cont}, \text{combine\_collect } \mathcal{G} \ \text{dst } \text{caller } t, \text{stk})"$ 

```

The three rules above follow a local edge, a call edge, or a return. In the running example (Figure 3.1), the call `bump(5)` pushes a frame, runs the body of `bump` with $n = 5$, and pops the frame at the result node, after which `main` continues with `a` holding 6. Calls and returns change the store through `call_enter` and `combine_collect`, which are built from the same functions as `pstep` (Section 3.2), so both executions can hold the same store. Like the trace semantics of Chapter 4, `cstep` is defined for an arbitrary graph, not only for compiled ones.

3.5. Compilation preserves source executions

So far, `pstep` and `cstep` are two separate definitions. One describes source programs, the other arbitrary graphs, and nothing yet says that a compiled graph behaves like its program. The soundness theorem is stated over `pstep`, but the analysis solves equations generated from the graph, so every source execution must be matched by an execution of the compiled graph that holds the same store at corresponding points. This is proved with

a *forward simulation*, a relation between source and graph configurations that every source step preserves.

The relation cannot simply pair each command with a node. When one procedure calls another, the source keeps the caller's remaining work inside the running command, behind a **Restore** marker, while the graph keeps it on the frame stack as the continuation node k . The relation therefore has to match a nested command against a stack of frames. The two executions also do not step in lockstep. Some source steps only rearrange the command, such as unfolding a loop into a conditional, discarding a finished **skip**, or propagating **Unwind** past the commands it skips. They have no graph counterpart, so the graph answers one source step with zero or more steps. Only this direction is proved. An analysis that covers every graph execution from the entry (Chapter 4) then also covers every source execution, so soundness needs no converse.

3.5.1. Simulation relation and step matching

Definition 3.6 (Simulation relation, inductive **csim**). **csim**, written $\Pi, g, p \vdash (c, s, \text{frs}) \approx (v, s, \text{stk})$, relates a source configuration to a graph configuration holding the same store, where p is the procedure of the outermost activation, **main** for a program run.

```

1 (* src/Program_Model/Compile/Simulation/Simulation_Relation.thy *)
2 inductive csim :: "'a ptable ⇒ cfg ⇒ pname ⇒ 'a acom × store × frame list
3   ⇒ cfg_node × store × cframe list ⇒ bool"
4   ("(_ , _ ⊢ / _ ≈ / _)" [51, 51, 51, 51, 51] 50) for Π g where
5   Base:
6     "control_at Π p body_p k n c v ⇒ compiled_at Π g p body_p k n ⇒
7     Π, g, p ⊢ (c, s, []) ≈ (v, s, [])"
8   | Nested:
9     "Π, g, q ⊢ (inner, s, frs) ≈ (v, s, stk) ⇒
10    control_at Π pc body_pc kc nc (seq_after (SKIP i) afters) cont ⇒
11    control_at Π pc body_pc kc nc (seq_after (Call i dst q args) afters) site ⇒
12    compiled_at Π g pc body_pc kc nc ⇒
13    Π, g, pc ⊢ (seq_after (Seq inner (Restore i)) afters, s, frs @ [Frame caller dst])
14    ≈ (v, s, stk @ [(cont, dst, caller)])"
15   | Returning:
16     "pop_ready w ⇒ pop_ann w = Some i ⇒
17     control_at Π pc body_pc kc nc (seq_after (SKIP i) afters) cont ⇒
18     control_at Π pc body_pc kc nc (seq_after (Call i dst q args) afters) site ⇒
19     compiled_at Π g pc body_pc kc nc ⇒
20     Π, g, pc ⊢ (seq_after w afters, callee, [Frame caller dst])
21     ≈ (FunctionResult q, callee, [(cont, dst, caller)])"

```

The relation has three rules. Each rule relates one activation at a time. Its residual command must be what remains of a procedure body, and the graph node must be where the compiler placed that residual (**control_at**, **compiled_at**). **Base** covers an execution without suspended callers, where both stacks are empty. **Nested** adds one suspended caller to both sides. In the source, the caller's remaining work follows the **Restore** marker in the residual command. In the graph, it lies at the continuation node of the new frame. A further premise locates the call that suspended the caller, so that the inner activation runs the body of the procedure this call names. **Returning** covers the moment after a callee finishes. The graph is already at the callee's result node, while the source still has to reach the **Restore** marker (**pop_ready**).

The relation is useful only if it still holds after every step of the source execution. Suppose the source is about to run $x = x + 1$; and the relation places it at node v . The source executes the assignment and continues with the rest of its command. The graph follows the assignment edge out of v to the node where the compiler placed that rest, and both

now hold the same new store. When the source only unfolds a loop into a conditional, the graph takes no step at all. The following theorem states that the graph can always follow in this way.

Theorem 3.7 (One step is matched, `csim_step`). *If a source configuration and a graph configuration are related by `csim` and the source takes one step, the graph takes zero or more steps to a configuration related to the new source configuration.*

```
1 (* src/Program_Model/Compile/Simulation/Simulation_Preservation.thy *) Isabelle
2 theorem csim_step:
3   assumes SIM: " $\Pi, g, \varrho \vdash (c, s, \text{frs}) \approx (v, t, \text{stk})$ "
4     and PC: "procs_embedded  $\Pi$  g"
5     and WF: "return_safe c"
6     and STEP: " $\mathcal{G}, \Pi \vdash (c, s, \text{frs}) \rightarrow_p \text{src}'$ "
7   shows " $\exists \text{cfg}'. \mathcal{G}, g \vdash (v, t, \text{stk}) \rightarrow_c^* \text{cfg}' \wedge \Pi, g, \varrho \vdash \text{src}' \approx \text{cfg}'$ "
```

The theorem has two premises beyond the related pair and the source step. `procs_embedded  $\Pi$  g` says that the body of every declared procedure lies in g with its entry and return edges. `return_safe c` says that the residual command returns only inside an activation that a call opened, the runtime form of the rule that `main` contains no `return` (Section 3.2). Both premises hold for every well-formed program (`procs_embedded_compile_prog, wf_compile_input_return_safe`), and every source step preserves the second (`return_safe_pstep`). By induction on the run, `csim_star` extends the theorem to finite source executions, as in CompCert’s simulation diagrams [21, §4.3].

The initial source configuration is related to the node after the `EA_Body` edge of `main` (`compile_prog_main_base`). Every finite source execution of a well-formed program is therefore matched by a graph execution from the entry that reaches corresponding nodes with the same store. This gives the first inclusion of the soundness chain of Figure 1.3.

3.5.2. Locating source configurations at graph nodes (thy)

The simulation names no node for a source configuration taken on its own. Chapter 10 needs one, because the analyzer’s verdicts are stated at graph nodes, while a source configuration carries only its residual command. The residual does not determine the node either, because equal commands can occur at different places. In `if (b) { x = 1; } else { x = 1; }`, the residual `x = 1` is related to both branch nodes. The node must come from the run.

Every command therefore carries an annotation (Section 3.1). A parsed program leaves it empty, and the proof fills it in. The program’s *elaboration* (`elab_pt`) annotates each statement with the nodes the compiler emits for it. The elaboration compiles to the same graph (`compile_prog_elab_pt`), and on it, `point` reads the related node off the residual’s annotation (`csim_point_compile_prog`). Every source execution of a well-formed program is the *erasure* (`erase`) of a run of its elaboration (`psteps_lift`), so the node can be read off the elaborated run.

Definition 3.8 (Observed at a node, `fun at_node`). A configuration (r, s, frs) is *observed* at v , written $p, s_0 \vdash (r, s, \text{frs}) \triangleright v$, if a run of the elaboration of p from s_0 reaches an annotated configuration that erases to it, names v , and is related to v by the simulation. A configuration reachable along two paths can be observed at two nodes.

3.6. Adequacy of VIMP relative to C

Every source-level guarantee in this thesis is interpreted against the executions of `pstep`, the source semantics of VIMP, so no guarantee can show that `pstep` describes the intended language. This can only be argued, and this section collects the argument. Table 3.1 compares VIMP with C11 [7] (committee draft N1570) wherever both languages have a construct but model it differently. The comparison is not complete, so C may differ from VIMP in further places.

	VIMP	C11
integers	unbounded	signed overflow undefined (§6.5p5) unsigned arithmetic wraps (§6.2.5p9)
$a / 0, a \% 0$	0 and a	undefined (§6.5.5p5)
$/, \%, b \neq 0$	truncated; $(a / b) \cdot b + a \% b = a$	the same where a / b is representable (§6.5.5p6)
<code>&&, </code>	both operands evaluated	second operand only if needed (§6.5.13p4, §6.5.14p4)
initial values	globals 0, <code>main</code> 's locals arbitrary	static 0, automatic indeterminate (§6.7.9p10)
callee locals	0 on every entry	indeterminate (§6.7.9p10)
checks	never stop the execution, never refine	<code>assert</code> aborts on 0 unless <code>NDEBUG</code> (§7.2p1, §7.2.1.1)
<code>min, max</code>	library calls (<code>special_table</code>)	no counterpart
<code>main</code>	no explicit <code>return</code>	<code>return</code> gives the exit status (§5.1.2.2.3)

Table 3.1: Constructs that both languages have but model differently. The VIMP rows restate `c_div`, `c_mod`, `aval`, `cinit_stores`, `enter_state` and `pstep`.

In each row, a statement true of a VIMP program may fail for the corresponding C program. Unbounded integers and defined division by zero can flip a verdict. After `z = 5 / 0`; the check `z == 0` holds in VIMP, so a `PROVED` verdict on a program that divides by zero says nothing about C. The analyzer therefore reports a possible zero divisor as a separate diagnostic, and its absence at a reached node proves the divisor nonzero there (`run_voblint_arithmetic_safe`, Section 10.5). Evaluating both operands of `&&` gives the short-circuit value, because expressions are total and have no effects.

The initial values that VIMP fixes are ones C leaves open. Every source-level theorem assumes an initial store from `cinit_stores`, with globals 0 as for C's static objects and the locals of `main` arbitrary. VIMP does not model C's indeterminate values, some reads of which are undefined (§6.3.2.1p2). Callee locals start at 0 (`enter_state`). Checks never prune executions, so the verdict for one check does not depend on an earlier one.

A fragment of Goblint's input. Goblint analyzes C after its CIL front end [47] has normalized it, so that expressions have no side effects and a call is a separate instruction with an optional destination (`cil.mli` at the revision Goblint pins). VIMP makes the same choice, and every edge kind of Goblint's CFG, except inline assembly and in-place declarations, has a VIMP counterpart (`edge.ml`).

No second semantics. No external semantics cross-checks `pstep`, such as the Simpl programs that the C parser of AutoCorres2 [48] produces [45] or CompCert's Clight [21]. Relating VIMP to them would need a translation with its own adequacy argument.

These differences limit what a verdict says about a C program, but the analysis of VIMP still answers the question this thesis asks. The subject of the formalization is the analyzer architecture, and VIMP has the features the architecture has to handle: procedures, recursion, locals and globals, and calls with results. Since VIMP also has the shape of Goblint's normalized input, a C front end with a verified translation could reuse the framework, although each omitted feature would need its own preservation argument.

4. Traces and the Coverage Contract

Chapter 3 matched every source execution with a graph execution. For an analysis that computes one abstract state per node, this suffices. A context-sensitive analysis, however, computes a separate state for each node and calling context, so justifying its result requires knowing which executions each context stands for. This chapter gives that distinction a concrete meaning. It groups executions into *activation traces*, reads calling contexts off them, and states the *coverage contract*, the local conditions under which an analysis result contains every store that an execution brings to a node in a context.

4.1. Why calling contexts need activation traces

A context-sensitive analysis splits the unknown $[v]$ of a node into unknowns $[v, c]$, one per calling context c (Section 2.6). Seidl et al. state that the invariant for $[u, c]$ “only need[s] to take into account executions reaching u that satisfy the restriction imposed by context c ” [2, §4]. The abstract state for $[v, c]$ therefore has to describe only a subset of the stores that the state for $[v]$ describes, and it can be correspondingly more precise.

For the calls `bump(5)` and `bump(4)` of the running example (Figure 3.1), an analysis with entry-state contexts claims the result 6 in the context of the first call and 5 in that of the second (Section 2.6). To justify these claims, it must be known which store at the result node of `bump` belongs to which call. The collecting semantics, however, is too weak to tell. It holds both stores at the result node but does not record which activation reached each of them, so a richer semantics that keeps this information is needed.

Contexts depend on how an activation was created, for example, on its call site or arguments and on the context of its caller (Section 4.3). After a return, the caller must continue in the contexts it had before the call. Neither the graph execution nor a flat record of its runs exposes this. Each frame of the stack in Section 3.4.2 holds only what execution needs to resume its caller, and the frame disappears once the call has returned. The list of node-store pairs (v, s) that a run passes through (Figure 4.1, top) interleaves the steps of all activations. In the program of Figure 3.4, `sum(2)` calls `dec` and then itself, so one run visits the nodes of `sum` in three activations, and the list does not say which call created each of them or which return resumes which caller.

Grouping the same run by activation, in contrast, makes these relationships explicit (Figure 4.1, bottom). Each call of a procedure starts a new *activation*, which lasts until the call returns [38, §7.2.1], and each activation has its own activation path. A call creates a new activation attached to its caller, and a return composes the finished callee back into that caller.

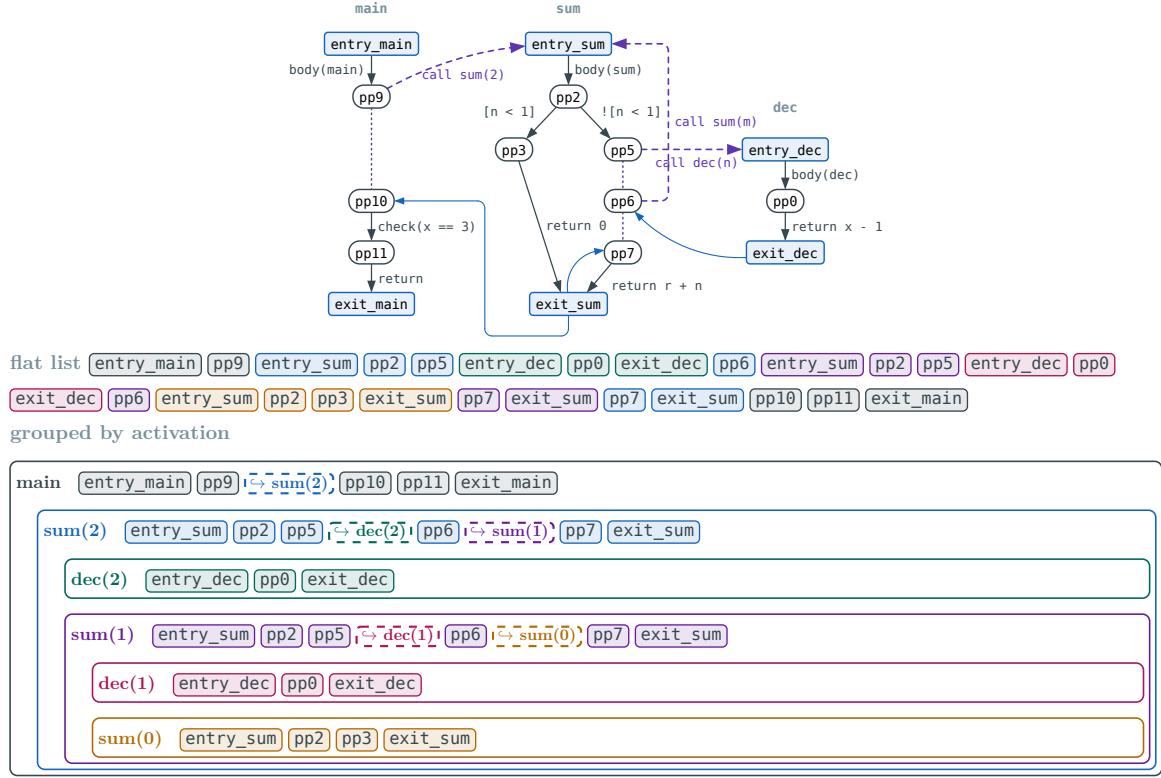


Figure 4.1: The compiled graph of the program in Figure 3.4 (top) and one run on it, as the flat list of its steps and grouped by activation. Colors mark each step’s activation, which the flat list itself does not record. A dashed tag $\hookrightarrow$ marks a call of the named callee. The run is written out by hand, and stores are omitted.

4.2. From graph executions to activation traces

Activation traces adapt the *local traces* of threads [5] to procedure activations.

4.2.1. Grouping a run by activation thy

The grouping of Figure 4.1 becomes a datatype with one constructor per way a box arises. The outermost box, the activation of `main`, is a root. A box opened by a call is a callee trace and remembers its caller. When a callee finishes, its caller continues in a new trace that remembers both the caller at the call and the finished callee.

Definition 4.1 (Activation trace, datatype `activation_trace`). A trace has one constructor per way an activation begins or continues. Its `activation_path` is the path of one activation, a list of pairs of a CFG node and a store.

```

1 (* src/Program_Model/CFG/Collecting/Activation_Trace_Def.thy *)
2 type_synonym activation_path = "(cfg_node × store) list"
3
4 (* src/Program_Model/CFG/Collecting/Activation_Trace_Def.thy *)
5 datatype activation_trace =
6   Root activation_path
7   | Call (activation_trace_caller: activation_trace) activation_path
8   | Resume (activation_trace_current: activation_trace) (activation_trace_callee: activation_trace)
9   activation_path
  
```

Root π is the initial activation of the program with activation path π . A run starts with **Root** $[(\text{entry}, s_0)]$, the entry node of the graph, which is the entry of `main`, paired with an initial store s_0 . **Call** $t' \pi$ is a callee created by the caller t' . Its activation path begins at the callee's entered store. **Resume** $t' t'' \pi$ is the activation t' continued past the call that produced the finished callee t'' . The observer `path_of t` returns the activation path, `sink_node t` the node of its last entry, `sink_store t` that entry's store, and `caller_of t` the caller that created the activation, which a **Root** does not have. The [explainer page](#) shows these observers interactively on a run of the program of Figure 4.1.

Each trace represents the history of one activation up to a program point, so on well-formed compiled programs, its activation path stays inside the nodes of one procedure (`valid_activation_trace_frag_callers`). A callee is a new activation and gets a trace of its own, which holds its caller as a field. When the callee finishes, a **Resume** creates a new trace for the resumed caller, which holds both the frozen caller and the finished callee. A whole run is therefore a family of traces linked by these fields.

Because a **Call** stores its caller exactly, frozen at the call site, a finished callee composes back into the activation that created it, and no search for a compatible stack frame is needed. This is what makes context-specific return flow sound. A return continues exactly the caller that created the callee, in each context that caller carries, so the exit state of a callee entered in context c' flows back only to the callers whose calls admitted c' . The abstract state for a context therefore needs to describe only the executions in that context.

This design follows the local traces of Schwarz et al. [5, §3]. A local trace is the view of one thread on a concurrent execution and ends at a program point with that thread's local state. The set of local traces is defined inductively, from initial traces, a rule that starts a created thread with the local state of its creator, and one rule per edge that extends a trace by a step. For a lock, this rule also incorporates a second trace, one that ends in an unlock of the same mutex. Activation traces take over this scheme with an activation in place of a thread. **Root** corresponds to an initial trace, **Call** to thread creation, a local step to a unary edge rule, and **Resume** to the binary lock rule, which here incorporates the finished callee. The traces themselves differ. A local trace is an acyclic graph of events from several threads, ordered by program order and by the creation and locking orders, and a lock may incorporate a trace of whichever thread last released the mutex. An activation trace holds the path of a single activation and refers to at most two other traces, and a **Resume** accepts only the callee that this activation itself created. Activations do not run concurrently, so nothing corresponds to reading globals written by another thread.

4.2.2. Valid traces follow graph steps thy

The datatype also admits terms that no execution produces. A path may contain a step that follows no edge of the graph, for instance, a jump from the entry of `sum` directly to its result node. A **Resume** may pair a caller with a callee that another activation created, for instance, continue `main` after its call `sum(2)` with the finished inner activation `sum(1)`, whose caller is `sum(2)`. Validity rules out such terms by defining how traces arise. A trace is valid if it can be built from an initial store by rules that mirror the steps of graph execution, one for starting the program, one for a local edge, one for a call, and one for a return. Every step that a valid trace records then follows an edge of the graph, and every **Resume** combines a caller with the callee that it created itself.

Definition 4.2 (Valid traces, inductive_set `valid_activation_trace`). For a global-variable classifier $\mathcal{G}$, a graph g , and a set S of initial stores, $\mathcal{T}_{\mathcal{G},g,S}$ (`valid_activation_trace`) is the least set of traces closed under the following four rules.

```

1  (* src/Program_Model/CFG/Collecting/Activation_Trace_Def.thy *)
2  inductive_set valid_activation_trace ::
3    "(vname  $\Rightarrow$  bool)  $\Rightarrow$  cfg  $\Rightarrow$  store set  $\Rightarrow$  activation_trace set"
4    ("T_,_,_")
5    for  $\mathcal{G}$  and  $g$  and  $S$  where
6    root:
7      "s  $\in S$ 
8       $\Rightarrow$  Root [(cfg_entry  $g$ , s)]  $\in \mathcal{T}_{\mathcal{G},g,S}$ "
9    | intra:
10      "t  $\in \mathcal{T}_{\mathcal{G},g,S}$ 
11       $\Rightarrow$  (sink_node t, a, v)  $\in$  intra  $g$ 
12       $\Rightarrow$  s'  $\in$  edge_step a (sink_store t)
13       $\Rightarrow$  extend t (v, s')  $\in \mathcal{T}_{\mathcal{G},g,S}$ "
14    | call:
15      "caller  $\in \mathcal{T}_{\mathcal{G},g,S}$ 
16       $\Rightarrow$  (sink_node caller, CallEdge dst pars args, FunctionEntry p, cont)
17           $\in$  calls  $g$ 
18       $\Rightarrow$  Call caller
19          [(FunctionEntry p,
20            call_enter  $\mathcal{G}$  (CallEdge dst pars args) (sink_store caller)))]
21           $\in \mathcal{T}_{\mathcal{G},g,S}$ "
22    | ret:
23      "callee  $\in \mathcal{T}_{\mathcal{G},g,S}$ 
24       $\Rightarrow$  caller_of callee = Some caller
25       $\Rightarrow$  sink_node callee = FunctionResult p
26       $\Rightarrow$  (sink_node caller, CallEdge dst pars args, FunctionEntry p, cont)
27           $\in$  calls  $g$ 
28       $\Rightarrow$  Resume caller callee
29          (path_of caller
30            @ [(cont, combine_collect  $\mathcal{G}$  dst
31              (sink_store caller) (sink_store callee)))]
32           $\in \mathcal{T}_{\mathcal{G},g,S}$ "

```

The root rule corresponds to the initial graph configuration, and the other three rules correspond to the three rules of `cstep`. Intra appends a step along a local edge. Call starts a new trace that holds the caller, where `cstep` pushes a frame. Ret builds a third trace that holds the caller and the callee, where `cstep` pops the frame.

The ret rule applies when a callee trace has reached a result node `FunctionResult p`, which no edge leaves in a compiled graph. The rule finds the caller through `caller_of callee = Some caller` and takes the `calls` edge from its call site into p . That edge names the continuation $cont$, where the caller continues with the store that `combine_collect` builds from both final stores.

4.2.3. Every source execution yields a valid trace (thy)

The rules of valid traces mirror the rules of `cstep`, but nothing so far states that a graph execution actually yields a valid trace. Graph configurations are therefore related to traces. Besides the node and the store, the relation matches the trace's callers with the frame stack, so that it still holds after a return.

Definition 4.3 (Trace representation, definition `activation_trace_repr`). A valid trace represents a graph configuration if it ends at the configuration's node and store, and its chain of creating callers matches the runtime frame stack.

```
1 (* src/Program_Model/Compile/Source_To_Trace.thy *)
2 inductive stack_repr :: "cfg ⇒ cframe list ⇒ activation_trace ⇒ bool" for g where
3   empty [intro]: "caller_of t = None ⇒ stack_repr g [] t"
4 | frame [intro]: "caller_of t = Some caller ⇒ sink_store caller = s
5   ⇒ fst (hd (path_of t)) = FunctionEntry p
6   ⇒ (sink_node caller, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
7   ⇒ stack_repr g stk caller
8   ⇒ stack_repr g ((cont, dst, s) # stk) t"
```

```
1 (* src/Program_Model/Compile/Source_To_Trace.thy *)
2 definition activation_trace_repr ::
3   "(vname ⇒ bool) ⇒ cfg ⇒ store set ⇒ cconf ⇒ activation_trace ⇒ bool" where
4   "activation_trace_repr G g S cf t = (case cf of (v, s, stk) ⇒
5     t ∈  $\mathcal{T}_{G,g,S}$  ∧ sink_node t = v ∧ sink_store t = s ∧ stack_repr g stk t)"
```

For a well-formed compiled program (`wf_compile_input`), the correspondence is lock-step. A local graph step extends the current trace, a call creates a `Call` trace, and a return creates a `Resume` trace (`cstep_preserves_activation_trace_repr`). The initial configuration has a representing trace, a `Root` trace (`located_activation_trace_entry`). Hence, every configuration that a graph execution of such a program reaches has a representing valid trace (`csteps_preserve_located_activation_trace`). The converse is neither proved nor needed.

Theorem 4.4 (Source executions are traces, `source_run_has_activation_trace`). *For a well-formed compiled program, every finite source execution from an initial store in S has a matching graph configuration and a valid trace ending at the same node with the same store, whose caller chain represents the graph's frame stack.*

```
1 (* src/Program_Model/Compile/Source_To_Trace.thy *)
2 theorem source_run_has_activation_trace:
3   assumes wf: "wf_compile_input G  $\Pi$  ps"
4   and s0: "s0 ∈ S"
5   and run: "G,  $\Pi$  ⊢ (main_body  $\Pi$ , s0, []) →p* (residual, s, frs)"
6   shows "∃v stk t.  $\Pi$ , compile_prog  $\Pi$  ps, prog_main_name ⊢ (residual, s, frs) ≈ (v, s, stk)
7     ∧ activation_trace_repr G (compile_prog  $\Pi$  ps) S (v, s, stk) t"
```

The proof composes the simulation of Section 3.5 (`csim_star`) with the lock-step correspondence above. The source configuration, the graph configuration, and the trace hold the same store, so every store a source execution reaches is the final store of a valid trace.

4.2.4. Collecting trace stores per node [thy](#)

Collecting the final stores of the valid traces per node gives the *node collecting semantics*. It plays the role of the collecting semantics of Chapter 2 for programs with procedures, with valid traces in place of runs.

Definition 4.5 (Node collecting semantics, definition `node_collect`).

```
1 (* src/Program_Model/CFG/Collecting/Activation_Trace_Collect.thy *)
2 definition node_collect ::
3   "(vname ⇒ bool) ⇒ cfg ⇒ store set ⇒ cfg_node ⇒ store set"
4   ("C_._" where
5     "C_ $G,g,S$  v =
6     {sink_store t | t. t ∈  $\mathcal{T}_{G,g,S}$  ∧ sink_node t = v}"
```

A store is in $C_{G,g,S} v$ exactly when some valid trace ends at v with this store. The classifier, graph, and initial stores are those of the program at hand, written G , g , and S from here

on. A procedure's result is an ordinary collected node, and the stores in which a run of the whole program completes are those collected at `FunctionResult main`. Every store that a finite source execution of a well-formed program reaches therefore lies in the node collecting semantics at a corresponding node (`source_reaches_node_collect`).

The node collecting semantics forgets the traces, so a soundness statement over it alone is context-insensitive. The valid traces themselves, however, keep their structure. The same set of traces thus supports two views: collected per node, and split by context (Section 4.3).

4.3. Assigning calling contexts to traces thy

A context-sensitive analysis of the running example claims the result 6 for `bump` in the context of the call `bump(5)` and 5 in that of `bump(4)`. Checking such claims requires knowing which executions each context stands for. This section assigns contexts to activation traces and uses them to split the stores collected at each node.

Contexts are chosen at calls. When `main` calls `bump(5)`, the analysis decides in which context it analyzes the callee. With call strings of length one, the context is the call site in `main`. With entry-state contexts, it is the abstract value of the formal parameter at the callee's entry, `[[5, 5]]` under Interval (Section 8.2). In general, the choice may depend on the call site, on the caller's context, which call strings extend, on the call itself, which names the callee, its formals, arguments, and destination, and on the caller's store and the *entered store*, the store at the callee's entry after the formals have been bound. A *context policy* fixes this choice for every concrete call.

Definition 4.6 (Context policy, type_synonym `context_policy`). A context policy maps a concrete call to the set of callee contexts it admits.

```
1 (* src/Program_Model/CFG/Collecting/Activation_Trace_Context.thy *)
2 type_synonym 'c context_policy =
3   "cfg_node ⇒ 'c ⇒ call_info ⇒ store ⇒ store ⇒ 'c set"
```

Isabelle

A policy returns a set of contexts rather than one, because an analysis may split a call into cases. Goblint's entry operation may return several alternatives for one call, each a pair of a resume state and an entry state, and each alternative gets its own context. Suppose a caller passes an argument that is either negative or positive, and its state keeps the two cases apart, for instance, as two path-sensitive states. With one alternative per case, the callee is analyzed in one context for negative arguments and in another for positive ones, and a sign analysis keeps the sign in both, whereas a single entry state would join the cases to $\top$. When the cases overlap, one concrete call belongs to several contexts at once (`ov_two_contexts_admitted`).

The set may also be empty, for instance, at a call that the analysis considers unreachable. An entry-state policy admits no context there, because the analysis's state at the call site describes no caller store (Section 8.2). Stores that the analysis result excludes need no context. At a call that the program actually makes, however, an empty set would lose the callee, which the coverage contract rules out (Section 4.4).

A policy speaks about a single call. The contexts of a whole trace follow by applying it along the calls that created the trace. The root activation carries the initial context c_0 . A callee carries every context c' that the policy admits for its call from some context c of its caller (`admits_call_context`, which passes the policy the caller's store and the entered store that `call_enter` computes from it). A local step keeps the contexts of the

trace (`activation_context_rel_extend`), and a resumed caller carries the contexts it had before the call (`activation_context_rel_Resume_iff`). An activation may therefore carry several contexts or none. The [explainer page](#) draws, for each policy, the contexts that the analyzer creates for one program.

Definition 4.7 (Context of a trace, inductive `activation_context_rel`). The contexts that a trace may carry.

```

1 (* src/Program_Model/CFG/Collecting/Activation_Trace_Context.thy *)
2 inductive activation_context_rel ::
3   "(vname ⇒ bool) ⇒ 'c context_policy ⇒ 'c ⇒ cfg ⇒ activation_trace ⇒ 'c ⇒ bool"
4   for g :: "vname ⇒ bool" and adm :: "'c context_policy" and c₀ :: "'c and g :: cfg
5   where
6     Root [intro]: "activation_context_rel g adm c₀ g (Root xs) c₀"
7   | Call [intro]:
8     "activation_context_rel g adm c₀ g caller c
9     ⇒ admits_call_context g g adm (sink_node caller) c p (sink_store caller) es c'
10    ⇒ activation_context_rel g adm c₀ g (Call caller ((FunctionEntry p, es) # xs)) c'"
11   | Resume [intro]:
12     "activation_context_rel g adm c₀ g current c
13     ⇒ activation_context_rel g adm c₀ g (Resume current callee xs) c"

```

Voblint offers three policies (`context_mode`): the unit context policy with one context for every activation, call strings bounded to the k most recent call sites, and entry-state contexts [31, §§2.6, 2.9], [39]. Entry-state contexts follow Goblint’s partial tabulation, which tabulates procedure summaries only for the calling contexts the analysis meets [27, §3.3], [49, §5, p. 110]. Unlike bounded call strings, they can form an infinite set, so the analysis may not terminate (Section 9.3). The traces themselves store no context, since the concrete semantics should not change with the analysis. The policy is instead a parameter of the theorems, which therefore hold for every policy. It may even be defined from the analysis result, as the entry-state policy is (Section 8.2). This is not circular, because the final guarantee concerns the stores collected per node, which no policy changes. The policy only decides how these stores are split among contexts.

The node collecting semantics can now be split by context. A store belongs to context c at node v if some valid trace ending at v with that store carries c .

Definition 4.8 (Activation collecting semantics, definition `activation_collect`).

```

1 (* src/Program_Model/CFG/Collecting/Activation_Trace_Context.thy *)
2 definition activation_collect ::
3   "(vname ⇒ bool) ⇒ 'c context_policy ⇒ 'c ⇒ cfg
4   ⇒ store set ⇒ cfg_node ⇒ 'c ⇒ store set"
5   ("A_{g,adm,c₀,g,S} v c =
6     {sink_store t | t. t ∈ T_{g,g,S} ∧ sink_node t = v
7     ∧ activation_context_rel g adm c₀ g t c}"
8

```

The set $A_{g,adm,c₀,g,S} v c$ is called the *activation collecting semantics* at v and c . It contains the final stores of the valid traces that end at v and carry c , and the abstract state computed for $[v, c]$ must describe it. In the running example under interval entry-state contexts, `bump(5)` gets the context $[[5, 5]]$ and `bump(4)` the context $[[4, 4]]$ (Section 8.2). The final store of `bump(5)`, with result 6, lies in the activation collecting semantics of context $[[5, 5]]$ at `bump`’s result node, and the final store of `bump(4)`, with result 5, in that of context $[[4, 4]]$. The node collecting semantics there contains both stores.

Splitting by context makes the result more precise. A single abstract state for the result node must describe both stores, so an interval analysis can at best bound the result by $[5, 6]$. A state per context only has to describe the stores of its own context, 6 in $[[5, 5]]$

and 5 in $[[4, 4]]$. No store is lost as long as every valid trace carries a context. An analysis therefore needs one set of stores per node and context that contains the activation collecting semantics, and the next section reduces this requirement to local checks.

4.4. The coverage contract and its soundness thy

An analysis result is sound if, at every node v and in every context c , it accounts for the stores of the activation collecting semantics $\mathcal{A}_{\mathcal{G}, \text{adm}, c_0, g, s} \vee c$. This is stated over a *cover*, a function `cover` that assigns a set of stores to each node and context. Since valid traces grow by only four rules, it suffices to check the cover against each rule once. The *coverage contract* does this with one local obligation per rule, each mentioning only stores. For a call edge from u into procedure p with continuation k , it requires

- INIT** $s \in S \implies s \in \text{cover}(\text{cfg_entry } g, c_0)$
- INTRA** $(u, a, v) \in \text{intra } g, s \in \text{cover}(u, c), s' \in \text{edge_step } a \implies s' \in \text{cover}(v, c)$
- CALL** $s \in \text{cover}(u, c), c' \text{ admitted from } s$
 $\implies \text{call_enter } (s) \in \text{cover}(\text{FunctionEntry } p, c')$
- RETURN** $s \in \text{cover}(u, c), c' \text{ admitted at } u \text{ from } s \text{ for } (p', e), t \in$
 $\text{cover}(\text{FunctionResult } p, c')$
 $\implies \text{combine_collect } (s, t) \in \text{cover}(k, c)$

In **CALL**, “ c' admitted from s ” means that the policy admits c' for this call edge from s in context c . **RETURN** must hold for every admitted c' , since one call may admit several contexts and the returning activation may carry any of them. Because c' ranges only over admitted contexts, the return of `bump(5)` combines only callee results of contexts entered from this call. In “ c' admitted at u from s for (p', e) ”, the admitting call edge may be any edge leaving u , into p' with entered store e (`admits_call_context`). A compiled graph has one call edge per call site (`calls_source_unique`, `compile_prog_calls_source_unique`), so on such a graph the admitting edge is always the call edge of the return. In the running example under entry-state contexts, **CALL** requires the cover at the entry of `bump` in context $[[5, 5]]$ to contain the entered store with $n = 5$, and **RETURN** requires the cover at the continuation in `main` to contain the store in which `a` holds 6. Figure 4.2 shows the obligations at one call, together with a fifth, **TOTAL**.

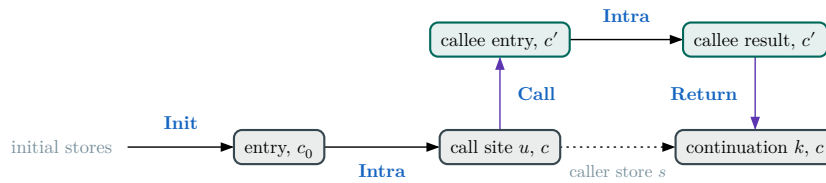


Figure 4.2: The obligations at one call (schematic). Boxes are (node, context) pairs, and each solid arrow is labeled with the obligation it represents. The dotted arrow is no obligation. It marks the caller store s that **RETURN** reads.

CALL and **RETURN** constrain only the contexts that the policy admits. If a call from a store in the cover gets no context at all, both hold vacuously, and no obligation requires the cover at the call’s continuation to contain the store the call returns with. The resumed caller, however, still carries its contexts there (`resume_keeps_context`), so the cover can

miss a store that the program reaches (**total_dropped_unsound**). **TOTAL** therefore requires the policy to admit at least one callee context for every call from a store $s \in \text{cover } u \ c$ (**call_context_total_on**). It asks only about stores in the cover, because entry-state policies admit no context for the stores that the analysis excludes (Section 4.3). Call strings and the unit context policy meet **TOTAL** for every cover (**call_context_total_on_of_fun**). For entry-state contexts, it follows from the coverage of the analysis's entry (Section 8.2).

Isabelle states the five obligations as the assumptions of the locale **activation_coverage**.

```

1 (* src/Program_Model/CFG/Collecting/Activation_Trace_Abstract.thy *)
2 locale activation_coverage =
3   fixes g :: cfg and S :: "store set"
4   and cover :: "cfg_node ⇒ 'c ⇒ store set"
5   and adm :: "'c context_policy"
6   and c₀ :: "'c"
7   and G :: "vname ⇒ bool"
8   assumes INIT[intro]: "∧s. s ∈ S ⇒ s ∈ cover (cfg_entry g) c₀"
9   and INTRA[intro]: "∧u a v c s s'. (u, a, v) ∈ intra g
10    ⇒ s ∈ cover u c ⇒ s' ∈ edge_step a s ⇒ s' ∈ cover v c"
11   and CALL[intro]: "∧u dst pars args p cont c c' s.
12    (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
13    ⇒ s ∈ cover u c
14    ⇒ c' ∈ adm u c (call_info_of (CallEdge dst pars args) p) s
15    (call_enter G (CallEdge dst pars args) s)
16    ⇒ call_enter G (CallEdge dst pars args) s ∈ cover (FunctionEntry p) c'"
17   and RETURN[intro]: "∧u dst pars args p cont c c' p' s t es.
18    (u, CallEdge dst pars args, FunctionEntry p, cont) ∈ calls g
19    ⇒ s ∈ cover u c
20    ⇒ admits_call_context G g adm u c p' s es c'
21    ⇒ t ∈ cover (FunctionResult p) c'
22    ⇒ combine_collect G dst s t ∈ cover cont c"
23   and TOTAL: "call_context_total_on cover adm G g"

```

The contract implies the goal by induction over the four rules of valid traces, one obligation per rule.

Theorem 4.9 (Context-indexed soundness, **activation_collect_sound**). *If the cover meets the five obligations of **activation_coverage**, then for every node v and context c , the activation collecting semantics at v and c is contained in $\text{cover } v \ c$.*

```

1 (* src/Abstract_Interpreter/Framework/Activation/Activation_Backbone.thy *)
2 theorem activation_collect_sound:
3   assumes "activation_coverage g S cover adm c₀ G"
4   shows "A_{g,adm,c₀,g,S} v c ⊆ cover v c"

```

The theorem bounds each context separately. Since **TOTAL** gives every valid trace a context, the sets of all contexts together contain the whole node collecting semantics:

Corollary 4.10 (The cover contains the collection, **node_collect_covered**). *Under the five coverage obligations, every store of the node collecting semantics at v lies in $\text{cover } v \ c$ for some context c .*

```

1 (* src/Program_Model/CFG/Collecting/Activation_Trace_Abstract.thy *)
2 corollary node_collect_covered: "C v ⊆ (∪c. cover v c)"

```

This completes the concrete half of the soundness argument. Every store that a finite source execution reaches appears in a graph execution (Chapter 3), as the final store of a valid trace (Section 4.2.3), in the activation collecting semantics of the trace's contexts (Section 4.3), and hence in every cover that meets the coverage contract. An analysis therefore never reasons about whole runs, only about sets of stores that each kind of step preserves. The following chapters build such analyses from abstract domains (Chapter 5), transfer functions (Chapter 6), and a solved equation system (Chapter 8, Chapter 9).

PART III

Building Analyses

5. Abstract Domains

Chapter 4 reduced soundness to five obligations on a cover, which assigns each node and context a set of stores. The obligations are local conditions, and together they guarantee that the cover contains every store that an execution in that context reaches at the node (`activation_collect_sound`). These sets are arbitrary and usually infinite. An analyzer, however, computes with finite abstract values, and its solver (Section 2.8) compares, joins, widens, and narrows them without knowing what they mean. Recall from Section 2.3 that an abstract value a denotes a set $\gamma(a)$ of concrete values. A value of a numeric domain such as Sign or Interval denotes a set of integers, $\gamma(a) \subseteq \mathbb{Z}$. An abstract state d , in contrast, denotes a set of stores, $\gamma(d) \subseteq (\text{vname} \rightarrow \mathbb{Z})$ (Section 5.2). An arbitrary lattice of abstract values is not enough, for two reasons. First, the order must agree with the concretization. The solver only proves inequalities $a \sqsubseteq b$ in the abstract order, while the obligations of Chapter 4 are inclusions between sets. The law

$$a \sqsubseteq b \implies \gamma(a) \subseteq \gamma(b)$$

turns each such inequality into an inclusion (`gamma_mono`, Figure 5.1). Second, a value can denote no integer without being the lattice's bottom element. The interval pair $[5, 3]$, whose lower bound exceeds its upper bound, is such a value. An analyzer that recognizes only the bottom element as empty misses it. This is still sound, since the analyzer then assumes more stores than exist, but it costs precision. After the next join, the information that a program point is unreachable is gone (Section 5.3).

This chapter derives what a domain must provide. The solver needs operations it can compute with, the soundness proof needs laws that turn the solver's inequalities into inclusions between sets of stores, and the analyzer needs to recognize every value that denotes no integer, so that an unreachable program point stays unreachable.

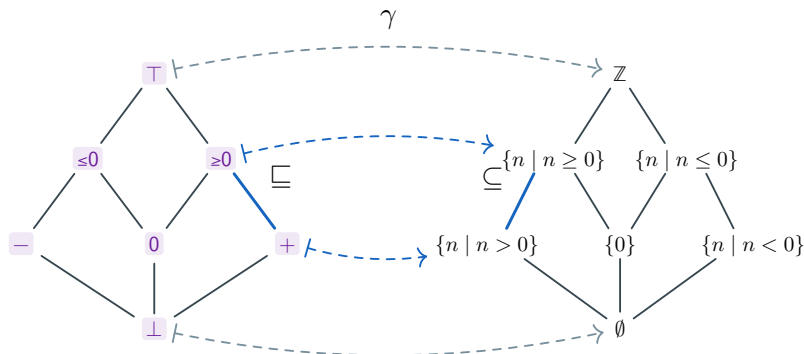


Figure 5.1: The Sign domain (left) and the sets of integers its values denote (right, mirrored), both ordered bottom to top. The dashed arrows show four instances of γ . Since γ is monotone, the blue step $+$ $\sqsubseteq$ ≥ 0 becomes the inclusion $\gamma(+) \subseteq \gamma(\geq 0)$. Adapted from [6, Fig. 13.5].

5.1. Numeric domains and their concretization laws [thy](#)

The two requirements of the chapter opening become laws on the carrier of a domain (Section 2.3), which Voblint represents as an Isabelle type A of abstract values. The solver compares its values, joins them where control flow merges, starts from the least value $\perp$, and extrapolates with the widening ∇ and the narrowing Δ of Section 2.5. The analysis also needs a greatest value $\top$ for unknown values. A domain further supplies an emptiness test, a printer, and a concretization $\gamma : A \rightarrow \mathcal{P}(\mathbb{Z})$ with four laws. Such a type is a *numeric domain*:

```
1 (* src/Abstract_Interpreter/Domain/Lattice/Abstract_Domain.thy *)
2 class numeric_domain = executable_domain +
3   fixes gamma :: "'a::bounded_semilattice_sup_bot, order_top" => int set ("γ")
4   assumes gamma_bot[simp]: "γ ⊥ = {}"
5   assumes gamma_top[simp]: "γ ⊤ = UNIV"
6   assumes gamma_mono: "a ≤ b ⟹ γ a ⊆ γ b"
7   assumes is_empty_correct: "is_empty a ⟷ γ a = {}"
```

The third and fourth laws, `gamma_mono` and `is_empty_correct`, are the two requirements of the chapter opening. The class extends `executable_domain`, the part the analyzer runs, by the concretization and its laws.

Intervals (Section 2.3) are this chapter's running example [thy](#). As in Nipkow and Klein [6, §13.8.1], bounds are extended integers `int`, and an interval is a pair of bounds with $\gamma([l, u]) = \{n \in \mathbb{Z} \mid l \leq n \wedge n \leq u\}$. On nonempty pairs, the order and join (`less_eq_ivl`, `sup_ivl`) are those of Section 2.3. Nipkow and Klein identify all empty pairs by a quotient type. Voblint's `ivl` is a plain datatype of bound pairs, so it must handle them itself. Every pair with $l > u$ is empty, and so are $[+\infty, +\infty]$ and $[-\infty, -\infty]$, since no integer equals an infinite bound. The bottom value $\perp = [+\infty, -\infty]$ is only one of these empty pairs. An empty pair such as `[5, 3]` denotes the same set as $\perp$, yet it is neither $\perp$ nor below it.

The arithmetic operations build their results through `normalize_ivl`, which replaces every empty pair by $\perp = [+\infty, -\infty]$ and leaves nonempty pairs unchanged. Otherwise, `[1, 0] + [1, 1] = [2, 1]` would keep creating new empty values, and the solver, which tests whether a value changed (Section 2.8), could fail to stabilize. Normalizing never changes the denoted set (`normalize_ivl_gamma`), so it is sound wherever it is applied.

The meet intersects the bounds and does not normalize, since $\perp$ would not be the greatest lower bound of, say, `[0, 3]` and `[5, 9]` in the order on pairs (`meet_ivl_normalized_breaks_greatest`). An empty pair is therefore not necessarily $\perp$, and the emptiness test inspects the bounds (`is_bottom_ivl`): a pair is empty if $l > u$, if its lower bound is $+\infty$, or if its upper bound is $-\infty$. The widening is the standard one of Section 2.5, except that widening with $\perp$ returns the other operand (`widen_ivl_core`), and the narrowing refines only infinite bounds (`narrow_ivl_td`).

5.2. Abstract states as sets of stores

A numeric domain describes the possible values of one variable. The obligations of Chapter 4, however, are stated over sets of stores. A store maps every variable name to an integer, `store = vname  $\rightarrow$   $\mathbb{Z}$` , so at a program point, the analysis must describe whole stores. This thesis uses the two forms introduced in Section 2.4, non-relational and relational states. A non-relational description gives each variable its own abstract value, and a relational one records how variables relate to each other. The two differ in what they can state. Take the

stores with $0 \leq x \leq y \leq 5$ (Figure 5.2). The best non-relational description over intervals is $\{x \mapsto [0, 5], y \mapsto [0, 5]\}$. It shows $x \geq 0$ and $y \leq 5$, but not $x \leq y$, since it also admits $x = 5, y = 0$. A relational description that records $x \leq y$ shows exactly that fact, but it bounds neither variable, so it also admits $x = y = -7$. Therefore, neither description is stronger than the other. Together, however, they describe exactly the stores with $0 \leq x \leq y \leq 5$.

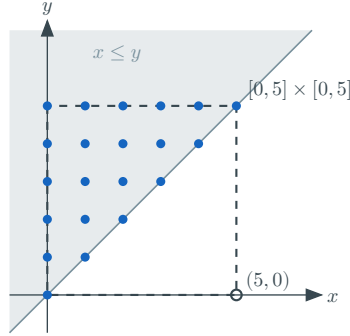


Figure 5.2: Stores over two variables as points (x, y) . The stores with $0 \leq x \leq y \leq 5$ are the dots. The best non-relational state denotes the square (dashed), which also admits $(5, 0)$. The relational state $\{x \leq y\}$ denotes the half-plane above the diagonal (shaded), which is unbounded. Their intersection is exactly the dots. Illustrative.

An *abstract state* describes a set of stores. It is a type D with the order and join the solver needs, and a monotone concretization that maps each state d to the set $\gamma(d)$ of stores it describes. Each kind of state defines its own concretization, and Isabelle overloads one constant, `gamma_S`, written $\gamma(d)$, for all of them. The numeric analyses use non-relational states. An analysis that records relations $x \leq y$ between variables uses a relational state, whose concretization maps a whole state to a set of stores instead of each variable to a set of integers. The interface of this chapter admits such a state, because the solver and the framework ask only for order, join, and a monotone concretization. Neither requires the state to map variables to abstract values. The relational state of the order analysis instantiates the solver's classes and `executable_domain` without being a numeric domain. The remaining sections of this chapter develop the non-relational case, and Section 11.3 presents the relational one.

5.3. Pointwise states and the unreachable state thy

A non-relational state is represented pointwise. It replaces the integer of a store by an abstract value: `abs_state = vname ⇒ 'a`, so a state d maps every variable name x to $d(x)$. Its concretization admits a store if every variable's value lies in the concretization of its abstract value:

```
1 (* src/Abstract_Interpreter/Domain/State/Nonrelational_State.thy *)
2 definition gamma_state :: "('a::numeric_domain) abs_state ⇒ store set" where
3   "gamma_state d = {s. ∀x. s x ∈ γ (d x)}"
```

Isabelle

Order and join work variable by variable, as for any function type in HOL. A state d_1 lies below d_2 if $d_1(x) \sqsubseteq d_2(x)$ for every variable x , and the join is $(d_1 \sqcup d_2)(x) = d_1(x) \sqcup d_2(x)$ (Figure 5.3). Because γ is monotone, so is this concretization, and a larger state describes more stores.

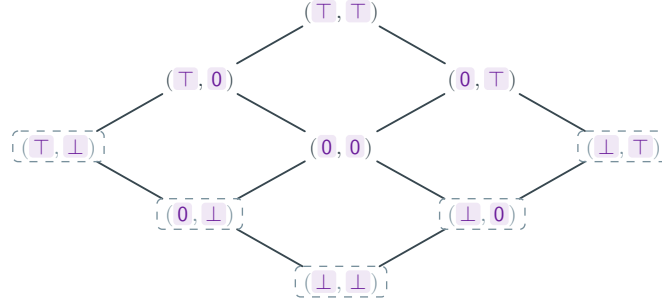


Figure 5.3: Pointwise states $(d(x), d(y))$ over two variables, with values from the fragment $\perp \sqsubseteq 0 \sqsubseteq T$ of Sign, ordered variable by variable. The dashed states have a $\perp$ component and denote no store, although only the lowest one is the bottom state.

Some program points are reached by no execution, for example, code behind a condition that never holds. The analysis should recognize them from their state. A pointwise state makes this hard, because it can say “unreachable” in many ways. Every state with one empty variable describes no store, and only one of them is the bottom state (Figure 5.3).

Neither simple test works. Treating only the all-bottom state as unreachable misses most cases, because refining a branch condition (Section 5.6) usually empties a single variable, as in $\{x \mapsto \perp, y \mapsto T\}$. Treating every empty state as unreachable is correct at the moment of the test, but the information does not survive the next step. Joining two differently empty states can give a nonempty one (Figure 5.4). This is sound, since it only adds stores, but it loses the information that the point is unreachable and is thus imprecise.

```

1 fun main() {
2   x = __voblint_nondet_int();
3   y = __voblint_nondet_int();
4   if ((x == 0 && x == 1) || (y == 0 && y == 1)) {
5     __voblint_check(false);
6   }
7 }

```

Figure 5.4: A branch no execution enters, since no integer equals both 0 and 1. Sign refines the left disjunct to $\{x \mapsto \perp, y \mapsto T\}$ and the right one to $\{x \mapsto T, y \mapsto \perp\}$. Both are empty, but their pointwise join is $\{x \mapsto T, y \mapsto T\}$. The branch turns each empty arm into **Bot** before the join, and the analyzer reports the check in the branch as unreachable (claim [dom-disjunct-sign](#)).

Voblint therefore makes unreachability a value of its own, as Nipkow and Klein do with an option type [6, §13.7] and as Goblint does. Goblint’s framework lifts the local state by an outer bottom element for dead code ([Analyses.Dom](#)), and a transfer function that finds its path unreachable raises the **Deadcode** exception instead of returning a state. In Voblint, a lifted state (**lifted**) is either **Bot**, meaning unreachable, or **Lifted**(d) for an ordinary state d , with **Bot** below every other value and $\gamma(\text{Bot}) = \emptyset$.

After every transfer, the analysis tests whether the result is empty and, if so, replaces it with **Bot** ([normalize_lift](#)). Comparing with the bottom state cannot implement this test, for the reason above. Instead, the test relies on the domain’s emptiness test, which [is_empty_correct](#) makes exact for every value. A pointwise state is empty exactly when some variable’s value is ([is_empty_state](#), [is_empty_state_iff_gamma_state_empty](#)). The variable names form an infinite set, so the executable analyzer decides this on

a finite representation of the state (Section 12.1). Replacing an empty state by **Bot** only exchanges one abstract representation of the empty set for another, equivalent one (`gamma_state_normalize_lift`), so it cannot make the analysis unsound.

From then on, **Bot** stays **Bot**. A transfer applied to **Bot** is skipped and returns **Bot**, and a join with **Bot** returns the other operand. An unreachable program point thus keeps the value **Bot**, and the analysis can report it as unreachable.

5.4. Forward evaluation and truth tests thy

The carrier operations serve the solver. The analysis additionally needs operations that mirror the program's own arithmetic, comparisons, and branch conditions. They are called the domain's *primitives*. This section and the next two introduce them, one use at a time.

An assignment $x = e$ needs the abstract value of e . Forward evaluation computes it from an abstract state d , as in the generic abstract interpreter of Nipkow and Klein [6, §13.5.3]. The abstract result $\llbracket e \rrbracket^\# d$ must contain every concrete result (`sound_evaluator`):

$$s \in \gamma(d) \implies \llbracket e \rrbracket_e s \in \gamma(\llbracket e \rrbracket^\# d).$$

Interval evaluates by interval arithmetic on the bounds (`aval_ivl`). For example, if $d(x) = [0, 9]$, Interval evaluates $x + 1$ to $[1, 10]$.

A condition is an expression too, and VIMP reads its value as a truth value, with nonzero meaning true. If the abstract value of a condition admits only one truth value, a branch can drop the arm that needs the other. A domain therefore also supplies a truth test (`sound_truth_test`). The test `tobool(a)` has three possible answers. It returns `Some(b)` if every integer in $\gamma(a)$ has truth value b , and `None` otherwise:

$$\text{tobool}(a) = \text{Some}(b) \wedge i \in \gamma(a) \implies (i \neq 0) = b.$$

Interval's test `interval_tobool` returns `Some(true)` for $[l, u]$ if the interval excludes 0, that is, $u < 0$ or $l > 0$, and `Some(false)` if $l = u = 0$. Otherwise, it returns `None`. So $[1, 5]$ is true, $[0, 0]$ is false, and $[0, 5]$ is undecided.

5.5. Answering checks with three-valued queries

A check `__voblnt_check(c)` asks whether c holds in every store that reaches its program point. The analysis must answer this from the abstract state there alone. Voblnt answers it through queries, as does Goblnt's `assert` analysis.

A domain answers two comparisons on abstract values, `less(a, b)` for $a < b$ and `eq(a, b)` for $a = b$. Like the truth test, each has three possible answers. A definite answer must hold for every pair of integers the two values denote, and `None`, meaning unknown, is always allowed:

```
1 (* src/Abstract_Interpreter/Domain/Eval/Numeric_Queries.thy *)
2 locale sound_numeric_queries =
3   fixes less :: "'a::numeric_domain => 'a => bool option"
4   and eq :: "'a => 'a => bool option"
5   assumes less_sound[intro]:
6     "less a b = Some r => i ∈ γ a => j ∈ γ b => (i < j) = r"
7   and eq_sound[intro]:
8     "eq a b = Some r => i ∈ γ a => j ∈ γ b => (i = j) = r"
```

Domains differ in which comparisons they can decide. Interval decides $a < b$ when one interval lies entirely below the other, where a shared endpoint suffices for the answer false,

and $a = b$ when the intervals are disjoint or the same single integer. Sign decides $a < b$ whenever the signs separate the values, for example, for zero and a positive value. Parity and Congruence, the domains of Chapter 11, are weaker here. Parity can only refute equality between an even and an odd value and never decides $<$. Congruence answers only when both values are single integers.

Definition 5.1 (Query, fun `check_query`). A query evaluates a condition c on an abstract state d to one of three answers, *true*, *false*, or *unknown*, by recursion on c , using only the domain's two comparisons.

In `check_query`, a comparison first evaluates its operands forward to abstract values $\hat{a}$ and $\hat{b}$ (Section 5.4) and then asks one of the two comparisons, with the operands swapped or the answer negated where needed:

$$\begin{aligned} a < b &\rightsquigarrow \text{less}(\hat{a}, \hat{b}), & a > b &\rightsquigarrow \text{less}(\hat{b}, \hat{a}), & a == b &\rightsquigarrow \text{eq}(\hat{a}, \hat{b}), \\ a \leq b &\rightsquigarrow \neg \text{less}(\hat{b}, \hat{a}), & a \geq b &\rightsquigarrow \neg \text{less}(\hat{a}, \hat{b}), & a \neq b &\rightsquigarrow \neg \text{eq}(\hat{a}, \hat{b}). \end{aligned}$$

Negation keeps *unknown*. The connectives `!`, `&&`, and `||` combine the answers of their operands in three-valued logic. A definite *false* decides `&&`, and a definite *true* decides `||`, regardless of the other operand's answer. Both operands are evaluated in the same state, so for x in $[1, 2]$, $x == 1 \ || \ x == 2$ stays unknown, although every store satisfies it (claim `dom-disjunction-one-state`). Goblint is more precise here. CIL compiles `&&` and `||` into branches (`cabs2cil.ml`), and where $x == 1$ fails, Goblint's interval refinement narrows $[1, 2]$ to $[2, 2]$ (`baseInvariant.ml`, [27, §4.5.1, Rem. 8]). Voblint's interval inverse of a failed equality changes nothing (`inv_eq_ivl`). Any other expression e is read as $e \neq 0$, its truth value in VIMP. A new domain therefore gets checks on arbitrary conditions by providing these two comparisons. One proof covers every domain. The answer *true* certifies that c holds in every store the state describes, *false* that it holds in none, and *unknown* makes no claim (`check_query_sound`). `sound_check_query` combines the queries with forward evaluation. An analysis passes these answers to the analyses it runs with and to the verdicts as answers to the query `EvalInt` of Section 7.1 (Section 7.3, Section 10.2).

For example, the condition $x \leq y \ \&\& \ y \neq 0$ is answered from $\text{less}(\hat{y}, \hat{x})$ and $\text{eq}(\hat{y}, \hat{0})$ alone. In Sign, with x non-positive and y positive, both comparisons give a definite *false*, both negations give *true*, and so does the conjunction (Figure 5.5).

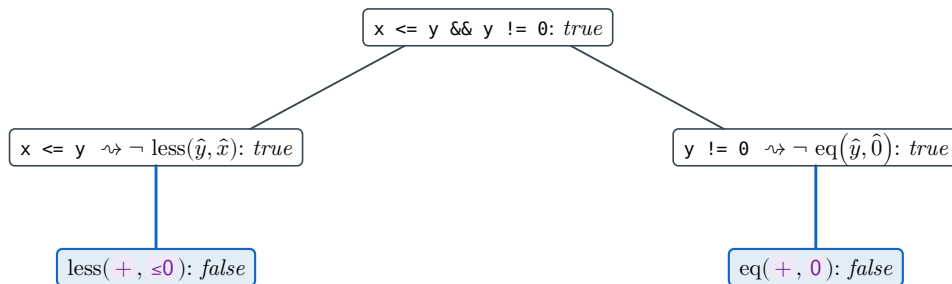


Figure 5.5: How `check_query` answers a compound condition, in Sign with $x \leq 0$, $y = +$ (claim `dom-query-tree-sign`). Each comparison becomes one of the domain's two comparisons (blue), as $\rightsquigarrow$ marks; the answer follows the colon. Negation flips a definite answer, and `&&` is true because both operands are, so the check is PROVED.

5.6. Refining states at branch guards [thy](#)

When the analysis enters a branch, it knows whether the condition held. On the true arm of `if (0 < x)`, for example, x is positive. Ignoring this knowledge is sound, since the state before the branch already contains every store that takes either arm. It costs precision, however, as the following program shows.

```
1 x = __voblint_nondet_int();
2 if (0 < x) {
3   y = x;
4 } else {
5   y = - x;
6 }
7 __voblint_check(y >= 0);
```

In this program, y ends up as $|x|$, so the check always holds. Without guard filtering, both arms keep $x \mapsto \top$, y becomes $\top$, and the state cannot show that the check holds. The guard filter runs the condition in reverse, as in the backward abstract interpretation of tests by Cousot [50, §§8.5, 10.1] and the backward analysis of Nipkow and Klein [6, §13.7]. Given abstract operands a_1, a_2 , and the result an operation must produce, an inverse operator returns refined operands a'_1, a'_2 that keep every concrete pair producing that result. For a comparison that must yield r :

$$n_1 \in \gamma(a_1) \wedge n_2 \in \gamma(a_2) \wedge (n_1 < n_2) = r \implies n_1 \in \gamma(a'_1) \wedge n_2 \in \gamma(a'_2),$$

and likewise for equality, addition, subtraction, and multiplication ([sound_inverse_ops](#)).

The refined operand is then combined with the value known before. Nipkow and Klein use the lattice meet for this and require it to be exact on the denoted sets. Voblint asks for less, namely an intersection that keeps every integer both operands share and lies below both ([sound_intersection](#), shown in Appendix A). For Interval, it is the normalized meet [intersect_ivl](#), which agrees with the meet on the operands below. In the program above, Interval refines x to $\top \sqcap [1, +\infty] = [1, +\infty]$ on the true arm and to $[-\infty, 0]$ on the false arm. Both arms then give y a non-negative value, and the join yields $y = [0, +\infty]$.

The guard filter [bfilter](#) takes a condition, the required truth value, and a state, and applies the inverse operators along the structure of the condition. For the arithmetic operands of a comparison, it calls [afilter](#), which refines a state so that an expression evaluates within a required value [6, §13.7.1]. Its soundness is exactly what the branch needs, since it never removes a store that satisfies the guard. The filter refines a conjunction from right to left. Each conjunct is applied once, to the state the conjuncts on its right have refined, and the filter does not iterate to a local fixpoint. The bounds that reach each variable can therefore depend on the order of the conjuncts. With Interval, $0 < i \ \&\& \ i < j \ \&\& \ j < 10$ bounds i to $[1, 8]$ but not j from below, and the reversed order bounds j to $[2, 9]$ but not i from above (claim [dom-conjunct-order](#)).

Lemma 5.2 (Sound guard filter, [bfilter_sound](#)). *If a store s lies in $\gamma(d)$ and the guard e has truth value res at s , then s lies in the concretization of the filtered state.*

```
1 (* src/Abstract_Interpreter/Domain/Eval/Backward_Domain.thy *)
2 lemma bfilter_sound [intro]:
3   assumes "s ∈ γ d" "truthy ([e]_e s) = res"
4   shows "s ∈ γ (bfilter e res d)"
```

The shipped inverse operators are sound but not the most precise possible. Interval and Sign, for instance, do not invert arithmetic and cannot learn $x < 4$ from $x + 1 < 5$, as the interval analysis of Nipkow and Klein does [6, §13.8.3].

5.7. The domain interface and its derived transfers [thy](#)

A domain provides two groups of operations, which Figure 5.6 shows with their laws. The *carrier operations* serve the solver: the order, the join, bottom and top, widening and narrowing, the emptiness test, and the printer, tied to the concretization by the laws of Section 5.1. The *primitives* mirror the program’s own operations: evaluation and the truth test (Section 5.4), the two comparisons (Section 5.5), and the inverse operators and the intersection (Section 5.6). A non-relational domain hands its primitives over as one record of type `nonrelational_ops` and proves them sound once by interpreting `sound_nonrelational_ops`. For Interval, the record is `ivl_ops`.

A domain supplies nothing else. In particular, it writes no transfer functions, because the primitives determine every edge (`tf_abs`, `enter_for`):

assignment $x = e$	$d \mapsto d[x \mapsto \llbracket e \rrbracket^\sharp d]$
guard b , required value r	the guard filter of Section 5.6 for b and r
return e	$d \mapsto d[\text{ret_var} \mapsto \llbracket e \rrbracket^\sharp d]$
entry with formals $\bar{x}$, arguments $\bar{e}$	every local set to $\top$, globals kept, then $\bar{x} \mapsto \llbracket \bar{e} \rrbracket^\sharp d$

Skips, procedure bodies, and checks leave the state unchanged. With Interval and $d(x) = [0, 9]$, for instance, the edge $x = x + 1$ yields $d[x \mapsto [1, 10]]$. The interpretation of `sound_nonrelational_ops` proves all of these transfers sound at once, so the domains of Chapter 11 mainly define their values and primitives and prove the primitive laws. The analyzer runs the same primitives on a finite representation of states, and their soundness carries over to the generated code without further proof (`exec_local_spec_sound`, Section 12.1).

5.8. Carriers without least upper bounds

The interface asks more of a carrier than an order with a join. The verified TD solver requires a `bounded_semilattice_sup_bot` (`TD_side_upd_rule`), whose join is the *least* upper bound, and the framework’s proofs use this leastness as well (`sound_refinement.bfilter_reductive`). Not every domain in practical use meets this.

Goblint’s exclusion-set domain `DefExc` describes an integer either by a definite value, `Definite n`, or by a finite set of excluded values, `Excluded S`, which stands for every integer outside S . A smaller exclusion set is less precise, so `Excluded S` lies below `Excluded S'` when S' is a subset of S . Goblint also attaches a bit range to every value, which makes the domain finite for each integer type. Read over mathematical integers, without this range, the domain has no least upper bound of `Definite 1` and `Definite 2`. An upper bound must describe both integers, so it is some `Excluded S` with $1, 2 \notin S$. Adding a further excluded value $p \notin S \cup \{1, 2\}$ gives `Excluded (S ∪ {p})`, which is again an upper bound and lies strictly below. Since S is finite and the integers are not, every upper bound has a smaller one, and no least upper bound exists.

Voblint could admit such a carrier only by bounding its integers, as Goblint does, or by weakening the join to an arbitrary upper bound throughout the solver and the framework and reproofing the results that rely on leastness. Both are left for future work.

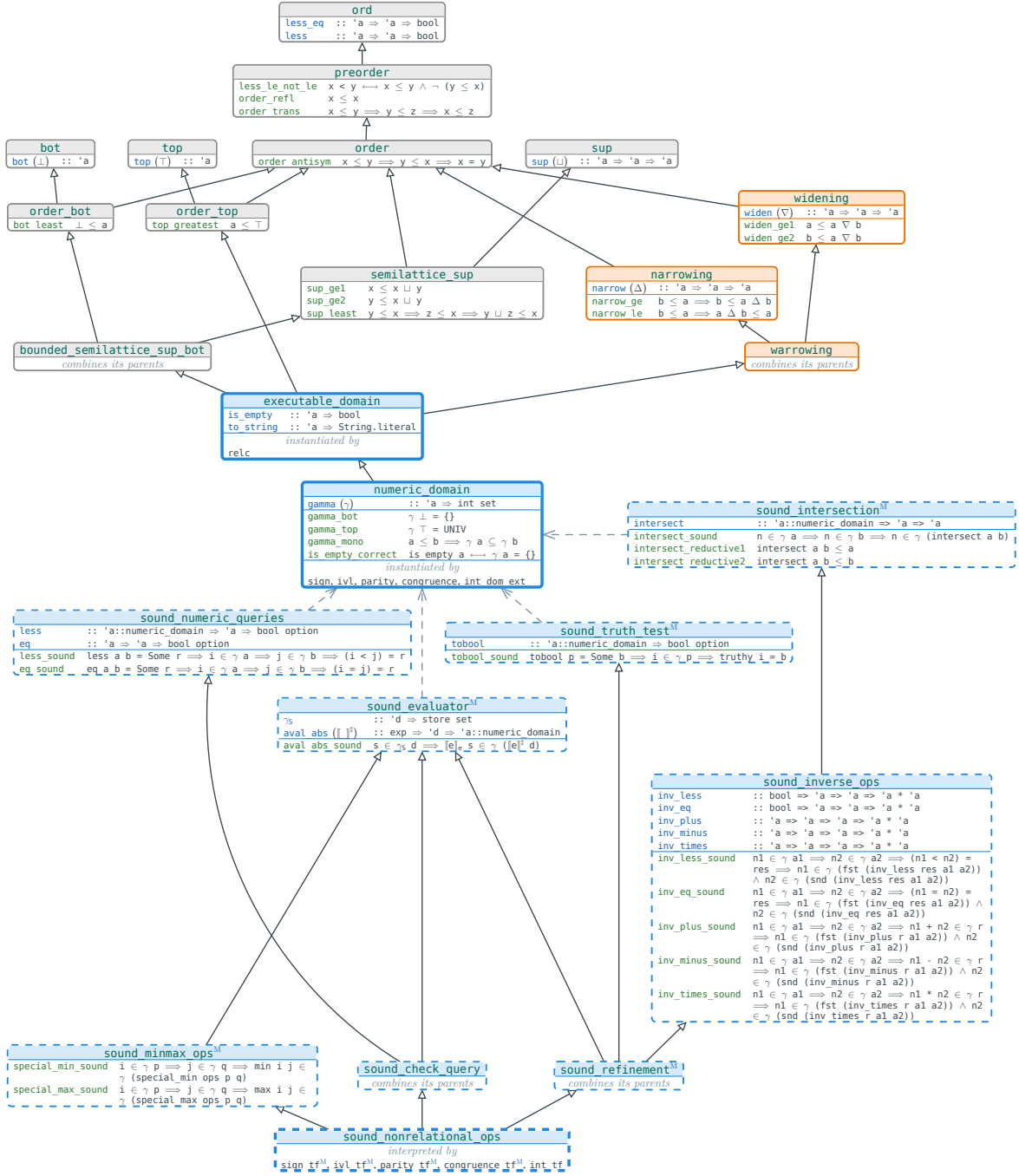


Figure 5.6: The operations and laws of a non-relational domain over its carrier type 'a', down to the soundness locale `sound_nonrelational_ops` an analysis interprets. Each class (solid) or locale (dashed) lists the operations and laws it declares, which the declarations below it inherit. Solid arrows point to what a declaration extends, dashed ones to the class its type variable is constrained to. An interface an analysis implements has a heavier border and lists its instances. A superscript ^M marks an interface with a monotone strengthening, and an interpretation that proves it (`mono_nonrelational_ops`). Color gives where a node is declared: HOL, the vendored TD solver, Voblint.

6. The Analysis Interface

Chapter 5 ended with laws about one abstract state. Its order agrees with its meaning, and its operations over-approximate the concrete ones. The obligations of Chapter 4, however, speak about edges, calls, and contexts, so something must say what an analysis does at each step of a program. The non-relational domains of Chapter 5 settle this through their primitives, which determine every edge (Section 5.7). Other analyses need more. A relational analysis does not store one value per variable (Section 11.3), an analysis may keep facts that belong to no program point, and analyses may need each other's facts to be precise (Chapter 7). Calls, moreover, need more than one transfer, because the caller's locals survive the call, while the callee's globals and result flow back.

Everything else between an analysis and the obligations is the same for every analysis, namely, the equations built from the control-flow graph, the calling contexts, and the solver. Goblint therefore separates an analysis from the framework that runs it [2, §6], and Voblint makes the same split in its program and in its proof. This chapter describes what an analysis must supply, and what it must prove about it, so that the framework can discharge the obligations of Chapter 4 once for every analysis. An analysis supplies a record of abstract operations, and the analysis contract states what they preserve.

6.1. Edge transfers and the enter/combine protocol

An ordinary edge of the control-flow graph carries an action a , such as an assignment or a guard, whose concrete meaning `edge_step` is fixed by Chapter 3. For a store s , `edge_step a s` is the set of stores the edge can produce. An analysis describes the stores at a program point by an abstract state d of its *local domain* D_L and supplies, for each action a , a transfer function $\llbracket a \rrbracket^\sharp$. The transfer is sound if it loses no successor:

$$s \in \gamma(d) \wedge s' \in \text{edge_step } a \ s \implies s' \in \gamma(\llbracket a \rrbracket^\sharp d).$$

This is the abstract counterpart of the obligation `INTRA` of Chapter 4 for one edge.

A call cannot be handled by one such transfer. After it returns, the caller's own locals are the ones from before the call, while the globals and the result come from the callee (`combine_collect`, `RETURN`). An analysis therefore needs two states at a call, one for the callee to start from and one that saves what the caller needs when execution returns. As in Goblint, it supplies two operations. *Enter* (`dgs_enter`) produces both states, and *combine* (`dgs_combine_env`, `dgs_combine_assign`) builds the caller's state after the call from the resume state and the callee's exit state. Take one call:

```
1 global g;  
2 fun inc(a) { g = g + a; return a - 1; }  
3 fun main() { g = 1; x = 5; y = inc(x); }
```

VIMP ↗

At the call `y = inc(x)`, the *entry state* e describes the store the callee starts with ($g = 1$, $a = 5$). The *resume state* q describes the caller's store ($x = 5$, $g = 1$). For soundness, enter

must return such a pair, q covering the caller's store and e covering the entered store, a property called *entry coverage* (`entry_pairs_cover`). When the callee exits with state r , combine merges q and r (Figure 6.1).

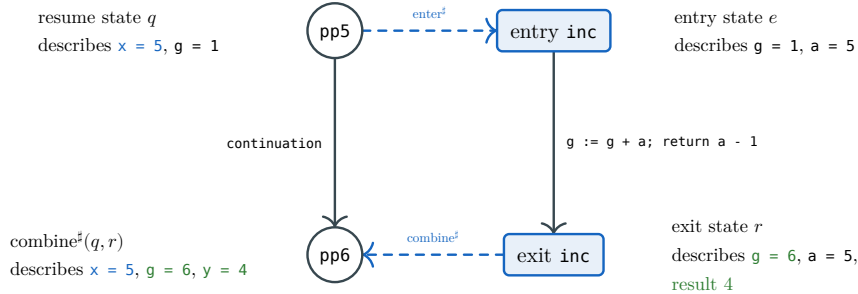


Figure 6.1: Enter and combine at the call $y = \text{inc}(x)$, with the concrete store each abstract state must describe (schematic, with the body of `inc` drawn as one edge). The combined state must describe the caller's locals (blue) together with the callee's globals and result (green), as `combine_collect` does.

Like Goblint's `enter`, `enter` may return several pairs, one per path in Goblint's [path-sensitivity lifter](#) [51, §5.7]. The analyses shipped with Voblint return one, so path-sensitive analyses remain future work.

6.2. Local soundness laws of an analysis

The operations of Section 6.1 say how an analysis propagates abstract states. They do not by themselves guarantee that these states cover the concrete executions, and the framework cannot check this without knowing what the analysis tracks. It therefore asks every analysis for local guarantees, which Chapter 8 composes into the five obligations of Chapter 4 for every program and context policy. Edge transfers account for `INTRA` and combine for `RETURN`. The obligations `CALL` and `TOTAL` depend on the context policy, so the framework derives them from the entry coverage of Section 6.1 together with the policy (Section 8.2), and `INIT` requires the initial state d_0 to cover the initial stores, which the analysis discharges when it is registered (`dg_analysis.init_sound`).

The framework takes the operations of an analysis as one parameter, the record `dg_spec` with one field per operation: the edge transfers, `enter`, the two combine stages, and a query handler that Chapter 7 uses to let analyses exchange facts.

```
1 (* src/Abstract_Interpreter/Framework/Spec/DG_Spec.thy *)
2 record ('x,'k,'v','dl','dg) dg_spec =
3   dgs_skip      :: "('x,'k,'v','dl','dg) man_transfer" ("skip")
4   dgs_assign    :: "vname ⇒ exp ⇒ ('x,'k,'v','dl','dg) man_transfer" ("assign")
5   dgs_special   :: "special_call ⇒ vname ⇒ ('x,'k,'v','dl','dg) man_transfer" ("special")
6   dgs_branch    :: "exp ⇒ bool ⇒ ('x,'k,'v','dl','dg) man_transfer" ("branch")
7   dgs_body      :: "pname ⇒ ('x,'k,'v','dl','dg) man_transfer" ("body")
8   dgs_return    :: "exp option ⇒ pname ⇒ ('x,'k,'v','dl','dg) man_transfer" ("return")
9   dgs_enter     :: "call_info ⇒ ('x,'k,'v','dl','dg) man_enter_transfer" ("enter")
10  dgs_event      :: "analysis_event ⇒ ('x,'k,'v','dl','dg) man_transfer" ("event")
11  dgs_combine_env :: "call_info ⇒ ('x,'k,'v','dl','dg) man_combine_transfer" ("combine' env")
12  dgs_combine_assign :: "call_info ⇒ ('x,'k,'v','dl','dg) man_combine_transfer" ("combine' assign")
13  dgs_query      :: "('x,'k,'v','dl','dg) man_query"
```

Every field except the query handler carries, marked $\sharp$, the name of the corresponding method of Goblint's `Spec`. A `call_info` names the callee, its formals, the arguments, and the destination of a call.

The record holds the operations, and the *analysis contract* `analysis_contract` states what their results must satisfy. For an analysis whose transfers depend only on the local state, it consists of three laws on the concretization γ :

$$\begin{array}{ll} \text{monotonicity} & d \sqsubseteq d' \implies \gamma(d) \subseteq \gamma(d') \\ \text{edges} & s \in \gamma(d), s' \in \text{edge_step } a \ s \implies s' \in \gamma(\llbracket a \rrbracket^\sharp d) \\ \text{combine} & s \in \gamma(q), t \in \gamma(r) \implies \text{combine_collect } (s, t) \in \gamma(\text{combine}(q, r)) \end{array}$$

Monotonicity turns the solver's inequalities into inclusions between sets of stores, as for a domain (Chapter 5). The edge law is the condition of Section 6.1 and covers `INTRA`. The combine law covers `RETURN`. If q describes the caller's store s and the exit state r describes the callee's final store t , the combined state describes the store in which the caller continues. Transfers that also use facts without a program point need a more general form of these laws, which Section 6.3.1 states.

6.3. Facts shared across program points

Not every fact belongs to one program point. An analysis may, for example, keep one range for a global variable g that holds throughout the execution.

6.3.1. Reading and publishing analysis globals [thv](#)

Seidl et al. describe a Goblint analysis with two domains [2, §6]. The local domain D_L holds facts per program point and context, and the *global domain* D_G holds the values of *analysis globals*, which are unknowns without a program point. An assignment to g publishes its value to the analysis global of g as a side effect, and an edge that reads g depends on that unknown. Goblint's `Spec` accordingly declares a local lattice $\mathbf{D}$ and a global lattice $\mathbf{G}$.

The word *global* thus has two meanings. A *program global* is a variable of the VIMP program. An analysis global is a fact the analysis keeps flow-insensitively. The two are independent, and by default, the numeric analyses keep program globals in the local state and use no analysis global (Section 6.3.2).

A transfer that uses analysis globals still returns the next local state, but it may read and publish them on the way, in Goblint through a *manager* argument whose contributions the framework passes to the solver [2, §6]. Voblint adopts this design. Its transfer (`man_transfer`) receives a manager of type `man` instead of a local state. It reads an analysis global v with `man_global` v and publishes d to it with `man_sideg` v d . As a result, the equations of Chapter 8 depend on an analysis global only where a transfer actually reads it. A transfer that is a pure function of the local state reads only the state at the edge's source and publishes nothing (`sp_compile_transfer_program_local_transfer`).

With analysis globals, a local state alone no longer fixes which stores it describes, because its meaning may depend on their values. The analysis contract therefore uses a concretization $\gamma_{DG}(d, \eta)$ of a local state d under an *environment* η , which maps every analysis global to its value. Each law of Section 6.2 takes η as an extra parameter (`analysis_contract.gammaDG_mono`, `analysis_contract.step_sound`, `analysis_contract.combine_sound`). Monotonicity holds in both arguments, and the edge

and combine laws describe their results under η joined with what the operation publishes. A technical premise (`dg_spec_wf`) requires that the program of every operation run its continuation exactly once, so that what it publishes is well defined. The locale states these laws over the solver's representation of a valuation:

```

1 (* src/Abstract_Interpreter/Framework/Spec/DG_Spec_Sound.thy *)
2 locale analysis_contract =
3   fixes S :: "('x,'k,'v','D'::bounded_semilattice_sup_bot,
4     'G'::bounded_semilattice_sup_bot) dg_spec"
5   and  $\gamma_{DG}$  :: "'D  $\Rightarrow$  ('v  $\Rightarrow$  'G)  $\Rightarrow$  store set"
6   and  $\mathcal{G}$  :: "vname  $\Rightarrow$  bool"
7   assumes spec_wf: "dg_spec_wf S"
8   and gammaDG_mono:
9     "[d  $\leq$  d'; e  $\leq$  e']  $\Rightarrow$   $\gamma_{DG}$  d e  $\subseteq$   $\gamma_{DG}$  d' e'"
10  and step_sound:
11    "edge_collect a ( $\gamma_{DG}$  (dg_local ( $\tau$  src)) (genv key  $\tau$ ))
12       $\subseteq$   $\gamma_{DG}$  (edge_out S a src key  $\tau$ ) (genv key  $\tau$   $\sqcup$  edge_pub S a src key  $\tau$ )"
13  and combine_sound:
14    "[s  $\in$   $\gamma_{DG}$  q (genv key  $\tau$ ); t  $\in$   $\gamma_{DG}$  r (genv key  $\tau$ )]  $\Rightarrow$ 
15      combine_collect  $\mathcal{G}$  (ci_dst ci) s t
16       $\in$   $\gamma_{DG}$  (combine_out S ci key q r  $\tau$ ) (genv key  $\tau$   $\sqcup$  combine_pub S ci key q r  $\tau$ )"

```

Here, σ (τ in the locale) is an arbitrary valuation of the solver's unknowns (Section 9.1.3), d is its local half `dg_local` at the edge's source, and η is `genv key  $\tau$` , its values at the analysis globals. `edge_out` and `edge_pub` are the local state a transfer returns and what it publishes, and `combine_out` and `combine_pub` are the same for combine. An analysis without analysis globals ignores η and meets the three laws of Section 6.2.

6.3.2. Where program globals are stored thy

The contract leaves open where program globals are kept. By default, program globals are part of the local state, as in Goblint's default base analysis for single-threaded programs (`base.ml`, [27, §5.6.1]). The other placement keeps each program global at an analysis global of its own, published by side effects and read where it is used. The analysis is then *mixed flow-sensitive*, flow-sensitive in the locals and flow-insensitive in the globals.

```

1 global Gx;
2 fun set() { Gx = 1; }
3 fun get() { return Gx; }
4 fun main() { x = 1; set(); y = get(); }

```

The flow-insensitive placement keeps one value of `Gx` for every program point, including those before `set` runs, so it must cover the initial `Gx = 0` as well as the 1 written by `set`. Sign therefore claims only ≥ 0 for `y` after `y = get()` (claim `mixed-flow-sign-shared`), although every execution ends with `y = 1`. Flow-sensitively, Sign carries $+$ from `set`'s exit into `get`'s entry and derives $+$ (claim `mixed-flow-sign`). In sequential VIMP, the flow-insensitive placement can thus cost precision.

The flow-insensitive placement matters beyond this example. Another thread may write a shared global at any point, so the simplest thread-modular analysis keeps shared globals flow-insensitively [2, §1, Ex. 4], [27, §5.3.1]. More generally, mixed flow-sensitivity lets an analysis choose per fact whether precision at each program point is worth a value per point and context. The placement is proved to meet the analysis contract for the analyzer's product state (`keyed_split_contract`), so the end-to-end soundness theorem covers both placements, and a future extension with threads can build on a verified flow-insensitive core.

7. Cooperating Analyses

An analysis of Chapter 6 sees only its own state. For some properties a user checks, however, a single analysis is not enough. They can only be proved by combining what several analyses know while they run together. Consider the example below:

```

1 fun main() {
2   x = __voblnt_nondet_int();
3   y = __voblnt_nondet_int();
4   if (x <= y) {
5     if (y <= x) {
6       z = (x == y);
7       __voblnt_check(z == 1);
8     }
9   }
10 }

```

The check holds in every run, since both guards hold at the assignment, so $x = y$ and z is 1. No single analysis, however, can prove it. Interval keeps one range per variable. After the guards, both x and y are still unbounded, so $x == y$ evaluates to $[0, 1]$ and the check stays **UNKNOWN** with $z=[0,1]$. The order analysis (Section 11.4) is relational. Its state is a set of pairs (x, y) , each meaning that the value of x is at most the value of y . After the guards, it holds (x, y) and (y, x) , so it knows that $x = y$. It tracks no values of variables, however, so it learns nothing about z and reports **UNKNOWN** too. Together, the two can decide it, if Interval's assignment can learn what the order analysis knows at that point (Figure 7.1).

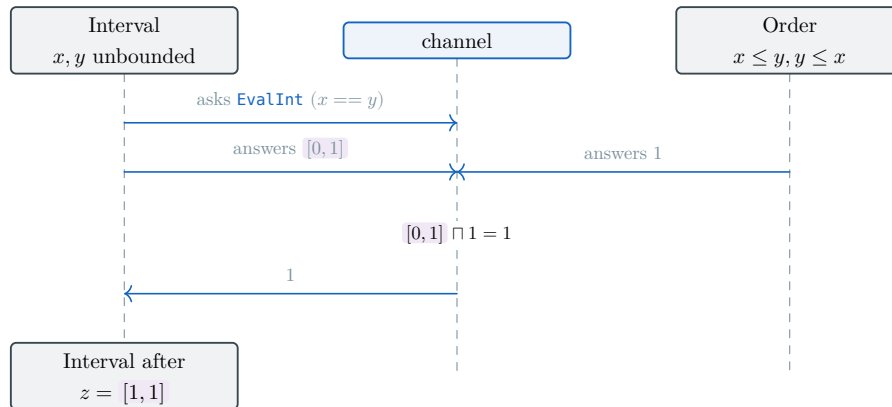


Figure 7.1: One step of Interval and the order analysis at $z = (x == y)$ in the program above (schematic). Interval's assignment asks **EvalInt** $(x == y)$. The channel asks the handler of every analysis and gives Interval the meet of their answers. Each analysis updates only its own field of the product state. The final states match the analyzer run **PROVED** with `interval: z=[1,1]; order: {x≤y ∧ y≤x}`.

Goblint solves this with its *master control program* (MCP). The selected analyses run side by side, each on its own part of one product state, and a transfer may ask all selected analyses a question through the manager's `ask` function. Goblint has no machine-checked argument that these exchanges are sound. A proof, however, has a modularity problem. If Interval's assignment is proved sound using the order analysis's invariant, the two proofs are coupled, every new pair of analyses needs a new proof, and adding an analysis reopens old ones. This chapter describes how each analysis instead gets one proof obligation that does not mention the analyses it runs with, and why analyses meeting it combine into one analysis that meets the analysis contract of Section 6.2.

7.1. Queries, answers, and channels thy

The analyses must agree on what a question means and on what an answer claims. A *query* asks for a fact about the stores a state describes. Voblint has one kind, named after Goblint's. The query `EvalInt e` asks which integers the expression e may evaluate to. Its answer is a value of the reduced product `Int` (`int_dom`, Section 11.2), lifted by a bottom and by a top element that claims nothing (`query_lift`). A comparison evaluates to 0 or 1, so the exact answer 1 to `EvalInt (x == y)` says that $x = y$.

Definition 7.1 (Truth of an answer, fun `eval_holds`). An answer a to `EvalInt e` *holds* at a store s if $\llbracket e \rrbracket_e s \in \gamma(a)$:

```
1 (* src/Abstract_Interpreter/Framework/Cooperation/Analysis_Query.thy *)
2 fun eval_holds :: "query  $\Rightarrow$  answer  $\Rightarrow$  store  $\Rightarrow$  bool" where
3   "eval_holds (EvalInt e) a s  $\longleftrightarrow$   $\llbracket e \rrbracket_e s \in \text{gamma\_answer } a$ "
```

Isabelle

Two laws, stated by the locale `query_algebra`, let answers from several analyses work together:

```
1 (* src/Abstract_Interpreter/Framework/Cooperation/Analysis_Query.thy *)
2 locale query_algebra =
3   fixes answer_holds :: "'q  $\Rightarrow$  'r::{semilattice_inf, order_top}  $\Rightarrow$  store  $\Rightarrow$  bool"
4   assumes top_sound: "answer_holds q  $\top$  s"
5   and inf_sound: " $\llbracket \text{answer\_holds } q \ a \ s; \text{ answer\_holds } q \ b \ s \rrbracket \implies \text{answer\_holds } q \ (a \sqcap b) \ s$ "
```

Isabelle

The top answer $\top$ holds at every store, so an analysis that cannot answer a query answers $\top$ and claims nothing. If two answers hold at a store, so does their meet, so a query asked of all analyses can be answered by the meet of their answers. In the example, when asked `EvalInt (x == y)` after both guards, Interval answers $[0, 1]$, which holds at every store, and the order analysis answers 1. Their meet is $[0, 1] \sqcap 1 = 1$, so $x = y$. Goblint's `MCP.query` combines answers the same way, as formalized by Apinis [51, §5.5].

A transfer may need answers to several queries. Instead of answering each query separately, the framework passes the transfer one function that answers every query, a *channel* ch of type `channel`, and the transfer applies it to the queries it needs. A single premise about the channel then covers all answers the transfer uses. A channel *holds* at a store s if every answer it gives holds at s :

```
1 (* src/Abstract_Interpreter/Framework/Cooperation/Analysis_Query.thy *)
2 definition channel_holds :: "('q  $\Rightarrow$  'r)  $\Rightarrow$  store  $\Rightarrow$  bool" where
3   "channel_holds ch s  $\longleftrightarrow$  ( $\forall q. \text{answer\_holds } q \ (ch \ q) \ s$ )"
```

Isabelle

By the two laws, the channel answering $\top$ everywhere holds at every store, and the point-wise meet of two channels that hold at s holds at s (`query_algebra.channel_holds_inf`).

7.2. Proving each analysis against true answers thy

Interval’s assignment in Figure 7.1 uses the wrapper `ask_assign`, which any analysis can put around its assignment $x := e$. The wrapper asks `EvalInt` e . When the answer is one integer n , it runs the analysis’s own assignment of the literal n , and otherwise, the assignment of e . At $z := (x == y)$, Interval receives 1 and sets z to `[1,1]`, which is right only if the answer is true. Proving that from the order analysis’s invariant would couple the two proofs.

Voblint avoids this coupling by separating two facts, that a transfer uses its answers correctly and that the answers are true. The transfer `ls_step c ch a x` of an analysis c receives the channel ch as an extra argument, besides the action a and the state x before the edge. Its soundness is proved under the assumption that the channel holds. The transfer must lose no successor, as in Section 6.1, but only of stores at which every answer is true (`ls_step_sound_iff`). Whether the answers are true is not the transfer’s concern, and Section 7.3 shows that the framework always supplies a channel that holds.

For `ask_assign`, this assumption is exactly what the proof needs. In the example, only the stores with $x = y$ remain, because the answer 1 is false at all others. At these stores, e evaluates to 1, so the analysis’s own assignment of the literal 1 is sound for e as well (`ask_assign_sound`). The proof never mentions the order analysis and works with any analyses whose answers hold.

An analysis without analysis globals is written as a *local specification* (`local_spec`). It bundles the operations of Section 6.1, each taking a channel, with a handler `ls_query` that answers queries about a state, so that the analysis can answer queries from the analyses it runs with.

```

1 (* src/Abstract_Interpreter/Framework/Cooperation/MCP_Spec.thy *)
2 record 's local_spec =
3   ls_query :: "channel ⇒ 's ⇒ channel"
4   ls_skip  :: "channel ⇒ 's ⇒ 's"
5   ls_assign :: "channel ⇒ vname ⇒ exp ⇒ 's ⇒ 's"
6   ls_special :: "channel ⇒ special_call ⇒ vname ⇒ 's ⇒ 's"
7   ls_branch :: "channel ⇒ exp ⇒ bool ⇒ 's ⇒ 's"
8   ls_body  :: "channel ⇒ pname ⇒ 's ⇒ 's"
9   ls_return :: "channel ⇒ exp option ⇒ pname ⇒ 's ⇒ 's"
10  ls_event  :: "channel ⇒ analysis_event ⇒ 's ⇒ 's"
11  ls_enter  :: "channel ⇒ call_info ⇒ 's × 's ⇒ ('s × 's) list"
12  ls_combine_env :: "channel ⇒ channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's"
13  ls_combine_assign :: "channel ⇒ call_info ⇒ 's ⇒ 's ⇒ 's"

```

Every operation receives a channel. The first combine stage receives two, one with answers about the caller’s state and one with answers about the callee’s exit state.

A local specification is sound (`sound_local_spec`) if its operations satisfy the laws of the analysis contract (Section 6.2), each under the assumption that its channels hold, as for the transfer above. This amounts to a monotone concretization and one law per operation (`sound_local_spec_iff`). The edge and combine laws correspond to those of the analysis contract, and the enter law to entry coverage. The handler adds a law of its own. Its answers about a state x must hold at every store of $\gamma(x)$ at which its own channel holds (`sound_query`), and the analyses rely on one another only through this law. Verasco proves each transfer function under the hypothesis that the channels it receives are correct [22, §7], following Astrée [52, §§5.3, 6], and Voblint applies the same form of proof to Goblint’s queries.

7.3. Combining analyses into one sound analysis [thy](#)

The analyses cannot be solved one after another, because each may need the other's facts at the same program point and context. In the program below, the exchange runs in both directions, here between the order analysis and Int, the reduced product (Section 11.2).

```

1 fun main() {
2   c = __voblint_nondet_int();
3   if (0 < c) { x = 0; y = 10; } else { x = 20; y = 30; }
4   z = (x <= y);
5   __voblint_check(z == 1);
6 }

```

Int alone joins the branches, so z lies only in $[0,1]$, and the check is **UNKNOWN**. The order analysis alone cannot compare a variable with a constant and reports **UNKNOWN**. Together, at both assignments to y , the order analysis asks Int whether x is at most the assigned constant and records the pair (x, y) on both branches. At $z = (x \leq y)$ Int asks **EvalInt** ($x \leq y$), the order analysis answers 1, and the check is **PROVED** (claims **coop-mutual-***).

The framework therefore runs all selected analyses as a single analysis whose abstract state is a tuple with one field per analysis. For the solver, the tuple is one abstract state per node and context, like any other. It is ordered, joined, and widened field by field (**analysis_product**), so each field evolves as it would alone, except that its transfers may ask the other fields. If one selected field shows that a node is unreachable, the whole product state does (**mcp_norm**).

The product state **mcp_st** has one field per analysis the analyzer ships, and a run uses the fields of the analyses the user selects (Figure 7.2). The product state describes a store if every selected field does:

```

1 (* src/Abstract_Interpreter/Framework/Cooperation/MCP_Spec.thy *)
2 definition mcp_gamma :: "('s ⇒ store set) list ⇒ 's ⇒ store set" where
3   "mcp_gamma Γ x = (⋂ γi ∈ set Γ. γi x)"

```

Here, Γ is the list of concretizations of the selected fields. Each $\gamma_i \in \Gamma$ reads its own field of x , so $\gamma_i(x)$ is the concretization of that field.

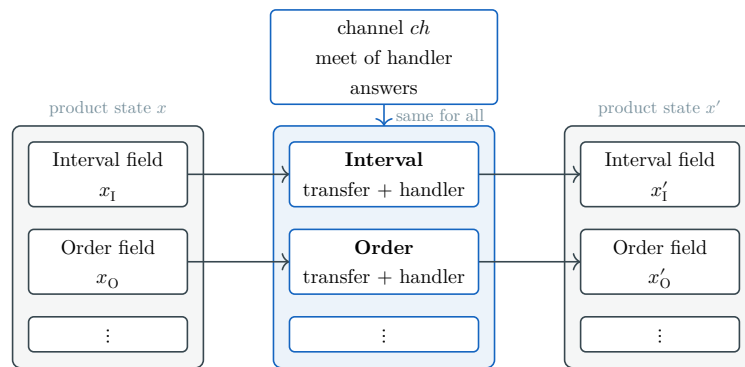


Figure 7.2: One step of the combination (schematic). Each component reads its own field of the product state x and writes its own field of x' , and the components run in turn. All components receive the same channel ch (blue).

A local specification that runs on one field is called a *component*. Each component reads and writes only its own field, through a *lens* (**lens_of**) that takes the field out of the

product state and puts a new value back, as Goblint's MCP hands each analysis its own part of the product state. A component therefore leaves the concretizations of all other fields unchanged (`field_frame`), which makes the components *independent* (`mcp_independent`). `mcp_combine` runs the components in turn, gives each of them the same channel, and meets the answers of their handlers. If the list of components is nonempty, the components are independent, and each is a sound local specification, then the combination is a sound local specification for the intersection of their concretizations (`mcp_combine_sound`). The components cannot invalidate one another's guarantees, since each guarantee is about a field no other component writes. Instantiated with the analyses Voblint ships, the theorem gives the result of this chapter. Any nonempty list `as` of distinct analyses combines into a sound local specification:

```
1 (* src/Executable_Surface/CLI/MCP_Analyses.thy *)
2 theorem mcp_comp_sound:
3   assumes "distinct as" and "as ≠ []"
4   shows "sound_local_spec (declared_global p)
5         (mcp_gamma (map (part_gamma (declared_global p)) as))
6         (mcp_comp as (declared_global p) p)"
```

Running the combined handler on the current state x gives a channel for the premise of the theorem (`ls_channel`). A query that comes back while it is being answered gets $\top$, as in Goblint's `MCP.query`. Every answer of this channel holds at every store of $\gamma(x)$ (`ls_channel_sound`), so `dg_spec_of` can pass it to the operations and turn the combination into a record that meets the analysis contract (`dg_spec_of_contract`).

Each analysis is thus proved sound once, against any channel that holds, and its proof never mentions the analyses it runs with. Their combination meets the analysis contract and decides the check of the opening example. Adding an analysis requires only its own proof, and the proofs of the others stay unchanged. Chapter 8 discharges the obligations of Chapter 4 for any analysis that meets the analysis contract, the combination included.

PART IV

Computing Sound Results

8. Context-Indexed Equations and Routing

Chapter 4 reduced soundness to a cover, which assigns to every node v and context c a set of stores `cover v c`, and stated five local obligations under which the cover contains every store that an execution brings to v in context c . Chapter 6 and Chapter 7 stated what an analysis supplies and proves. What is still missing is a way to compute such a cover. This chapter turns the procedure-aware graph of Chapter 3 into a system of equations and shows that every post-solution induces a sound cover.

Precisely, every post-solution σ of the equations should satisfy

$$\mathcal{A}_{g, \text{adm}, c_0, g, S} \text{ v } c \subseteq \gamma_M(\sigma(v, c))$$

at every node v and context c . The left side is the activation collecting semantics of Section 4.3, and $\gamma_M(\sigma(v, c))$ is the set of stores that the solved abstract state at (v, c) describes under the solved values of the analysis globals (Section 6.2). For the edges inside a procedure, the equations follow the graph directly. A call is harder, since the equations must also determine which context of the callee it enters. Under entry-state policies, this context is computed from the abstract state at the call, which is known only while the equations are being solved.

8.1. Context-indexed unknowns and call equations thy

Separate contexts give the activations of `bump(5)` and `bump(4)` separate copies of `bump`, so the first result stays exactly 6 (Section 2.6).

Voblint therefore indexes the unknowns of the equations by contexts. For every node v and context c , there is a *local* unknown (v, c) , whose value $\sigma(v, c)$ is the abstract state for the activations of context c at v . These are the indices of the cover. Goblint’s local unknowns have the [same shape](#) [51, §5.2]. Besides the local unknowns, the equations have *global* unknowns without a program point (Section 2.8). They hold the analysis globals of Section 6.3 and the entry states that calls publish to their callees.

The equation of (v, c) has to collect every abstract state that can reach v in context c , because a post-solution must cover every store that reaches v there. Such states come from four sources: the initial state at the program entry, the transfer along each local edge into v , the result of each call that returns to v , and, at a procedure entry, the entry states that calls publish to it. An entry state is the abstract state in which the callee starts, for example, with its formal parameters bound to the abstract values of the call’s arguments (Section 6.1). Along a local edge, the activation keeps its context (Section 4.3), so the second source applies the transfer to the predecessor’s value in the same context c . A callee, however, runs in a context of its own.

As in Section 2.4, a right-hand side reads the current values of other unknowns from a *valuation*, which assigns a value to every unknown. Given a valuation τ , the generated right-hand side `routed_node_rhs` of (v, c) computes exactly the join of the four sources:

```

1 (* src/Abstract_Interpreter/Framework/Constraints/DG_Indexed_Generator.thy *)
2 lemma routed_node_rhs_parts:
3   assumes wf: "\w. \p \in set (routed_contribution_programs pred_sel site_sel route it cmb
4     extra g c w). sp_wf p"
5   shows
6     "eq (routed_node_rhs pred_sel site_sel buffer_key_at route it cmb extra g bot0 d0 g0)
7       (v, c) \tau =
8       DG (rhs_init g bot0 d0 v
9         \ (\u, a) \leftarrow pred_sel g v c. rhs_edge it \tau c u a)
10        \ (\u, ca) \leftarrow site_sel g v. rhs_call cmb route \tau c u ca v)
11        \ rhs_seed extra route \tau c v) bot"

```

Here $\bigsqcup_{x \leftarrow xs} f(x)$ joins $f(x)$ over the list xs , and the empty join is $\perp$. The lists range over the local edges into v and over the calls that return to v . The parts `rhs_init`, `rhs_edge`, `rhs_call`, and `rhs_seed` are the four sources in this order. A post-solution σ assigns (v, c) a value at least as large as this join. Since γ_M is monotone, $\gamma_M(\sigma(v, c))$ therefore contains the stores that each source describes. Publications to analysis globals (Section 6.3) are side effects of the right-hand side and do not enter this join.

At a call, the analyzer chooses the callee's context with a *routing* function $\text{context}^\sharp(u, c, e)$ of the call site u , the caller context c , and the entry state e . Under entry-state contexts, the choice depends on e , which is known only while the equations are solved. The set of calls that contribute to a callee context is therefore not known in advance, and the callee's entry cannot list them in its equation. Each call instead publishes its entry state to the callee as a side effect (Section 2.7).

Figure 8.1 follows the first call of the running example. The equation of its continuation, $(pp3, c_0)$, connects the caller and the callee. It reads the caller's value at $(pp2, c_0)$ and applies `enter` (Section 6.1), which yields an *entry pair* of a resume state q_1 and an entry state e_1 that binds n to $[5, 5]$. The entry state selects the callee context $c_1 = [[5, 5]]$. The equation publishes e_1 to that context, reads the exit state r_1 of `bump` in c_1 , and joins $\text{combine}^\sharp(q_1, r_1)$ into its own value. The equation of the second call's continuation, $(pp4, c_0)$, does the same with $c_2 = [[4, 4]]$. The two calls thus use separate copies of `bump`, and the first result stays exactly 6.

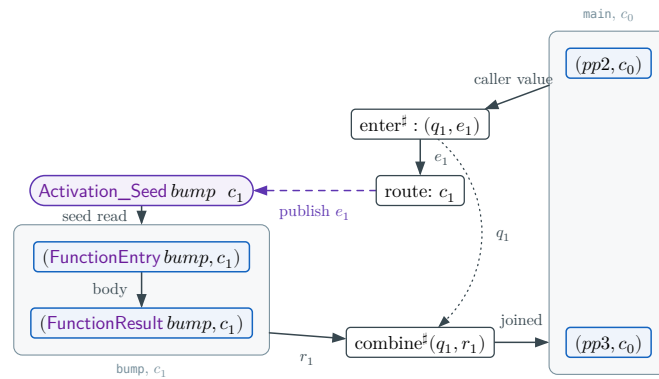


Figure 8.1: The first call of the running example under entry-state contexts. The right-hand side of the continuation $(pp3, c_0)$ reads the caller's value, applies `enter`, routes the entry state e_1 to c_1 , publishes it to the seed of `bump` in c_1 (purple, dashed), reads the exit state in the same context and joins the combined state into its own value. Blue boxes are local unknowns, the purple box is a global unknown, white boxes are steps of the right-hand side, and gray frames group the unknowns of one procedure in one context.

In this protocol, the caller's equation publishes the entry state to the callee and reads the callee's exit state. The verified TD solver, however, accepts side effects only to global unknowns (Section 9.1.2), so a call cannot publish to the local unknown of the callee's entry directly. Each callee entry p and context c' therefore has a global unknown, the *seed* `Activation_Seedp c' (global_unknown)`. Every call routed to (p, c') publishes to it, and the equation of `(FunctionEntryp c')` reads it. A call with several callees, or an enter that returns several entry pairs, routes and publishes each pair separately.

Since a caller reads nothing of the callee except its exit state, a recursive call needs no special treatment. Its equation reads an exit state that in turn depends on the caller's own unknowns, so the dependencies form a cycle. The solver handles this cycle like the cycle of a loop and widens at the unknown where a read closes it (Section 2.8).

The right-hand side handles one special case. A call whose entry state is bottom publishes nothing and does not read the callee's exit state. No store enters the callee in that case, and the read would only make the solver evaluate the callee's body from a state that describes no store. Goblint's normal-call transfer `skips` bottom entry states in the same way. Skipping them is sound because a state classified as bottom is required to have an empty concretization (`is_bot_sound`).

8.2. Routing that agrees with the context policy `thy`

Every call is assigned a callee context twice. The equations choose one from the abstract entry state, and the concrete semantics of Chapter 4 chooses one from the concrete call, through the context policy *adm* of Section 4.3. The unknowns of a context receive only what the equations publish there, but they must describe every concrete activation that the policy admits in it. The two choices must therefore agree. In the running example, the policy admits the activation of `bump` with $n = 5$ in the context $[[5, 5]]$, and the call `bump(5)` publishes an entry state with $n = 5$ to the same context. In general, whenever the policy admits a concrete call in a context, the equations must publish to that context an entry pair that describes the call (Figure 8.2). For a caller store s that the caller's solved state describes and the entered store s' :

$$\begin{aligned} \text{adm admits } c' \text{ for } (s, s') \implies & \text{some entry pair } (q, e) \text{ is routed to } c' \\ & \text{with } s \in \gamma(q) \text{ and } s' \in \gamma(e). \end{aligned}$$

This condition is called *adequacy* (`routing_adequate`). It also requires the callee's entry unknown in c' to be in the solved set. `CALL` and `RETURN` rely on adequacy, because the call publishes e and its continuation reads the callee's exit state in the same context c' .

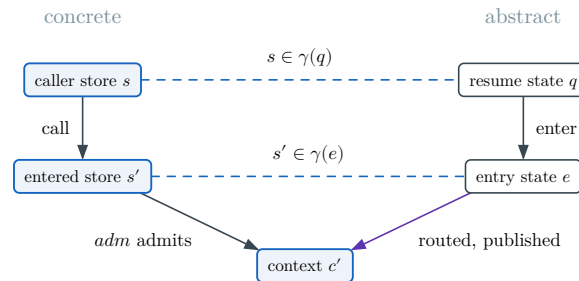


Figure 8.2: Adequacy. If the concrete semantics assigns a call to c' (left), the equations must route a covering entry pair to the same c' (right).

Adequacy says nothing about a call to which *adm* assigns no context. Such a call would disappear from the activation collecting semantics, which **TOTAL** of Section 4.4 rules out. The second condition, *totality* (**routing_total**), is what **TOTAL** asks of the generated equations. It requires *adm* to admit at least one context for every concrete call from a store that the caller's solved state describes. Both conditions are premises of the soundness theorem and are proved once for each context policy.

policy	<i>adm</i> admits c' iff	routed to	why they agree
unit	$c' = ()$	$()$	both constant
call strings	$c' = \text{take}_k(u \cdot c)$	$\text{take}_k(u \cdot c)$	the same term
entry states	$c' = e _{\text{formals}}, (q, e) \text{ covers } (s, s')$	$e _{\text{formals}}$	<i>adm</i> reads the routing

Table 8.1: The contexts *adm* admits for a call at site u from caller context c with caller store s and entered store s' , where the equations route the entry state e , and why the two agree. $u \cdot c$ prepends the site, take_k keeps the first k entries, $e|_{\text{formals}}$ lists the formals' values in e , and an entry pair (q, e) of the caller's solved state covers (s, s') when $s \in \gamma(q)$ and $s' \in \gamma(e)$. Unit and call strings are *functional* policies, since they choose the context without reading abstract states.

For unit and call strings, the correspondence is direct, as the last column of Table 8.1 shows. *adm* admits exactly the context that the routing function computes for the call, which does not depend on e (**context_policy_of_fun**), so totality is immediate. Adequacy then reduces to the entry coverage of Section 6.1 and a routed entry unknown in the solved set.

Figure 8.3 shows how the policies differ on a call chain. A call string of length one separates the two calls of **wrap** but merges them again in **scale**, where both arrive from the same call site. Length two and entry-state routing keep them apart.

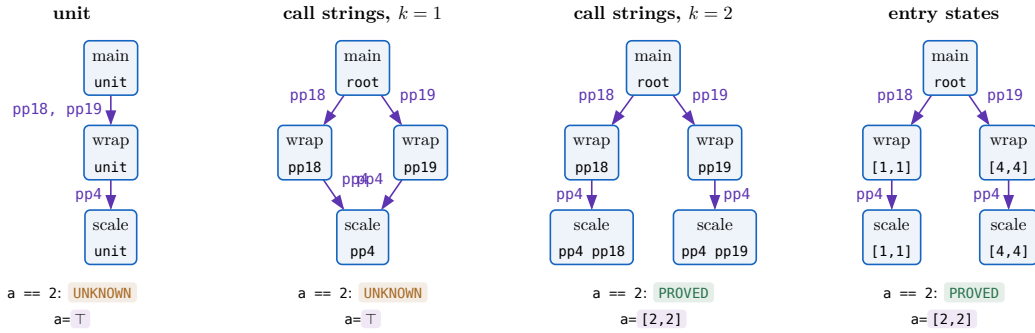


Figure 8.3: Procedure copies under four context policies, on a nested-call program: **scale**(v) returns $2 * v$, **wrap**(w) returns **scale**(w) from **pp4**, and **main** calls $a = \text{wrap}(1)$ at **pp18** and $b = \text{wrap}(4)$ at **pp19**, then checks $a == 2$. Boxes are copies labeled with their contexts (root is that of **main**), and arrows are calls labeled with their sites (Interval, claims **ctx-demo-***).

An entry-state context consists of the abstract values that the entry state e gives the callee's formal parameters, written $e|_{\text{formals}}$. For **bump**(n) and an e that binds n to $[4, 5]$, the context is $[[4, 5]]$, a list with one entry per formal. Unlike a call string, such a context depends on the abstract entry state, so the concrete call alone does not determine it.

The entry-state policy (`routed_entry_context_rel`) therefore reads the context of a concrete call from the solution at that call. It admits a context exactly when the equations, at this call site and in this caller context, route an entry pair that describes the call to it. Suppose the caller's solved state at a call `bump(x)` says $x \in [4, 5]$. The equations publish $n \mapsto [4, 5]$ to $[[4, 5]]$, so a run that passes this call with $x = 4$ is admitted in $[[4, 5]]$ and not in $[[4, 4]]$. At another call whose solved state says $x \in [4, 4]$, the equations route to $[[4, 4]]$, and a run with $x = 4$ through that call is admitted there. The policy thus copies the equations' routing call by call. Every admitted context has received an entry state that describes the call, which is adequacy. Entry coverage (Section 6.1) guarantees such an entry pair for every call from a store that the caller's solved state describes, which is totality.

8.3. Post-solutions satisfy the coverage contract thy

obligation	guarantee of the generated equations
INIT	the initial state at the program entry covers the initial stores
INTRA	the transfer along each local edge is sound
CALL	the routed entry state covers the entered store (adequacy)
RETURN	the exit state is combined soundly in the same context (adequacy)
TOTAL	every covered call is admitted in some context (totality)

Table 8.2: How the generated equations meet the five obligations of Chapter 4, for an arbitrary program.

Each obligation of Chapter 4 is met by one part of the generated equations (Table 8.2). The argument holds for every domain and context policy. Its locale `routed_context` assumes a post-solution σ on a solved set closed under local edges and call continuations, the `analysis_contract` of Chapter 6, a sound emptiness test, a sound callee resolution, and adequate and total routing. The theorem adds that the program entry is solved in the initial context and that the initial state d_0 describes the initial stores S_0 .

Theorem 8.1 (Routed collecting soundness, `activation_collect_dg_sound`).

```
1 (* src/Abstract_Interpreter/Framework/Context/Routed_Context.thy *)
2 theorem activation_collect_dg_sound:
3   fixes S0 :: "store set" and c0 :: 'c
4   assumes entry_cov: "(cfg_entry g, c0) ∈ vars"
5   and init_sound: "S0 ⊆ γDG d0 e0"
6   and init_env_le: "e0 ≤ genv global_of sigma"
7   shows "AG, adm, c0, g, S0 v c ⊆ cover v c"
```

Isabelle

The solver gives guarantees only for the unknowns in its solved set, so the cover `cover v c` is $\gamma_M(\sigma(v, c))$ there and empty elsewhere. No execution reaches an unknown outside the solved set, since the set contains the program entry and is closed under local edges, call continuations, and routed calls. The proof checks each obligation for this cover, one row of Table 8.2 each. Instantiated for the policies of Table 8.1, the theorem covers the analyzer's equations, and Chapter 9 computes such a post-solution.

9. Solving the Equations

Chapter 8 showed that a post-solution of the generated equations is sound. At every node and context, its abstract state describes every store that an execution reaches there in that context (`activation_collect_dg_sound`). The solver, however, makes the valuation a post-solution only on its solved set, so the theorem requires this set to contain the program entry and every unknown that an execution can reach from it. This chapter shows that the analyzer obtains such a solved set and post-solution. Voblint solves the equations with the verified TD solver of Tilscher et al. [4], which is proved partially correct. To use it, Voblint first brings the equations into the form the solver accepts and then derives the premises about the solved set from the solver’s correctness theorem.

9.1. Encoding the equations for the solver

The solver accepts a right-hand side only as a strategy tree, uses one value type for all unknowns, and allows side effects only to global unknowns. A direct encoding of the equations, as in Apinis et al. and Goblint, needs four things this interface does not offer (Table 9.1). Chapter 8 already stated the equations in the adapted form, and this section explains why each adaptation is needed. Buffering is the only adapter applied to a right-hand side after it is built. It changes when publications are issued but not what the right-hand side computes, publishes, or reads (Section 9.2). The adapters change the equations, not the solver.

the equations need	the solver provides	adapter
value-dependent reads	strategy trees	strategy tree (Section 9.1.1)
publish to local unknowns	publish to globals only	seed (Section 9.1.2)
two value types	one value type	product (Section 9.1.3)
publish twice to one target	updates that may not stabilize	buffering (Section 9.2)

Table 9.1: The requirements of the equations of Apinis et al. and Goblint, the alternative the vendored TD solver provides, and the adapter that closes each gap. A call chooses the result it reads from the caller’s state, publishes the callee’s entry state to a local unknown, keeps local states and analysis globals in different domains, and may publish to one seed twice in one evaluation. The solver accepts such publications but updates after each one, and under warrowing these updates may not stabilize.

9.1.1. Value-dependent reads as strategy trees thy

A local edge always reads the same unknown, its predecessor. A call, in contrast, chooses what to read from values it has already read. In the running example, the

continuation $(pp3, c_0)$ first reads the caller $(pp2, c_0)$. From that value, it computes the entry state $n = [5, 5]$, which determines the context c_1 and thus the result it reads next, $(\text{FunctionResult } bump, c_1)$. The vendored TD solver accepts a right-hand side only as a strategy tree, which issues one read at a time and passes each value read to a continuation (Section 2.8). Every right-hand side of Voblint is such a tree. For a local edge, the tree reads the predecessor and applies the transfer. For a call it chains reads whose continuations choose the next read from the value received. In the notation of Section 2.8, the value of a call at u in context c (rhs_call) is the value of the tree t_{call} ,

$$t_{\text{call}} = \text{QueryL } ((u, c), \lambda d. \\ \text{Side } (\text{Activation_Seed } p \ c', e, \\ \text{QueryL } ((\text{FunctionResult } p, c'), \lambda r. \text{Answer } (\text{combine}^\#(q, r))))),$$

The tree first reads the caller's state d at the call site and computes the entry pair $(q, e) = \text{enter}^\#(d)$ and the callee context $c' = \text{context}^\#(u, c, e)$. It then publishes the entry state e for the callee's entry in c' , reads the callee's exit state r , and answers $\text{combine}^\#(q, r)$, the abstract state of the caller once the call has returned. `routed_call_program` implements this tree and additionally handles bottom entry states and several callees or entry pairs.

9.1.2. Publishing entry states to global seeds

In t_{call} , the caller pushes the entry state to the callee with a `Side` step. Otherwise, the equation of the callee's entry would have to read every call routed to it. Under entry-state contexts, however, these calls depend on the entry states, which are known only while the equations are solved (Section 8.1). Apinis et al. push the entry state directly into the callee's entry (`FunctionEntry` p, c') [24, §6], and Goblint does the same with its local side effect `sidel` (`constraints.ml`). That entry, however, is a local unknown, and the vendored TD solver's `Side` accepts only global unknowns. Allowing local targets would mean changing the solver and redoing its correctness proof.

Voblint therefore publishes the entry state to the callee's seed, a global unknown that the entry reads (Section 8.1).

Seeds have a known disadvantage compared with a side effect into the local entry. As global unknowns, they are merged by the update rule for globals (Section 9.2), which may widen an entry state even when no loop or recursion is involved. Without contexts, both calls of the running example publish to the one seed of `bump`, and warrowing widens it. The analyzer then reports `UNKNOWN` for `a == 6` with `a = [-∞, 6]`, whose lower bound $-\infty$ comes from this widening.

9.1.3. Local and global values in one product thy

A local unknown holds an abstract state of the analysis's local domain, and an analysis global holds a value of its global domain (Section 6.3). The vendored TD solver, however, has a single value type $'d$ for all unknowns, so Voblint stores both kinds in their product: every unknown holds a pair `dg_state` of a local and a global half, `dg_local` and `dg_global`, uses one half and leaves the other at $\perp$ (Table 9.2). Order and join on pairs work componentwise, so the solver's requirements on the value type follow from those on the two halves. The half an unknown uses depends on what it stores, not on whether the solver treats it as

solver unknown	role	value
(v, c) , local	the state at v in context c	$(d, \perp)$
<code>Activation_Seed</code> , global	an entry state published by a call	$(e, \perp)$
<code>Analysis_Global</code> v , global	the analysis global v	$(\perp, g)$

Table 9.2: Which half of `dg_state` each kind of unknown uses. The global unknowns are named by `global_unknown`.

global. A seed, for example, is a global unknown, because calls publish to it, but it stores an entry state, which is a local state.

Transfers never see the pair or the seeds. Goblint instead combines the two kinds of value in a `lifted sum` in which each unknown holds a value of exactly one of the two kinds. Voblint uses the product because its componentwise lattice structure keeps the Isabelle proofs simple, at the price of an unused $\perp$ half in every unknown.

9.2. Update rules for global unknowns thy

A global unknown, such as a seed, receives contributions from several places. The *update rule* merges each new contribution into the value the unknown holds and decides both precision and termination. Tilscher et al. state the solver’s requirements on an update rule as the locale `update_rule` and prove that five rules satisfy it [4], so Voblint can offer all five as a user setting (`update_rule_update_global_of`). Table 9.3 compares them.

The rules differ in what they keep per *origin*, the right-hand side that published a contribution (Section 2.8), and in whether they widen. Of the two *joining rules*, *join* joins every contribution into the value, and *join per origin* joins each origin’s latest contribution. Of the three *warrowing rules*, *warrow* and *warrow per origin* warrow toward the same join or per origin, and *bounded narrowing* also limits how often an origin may switch from widening to narrowing. In `run_voblint`, the rule governs how a callee’s entry state and a flow-insensitive program global (Section 6.3.2) accumulate their contributions. No rule is more precise on every program (Section 13.3).

rule, vendored update	1: A publishes [0, 3]	2: B publishes [4, 4]	3: A publishes [1, 1]	4: A publishes [0, 8]
join	[0, 3]	[0, 4]	[0, 4]	[0, 8]
<code>update_global_always_join</code>	[0, 3]	[0, 4]	[0, 4]	[0, 8]
join per origin	[0, 3]	[0, 4]	[1, 4]	[0, 8]
<code>update_global_per_origin</code>	[0, 3]	[0, 4]	[1, 4]	[0, 8]
warrow	[0, 3]	[0, +∞] ∇	[0, 4]	[0, +∞] ∇
<code>update_global_warrowing_apinis</code>	[0, 3]	[0, +∞] ∇	[0, 4]	[0, +∞] ∇
warrow per origin	[0, 3]	[0, 4]	[0, 4]	[0, +∞] ∇
<code>update_global_warrowing_per_origin</code>	[0, 3]	[0, 4]	[0, 4]	[0, +∞] ∇
bounded narrowing, bound 5	[0, 3]	[0, 4]	[0, 4]	[0, +∞] ∇
<code>update_global_bounded_narrowing</code>	[0, 3]	[0, 4]	[0, 4]	[0, +∞] ∇

Table 9.3: One global unknown under each update rule, from $\perp$, after four interval contributions from origins A and B, which separate the rules and come from no program. Widened bounds are bold and marked ∇. The cells are computed by `update_rules_example` from the vendored rules.

One right-hand side can publish to the same global unknown twice, for example, when two calls that resume at the same node are routed to the same callee context and publish to the same seed. The equations read two publications to one target g as one bound:

Side (g, a) ; **Side** (g, b) means $\sigma(g) \sqsupseteq a \sqcup b$.

The vendored TD solver, however, applies the update rule at every **Side**. A per-origin rule (Section 2.8) therefore records first a and then $a \sqcup b$, so in every re-evaluation the recorded contribution shrinks and grows back. Under warrowing, the shrinking step narrows, the growing step widens again, and the solver may not stabilize. The update rules of Stemmler et al. assume one joined contribution per target and evaluation [53, §3], and Goblint, when `narrow-globs` is enabled, joins the side effects of an evaluation before its per-origin update ([td3UpdateRule.ml](#)).

Voblint joins them as well. `buffer_sides` collects the publications of a right-hand side and joins those to the same target until a flush point. Where several calls return at a node, it flushes only when the right-hand side answers. Elsewhere, it flushes before every local read, so that an entry state reaches the callee before the callee’s exit state is demanded, as in Goblint’s normal-call transfer ([constraints.ml](#)). The buffered generator `routed_node_rhs_buffered`, which the analyzer runs, answers the same join as the direct generator of Section 8.1. Under warrowing rules, the timing can change the computed result, though not the equations it solves. Section 15.3 discusses a solver extension that would make seeds and buffering unnecessary.

9.3. Solver certificates and partial correctness

The soundness theorem of Section 8.3 needs a post-solution on a set of unknowns that contains the program entry and is closed under execution steps. A solve that returns provides something different. It gives a valuation σ and a set V of local unknowns on which σ is a post-solution, and V is closed under the reads of right-hand sides, starting from the result of `main`. The solver thus works backward from that result, whereas the theorem follows executions forward from the entry. The two directions meet because every node of a procedure reaches the procedure’s result, even code after a `return`, since the compiler ends every body with a return edge (Section 3.4.1). Seidl and Fecht call this property (R) and show that under it, the unknowns on which the query for `main` depends account for every reachable state, for any solution and not only the least one [54, §9.1, Thm. 7(2), Thm. 9(2)]. Once a procedure’s result is solved in a context, closure under reads makes all its nodes solved in that context. Executions from the entry of `main` stay among these nodes, which `live_unknowns_cover` proves from a well-formed program and a returned solve alone. It mechanizes Seidl and Fecht’s argument and extends it to side effects and seed unknowns, to dependences that the solver discovers while it solves, and to context policies given as relations. The final theorem therefore needs no premise about which unknowns the solve visited.

The solver is not proved to terminate. HOL admits only terminating recursion, so Isabelle defines the solver with a domain predicate (`solve_dom`, Appendix A), the arguments on which its recursion terminates, and proves its correctness on that domain. The analyzer runs the executable form `solve_c`, which returns only when the recursion finishes, and `solve_dom_of_solve_c` turns a returned run into domain membership. The theorems about `run_voblint` therefore have no termination premise. The solve starts from one query, the

result of `main` in the initial context (`root_query`, as in [24, §3]), and evaluates an unknown only when a right-hand side it is solving reads it [4], [40]. It thus discovers contexts as it goes, and whether it returns depends on how many unknowns it reaches. Under entry-state contexts, a recursion can demand unboundedly many (Section 13.3). The existing termination theorems do not apply to Voblint (Section 14.4), so the end-to-end soundness theorem (Section 10.4) is a partial-correctness result about every returned report.

The rest of the soundness proof never looks inside the solver. It needs the returned valuation to be a post-solution on the solved set, which is the certificate below, and the solved set to be finite, so that Chapter 10 can join the finitely many contexts solved at a node into one verdict. Voblint therefore keeps the solver as a parameter of the analysis. The locale `certified_solver` states what the proof needs from a solver: these two requirements, and that the executable code computes the same result as the solver defined in the logic. The analysis locale `dg_analysis` extends it, and `td_certified_solver` proves it for the verified TD solver under every update rule.

The verified TD solver states its guarantee as the certificate `part_post_solution` of Section 2.7, over the equation system T , the valuation σ and the solved set `vars`:

```
1 (* vendor/td-verification/Basics_side.thy *) Isabelle
2 abbreviation part_post_solution ::
3   "'x, 'g, 'd :: bounded_semilattice_sup_bot) eqsT ⇒ 'x ⇒ ('x + 'g ⇒ 'd) ⇒ 'x set
4   ⇒ bool" where
5   "part_post_solution T x sigma vars ≡
6     x ∈ vars ∧ (∀u ∈ vars.
7       depL T sigma u ⊆ vars ∧
8       eq T u sigma ≤ sigma (Inl u) ∧
9       sides_of_rhs (T u) sigma ≤ sigma)"
```

T (`eqsT`) maps each unknown u to its strategy tree $T u$. Evaluating a tree under the final valuation σ determines three things: the answer it returns, the local unknowns it reads, and the contributions it publishes. In Isabelle these are `eq` $T u \sigma$, `depL` $T \sigma u$, and `sides_of_rhs`, which joins the published values per target and is $\perp$ where the tree publishes nothing. With them, the certificate states four facts about the query x and the solved set V :

$$x \in V \quad (C1)$$

$$\forall u \in V. \text{dep}_L T \sigma u \subseteq V \quad (C2)$$

$$\forall u \in V. \text{eq } T u \sigma \sqsubseteq \sigma(u) \quad (C3)$$

$$\forall u \in V. \text{sides_of_rhs } (T u) \sigma \sqsubseteq \sigma \quad (C4)$$

(C1) puts the query into the solved set, and (C2) closes the solved set under reading. (C3) is the post-solution bound for local unknowns, and (C4) bounds every published contribution.

The certificate supplies what Section 8.3 assumes about the solve. (C3) and (C4) make σ a post-solution for the local and the global unknowns. (C1) and (C2), by the closure argument above, give a solved set that contains the program entry and is closed under execution steps. Earlier chapters discharge the remaining premises. The states of every returned solve therefore cover the activation collecting semantics (`entry_state_activation_collect_sound_of_terminates`), and the five obligations of Chapter 4 hold as Table 8.2 shows. Chapter 10 turns these states into verdicts.

10. From Solutions to Verdicts

This chapter derives the guarantees of the analyzer’s report for the executions of the source program. Solving the equations of Chapter 8 (Chapter 9) gives an abstract state for each CFG node and context the solver reached, as well as a value for each global unknown. By Section 9.3, these states cover every execution. A user, however, reads neither the states nor the contexts. The user reads a report with one verdict for every `__voblnt_check` and a diagnostic wherever a division may fail. The chapter shows how the analyzer derives this report from the solution and why every verdict in it is sound.

10.1. The analyzer function and its report thy

The constructions of the previous chapters are packaged behind one HOL function, `run_voblnt`. It takes an analysis configuration and a VIMP program. The configuration (`analysis_config`) packages the choices of the earlier chapters: the analyses that run together (Chapter 7), the update rule (Section 9.2), the context policy (Section 8.2), and the placement of program globals (Section 6.3.2). The analyzer compiles the program to its graph `prog_cfg p` (Chapter 3), builds the equation system of the configured analyses and context policy (Chapter 8), solves it from the exit of `main` (Chapter 9), and builds a report from the solution.

```
1 (* src/Executable_Surface/CLI/Analysis_Run.thy *)
2 definition run_voblnt ::
3   "analysis_config ⇒ imp_prog ⇒ analysis_report analysis_answer"
4 where
5   "run_voblnt config p =
6     (if ¬ valid_config config then Invalid_Activation
7      else if ¬ wf_program_compile_input_exec p then Malformed_Program
8      else case analysis_report_of config p of
9        None ⇒ No_Answer
10       | Some res ⇒ Analysed res)"
```

Before it compiles anything, `run_voblnt` checks its two inputs, so that a report exists only where the soundness proofs apply. `valid_config` requires a nonempty list of distinct analyses, and `wf_program_compile_input_exec` decides the structural conditions that the compilation theorems of Chapter 3 assume. An input that fails is answered with `Invalid_Activation` or `Malformed_Program`, and `No_Answer` stands for a solve that produces no result. Every answer `Analysed res` thus comes from inputs that pass both tests (`run_voblnt_report_contract`), and the executable test implies the well-formedness premise of the proofs (`wf_program_compile_input_exec_sound`, `run_voblnt_wf`). The end-to-end soundness theorem of Section 10.4 therefore needs no premise about the configuration or the program.

The report that `Analysed` carries (`analysis_report`) keeps the solver’s states under their node and context, the values of the global unknowns, a verdict for every check (Section

10.2), and the arithmetic diagnostics (Section 10.5). A node that the solver solved in several contexts carries several states. At a node v , the report describes every store that one of them describes ([report_states_at](#), [report_sem](#)):

$$\mathcal{R}_{\text{res } v} = \bigcup_{d \in \text{report_states_at res } v} \gamma_{\text{res}} d,$$

where γ_{res} is the concretization of the configured analyses. The union drops the contexts, since a source execution carries no context to compare with, and a user asks what holds at a program point, whichever call led there. The proof, however, needs the contexts, since the coverage contract bounds the stores of each context separately (Section 4.4). The report therefore keeps them, and each state names its context. The states at v in context c describe the set [report_context_sem res v c](#) ([report_context_sem](#)), a subset of $\mathcal{R}_{\text{res } v}$. Section 10.3 shows that this set contains every store that an activation in context c reaches at v .

10.2. Computing check verdicts from the report

The command-line tool prints the report through its checks: each check at v receives one of four verdicts, [PROVED](#), [REFUTED](#), [UNKNOWN](#), or [DEAD](#). The [browser playground](#) shows the same verdicts inline in the editor and beside the graph, together with the state of every node and context, and a replay of the solve (Section 12.4, Section 12.3). Each verdict is printed at the source position of its check, and the text report checks that these positions are unique ([run_voblint_labelled_check_sound](#)).

The analyzer first computes one verdict per state. Each state d in [report_states_at res v](#) gives its own verdict for a check:

```
1 (* src/Abstract_Interpreter/Framework/Checks/Contextual_Check_Report.thy *) Isabelle
2 fun classify_point ::
3   "(exp ⇒ 'a ⇒ check_result) ⇒ exp ⇒ 'a lifted ⇒ contextual_verdict"
4 where
5   "classify_point classify c Bot = Dead"
6 | "classify_point classify c (Lifted st) = Decided (classify c st)"
```

The state [Bot](#) gives [DEAD](#). Any other state is classified by [mcp_classify](#), which asks every selected analysis for the truth value of the check's condition through the queries of Section 7.1 and takes the meet of the answers: an exact 1 gives [PROVED](#), an exact 0 gives [REFUTED](#), and any other answer gives [UNKNOWN](#). A definite answer holds in every store the state describes ([mcp_classify_proved](#), [mcp_classify_refuted](#)).

The verdict of the CFG node soundly joins the per-context verdicts of its states ([aggregate_verdicts](#)). If every state other than [Bot](#) gives [PROVED](#), the node is [PROVED](#). If every such state gives [REFUTED](#), it is [REFUTED](#). If the states disagree, or one of them gives [UNKNOWN](#), the node is [UNKNOWN](#). Disagreement cannot be resolved otherwise. If one context proves $x > 0$ and another refutes it, the stores reaching v in the first satisfy the condition and those in the second violate it, so no single definite verdict holds for every store at the node. States classified [DEAD](#), the [Bot](#) states, are neutral in this aggregation, so a node at which the report holds no state, or only [Bot](#) states, is [DEAD](#).

The chain of Section 10.3 has to end in something an end user reads, and an end user reads the verdicts rather than the states. To state that the verdicts are correct as an inclusion of sets, the verdicts at v are turned into the set of stores consistent with them, the *verdict set* $\mathcal{V}_{\text{res } v}$. If v carries the check $x > 0$ with verdict [PROVED](#), the verdict set is $\{s \mid$

$s(x) > 0\}$: the verdict tells the user that x is positive there and nothing else. **REFUTED** gives the stores in which the condition fails, and **UNKNOWN** gives every store (**verdict_holds**). A **DEAD** check says nothing about the values in a store. It claims that no store arrives at v , which is a statement about reachability, and Section 10.5 treats it. As a set of stores, it therefore admits every store and leaves the claims of the other checks at v untouched:

```
1 (* src/Executable_Surface/CLI/Analysis_Report.thy *)
2 definition check_stores :: "result_check ⇒ store set" where
3   "check_stores c = (case check_verdict c of
4     Decided r ⇒ {s. verdict_holds r (check_exp c) s}
5     | Dead ⇒ UNIV)"
```

A store is consistent with what the user reads at v only if no verdict there contradicts it, so the verdict set keeps exactly the stores that satisfy every check at v :

```
1 (* src/Executable_Surface/CLI/Analysis_Report.thy *)
2 definition verdict_stores :: "analysis_report ⇒ pp ⇒ store set" ("V_") where
3   "V_res v = (⋂ c ∈ report_checks_at res v. check_stores c)"
```

The intersection states this for any number of checks at a node. In a compiled program, each check has a node of its own, and the verdict set is then the set of that single check.

The verdict set can be much larger than $\mathcal{R}_{\text{res}} v$, because a verdict keeps only the truth value of one condition (last row of Figure 10.1). Soundness needs only that every store reaching v lies in it.

Since the verdicts at v are computed from the states at v , every store that one of these states describes satisfies every definite verdict there. Thus $\mathcal{R}_{\text{res}} v \subseteq \mathcal{V}_{\text{res}} v$ (**analysis_report_verdicts_sound**) for every report that **run_voblint** returns (**run_voblint_consistent**).

10.3. Four inclusions from source execution to verdict

Let res be a report **run_voblint** returns for a program p , $g = \text{prog_cfg } p$ its compiled graph, $\mathcal{G} = \text{declared_global } p$ its global-variable classifier, and $S = \text{cinit_stores } \mathcal{G}$ its initial-store set (Section 3.6, Chapter 4). A store is *reached* if some finite source execution, started from a store in S , arrives at it. The argument that a verdict is sound places a reached store at a CFG node v and follows a chain of inclusions between sets of stores there. It ends in the two sets of the report, $\mathcal{R}_{\text{res}} v$ and $\mathcal{V}_{\text{res}} v$. Let s be a store that a source execution reaches, $\mathcal{G}, \Pi \vdash (\text{main_body } \Pi, s0, []) \rightarrow_p^* (r, s, \text{frs})$, and let v be a node at which the run is observed, $p, s0 \vdash (r, s, \text{frs}) \triangleright v$ (Section 3.5.2). Each step of the chain is a theorem:

$$\begin{array}{ccc}
 s & \xrightarrow[\text{source_reaches_node_collect}]{\in} & \mathcal{C}_{\mathcal{G}, g, s} v \\
 & \xrightarrow[\text{node_collect_eq_Union_activation_of_has_context}]{=} & \bigcup_c \mathcal{A}_{g, \text{adm}, c0, g, s} v c \\
 & \xrightarrow[\text{run_voblint_covers}]{\subseteq} & \mathcal{R}_{\text{res}} v \\
 & \xrightarrow[\text{analysis_report_verdicts_sound}]{\subseteq} & \mathcal{V}_{\text{res}} v
 \end{array}$$

The first step places a store that a finite source execution reaches in the node collecting semantics, at a node v that the simulation relates to the run. On the elaboration, this is every node at which the run is observed (`csim_point_compile_prog`, Section 3.5.2). The second step splits `node_collect` into `activation_collect` over the contexts `adm` admits, with `main` in c_0 (Section 4.3). It needs only that every valid trace carries some context. Functional policies meet this for every trace, and the entry-state policy meets it through `TOTAL` on the solution (Section 4.4). The third inclusion is the analyzer's soundness result. Its proof combines the second step with the per-context bound of Chapter 8, which places the activation collecting semantics of each context inside the state solved for it (`activation_collect_dg_sound`). The same bound holds for the report itself, context by context. For the policy `adm` and root context c_0 of the run, every node v and every context c satisfy

$$\mathcal{A}_{g, \text{adm}, c_0, g, S} v \ c \subseteq \text{report_context_sem } \text{res } v \ c$$

(`run_voblint_context_covers`). The fourth inclusion is that of Section 10.2 and depends only on the report and its consistency.

Consider the example below (Figure 10.1), whose program calls `f` twice. The first call passes 1. The second passes the input x if $1 \leq x \leq 3$. So m is 2, 4, or 6 at the checks: $m \geq 2$ always holds, and $m == 2$ holds at the first call and fails at the second call when x is 2 or 3.

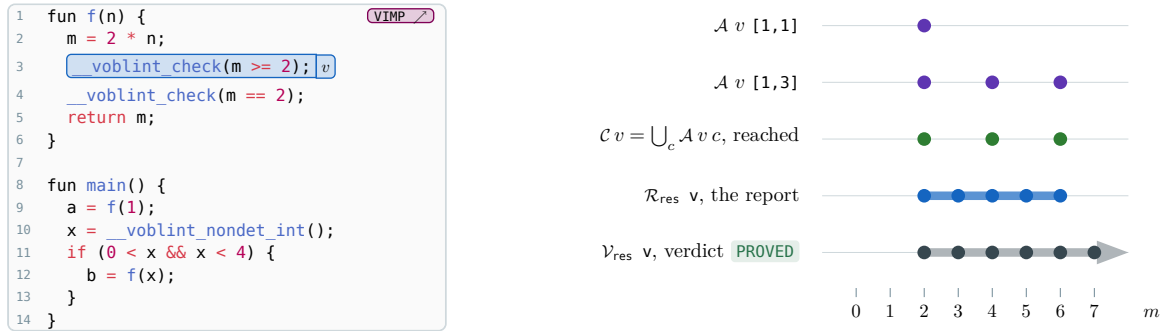


Figure 10.1: Left: the program. Right: the chain at the node v of the check $m \geq 2$ (line 3), projected on m , with Interval and entry-state contexts. Each row is a set of stores. Both per-context rows lie inside the union row, and each later row lies inside the rows below it. The report admits 3 and 5, which no run reaches, and the verdict set of `PROVED` is unbounded. The check $m == 2$ (line 4) sees the same stores. Report and verdict are analyzer output (claim `chain-split-entry`); the first three rows are derived by hand.

`f` is analyzed in two contexts, one per entry state. The check $m \geq 2$ holds for every admitted store in both, so its verdict is `PROVED`. The check $m == 2$ sees the same sets, but the contexts disagree. The first call enters with $n = 1$, and the check is `PROVED` there. The second enters with $n \in [1, 3]$. Its state $[2, 6]$ also admits 3 and 5, and the check is `UNKNOWN`, the only sound verdict, since the run with $x = 1$ satisfies it and the runs with $x = 2$ and $x = 3$ violate it.

10.4. The end-to-end soundness theorem `thy`

The chain of Section 10.3 argues about one store at one node, and each of its steps is a separate theorem about an intermediate set of stores. A user of the analyzer needs a single

statement instead, one that holds for every program and every analysis configuration. The end-to-end soundness theorem composes the four steps into such a statement. It quantifies over an arbitrary analysis configuration and program, the arguments of `run_voblint` (Section 10.1). Whenever the analyzer returns a report, every store that a finite source execution reaches is covered by that report and satisfies its definite verdicts. Of the sets in the chain, its conclusion keeps the node collecting semantics, $\mathcal{R}_{\text{res}} v$, and the verdict set. Its per-context counterpart is `run_voblint_context_covers` (Section 10.3).

The theorem is about `run_voblint` itself, the constant that code generation exports. It therefore concerns the analyzer that the command-line tool and the playground run, up to the trusted code generator and target toolchains (Section 12.2, Figure 1.4).

Theorem 10.1 (End-to-end soundness of the analyzer, `run_voblint_source_sound`).

```

1 (* src/Executable_Surface/CLI/Analysis_Certified.thy *)
2 theorem run_voblint_source_sound:
3   fixes p :: imp_prog and s0 s :: store
4   defines G_def: "G ≡ declared_global p"
5         and Pi_def: "Π ≡ prog_table p"
6         and g_def: "g ≡ prog_cfg p"
7   assumes s0: "s0 ∈ cinit_stores G"
8         and run: "G, Π ⊢ (main_body Π, s0, []) →p* (residual, s, frs)"
9         and ans: "run_voblint config p = Analysed res"
10  shows "(∃v. p, s0 ⊢ (residual, s, frs) ▷ v)
11         ∧ (∀v. p, s0 ⊢ (residual, s, frs) ▷ v →
12            s ∈ CG,g,cinit_stores G v
13            ∧ s ∈ Rres v
14            ∧ s ∈ Vres v)"

```

Here $\mathcal{G}$ and g are as in Section 10.3, and Π is the procedure table `prog_table p`. The theorem assumes three premises:

- (P1) $s0 \in \text{cinit_stores } \mathcal{G}$: the run starts from a store in `cinit_stores`, the initial stores S of Chapter 4, which zero every global, as C does for objects with static storage duration, and leave the locals unconstrained.
- (P2) $\mathcal{G}, \Pi \vdash (\text{main_body } \Pi, s0, []) \rightarrow_p^* (\text{residual}, s, \text{frs})$: a finite `pstep` execution of p runs from the body of `main` to a configuration with store s . Any stopping point is allowed, so nonterminating programs are covered through their prefixes.
- (P3) `run_voblint config p = Analysed res`: the analyzer returned a report. Input validity and solver termination are therefore not separate premises (Section 10.1), and the theorem is a partial-correctness result: termination is not proved for every program (Section 9.3).

The conclusions are stated at graph nodes, but a source configuration names none, since its residual command can occur at several places in the program. The node comes from the run, through the relation $p, s_0 \vdash (\text{residual}, s, \text{frs}) \triangleright v$ of Section 3.5.2 (`at_node`), which may observe one configuration at several nodes.

The theorem has two halves. The first states that the configuration is observed at some node. The second states that at every node v where it is observed

- (S1) $s \in C_{\mathcal{G},g,s} v$: the store is collected at that node;
- (S2) $s \in R_{\text{res}} v$: some state the report holds at v , under one of its contexts, describes s ;
- (S3) $s \in V_{\text{res}} v$: every `PROVED` check listed at v holds in s , and every `REFUTED` check is false in s .

10.5. What verdicts and diagnostics guarantee thy

By the theorem of Section 10.4, the four verdicts make the following statements about *covered executions*, the finite source executions from an initial store in `cinit_stores`, stopped at any point:

- **PROVED**: whenever a covered execution reaches the check, the condition holds (`run_voblnt_check_sound`).
- **REFUTED**: whenever a covered execution reaches the check, the condition fails (`run_voblnt_check_sound`). It is not a verified counterexample: it establishes no reaching execution.
- **UNKNOWN**: the analysis decides neither.
- **DEAD**: no covered execution reaches the check: nothing is collected at its node (`run_voblnt_dead_check_unreached`), and a check that an execution reaches is never **DEAD** (`run_voblnt_check_sound`).

PROVED and **REFUTED** hold vacuously when no execution reaches the check, and only **DEAD** claims anything about reachability. The corresponding theorem for **PROVED** and **REFUTED** is:

```

1 (* src/Executable_Surface/CLI/Analysis_Certified.thy *)
2 theorem run_voblnt_check_sound:
3   fixes p :: imp_prog and s0 s :: store
4   defines G_def: "G ≡ declared_global p"
5         and Pi_def: "Π ≡ prog_table p"
6         and g_def: "g ≡ prog_cfg p"
7   assumes s0: "s0 ∈ cinit_stores G"
8         and run: "G, Π ⊢ (main_body Π, s0, []) →p* (residual, s, frs)"
9         and chk: "next_check residual = Some (l, e)"
10        and ans: "run_voblnt_config p = Analysed res"
11 shows "∃ c ∈ set (report_checks res). check_label c = l ∧ check_exp c = e
12       ∧ s ∈ CG,g,cinit_stores G (check_point c)
13       ∧ check_verdict c ≠ Dead
14       ∧ (check_verdict c = Decided Check_Proved → truthy (⟦e⟧e s))
15       ∧ (check_verdict c = Decided Check_Refuted → ¬ truthy (⟦e⟧e s))"

```

The report also contains arithmetic diagnostics. At every division and remainder, the analyzer asks, through the queries of Section 7.1, whether the divisor can be zero. It reports an error if the analyses show that it is zero, a warning if they cannot exclude zero, and nothing otherwise. An error means that the divisor is zero whenever a covered execution reaches its node (`run_voblnt_arithmetic_refuted`). At a node v without a diagnostic, every divisor in an expression on an intraprocedural edge leaving v is nonzero in every store collected at v (`run_voblnt_arithmetic_safe` adds call arguments):

```

1 (* src/Executable_Surface/CLI/Analysis_Certified.thy *)
2 corollary run_voblnt_arithmetic_intra_safe:
3   assumes "run_voblnt_config p = Analysed res"
4   and "s ∈ Cdeclared_global p,prog_cfg p,cinit_stores (declared_global p) V"
5   and "∃ d ∈ set (report_diagnostics res). diagnostic_point d ≠ v"
6   and "(v, action, w) ∈ intra (prog_cfg p)"
7   and "e ∈ set (edge_expressions action)"
8   and "divisor ∈ expression_divisors e"
9 shows "⟦divisor⟧e s ≠ 0"

```

These theorems complete the chain of Figure 1.3. Each verdict holds for every covered execution that reaches its check, in every analysis configuration. Termination (Section 9.3) and the components outside the proof (Figure 1.4) are not covered.

PART V

Instances, Implementation, and Evaluation

11. Numeric and Relational Analyses

The preceding chapters stated what a domain, an analysis, and a combination of analyses must prove, and Chapter 10 showed that every report is sound, whichever analyses produced it. This chapter introduces and describes the analyses that Voblint implements. A user selects among the non-relational domains Sign, Interval, Parity, and Congruence, their reduced product Int, and a relational order analysis. The chapter compares the four base domains on one loop, then describes how Int lets them exchange facts and how the order analysis tracks relations between variables, which shows that the framework is not limited to one value per variable.

11.1. Four non-relational numeric domains

The four base domains, Sign [thy](#), Interval [thy](#), Parity [thy](#), and Congruence [thy](#), are the classical non-relational domains [29]; congruences go back to Granger [55]. The fifth domain, Int, combines all four. Its value holds one value of each base domain and denotes the intersection of their meanings (Section 11.2). All five are non-relational: they describe each variable separately, and the relational state of the order analysis follows in Section 11.3. Figure 11.1 shows one value of each and the integers it stands for.

domain	value	the integers it denotes																	
		-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	...
Sign	≥ 0	...																	...
Interval	$[-2, 5]$	...																	...
Parity	odd	...																	...
Congruence	$2+3\mathbb{Z}$	...																	...
Int	$+, [1, 9], 1+2\mathbb{Z}, 1+4\mathbb{Z}$	...																	...

Figure 11.1: One value of each non-relational domain, as the analyzer prints it, and the integers it denotes between -6 and 10 . Congruence values print as $c + m\mathbb{Z}$, the integers congruent to c modulo m ($m\mathbb{Z}$ when $c = 0$), and the Parity value shown is that of the odd integers. An end cell is filled when the set continues beyond the window. The values are illustrative. The Int value, the reduced state of x in the program of Section 11.2, denotes the intersection of its components' meanings, here $\{1, 5, 9\}$.

Each of these domains enters Voblint through the interface of Section 5.7. It instantiates the order classes for its carrier, supplies its operations as the record `nonrelational_ops`, and proves them sound once (`sound_nonrelational_ops`). From this proof, a generic construction derives the transfer functions of Chapter 6 together with their executable versions, and makes the domain a sound component of the product state of Chapter 7 (`sound_nonrelational_ops.dg_analysis_execI`). The only further obligation concerns the initial state, which must describe the initial stores.

The loop below is the fixture [05-mine_ex410_stride2_parity.vimp](#), adapted from Goblint’s regression test [56-witness/29-mine-tutorial-ex4.10.c](#) (revision d155e9e) after Example 4.10 of Miné’s tutorial [29]. In every run, v takes the odd values 1, 3, ..., 51 and the loop exits with $v = 51$, so every check holds.

```

1 fun main() {
2   v = 1;
3   while (v < 51) {
4     __voblnt_check(v >= 1);
5     v = v + 2;
6     __voblnt_check(v <= 52);
7   }
8   __voblnt_check(v >= 51); line 8
9   __voblnt_check(v <= 52);
10  __voblnt_check(v == 51); line 10
11 }

```

domain	records	state at the exit	$v \geq 51$ line 8	$v == 51$ line 10
Sign	sign of a value; finite	$v=+$	UNKNOWN	UNKNOWN
Interval	bounds; infinite ascending chains	$v=[51,52]$	PROVED	UNKNOWN
Parity	evenness	$v=\text{odd}$	UNKNOWN	UNKNOWN
Congruence	residue modulo m	$v=1+2\mathbb{Z}$	UNKNOWN	UNKNOWN
Int	all four, reduced	$v=51$	PROVED	PROVED

Table 11.1: The stride-2 loop under each domain (claims [dom-stride2-*](#)), at the two highlighted checks of the listing. Only Int, which combines the bound and the oddness, proves $v == 51$.

Each base domain records a different property of v . Interval finds the bound through widening and narrowing, but its exit state still allows 52, and an interval has no way to say “odd”. Parity and Congruence know that v is odd but nothing about its size, and Sign knows only that v is positive. Every answer is sound, yet none of the four proves $v == 51$. Only Int, which puts the bound and the oddness together, does (Section 11.2).

11.2. Combining the domains in a reduced product [thy](#)

An Int value ([int_dom](#)) holds one value of each base domain and stands for the integers that all four allow. Running the four side by side would be sound, but each would work only with its own value, so a fact that needs two of them would not be found. Table 11.1 shows this: no base domain alone proves $v == 51$. In the program below, the guards keep x between -2 and 10 , and the assignment makes x one more than a multiple of 4. Only 1, 5, and 9 satisfy both, so $x \geq 1 \ \&\& \ x \leq 9$ holds. Yet the interval alone still allows -2 , and the residue class alone allows -3 , so neither component can decide the check.

```

1 fun main() {
2   n = __voblnt_nondet_int();
3   x = 4 * n + 1;
4   if (x >= -2) {
5     if (x <= 10) {
6       __voblnt_check(x >= 1 && x <= 9);
7     }
8   }
9 }

```

A *reduction* lets the components sharpen each other, as in the reduced product [29, Def. 6.3, Ex. 6.2], [32]. Here the residue class moves the interval’s bounds to the nearest values that are 1 modulo 4, giving $[1, 9]$, and the now-positive interval makes the sign positive. The reduced value stands for the same integers but decides the check. Without reduction, the analyzer reports **UNKNOWN** there, and with it **PROVED** (claims [int-reduction-never](#) and [int-reduction-fixpoint](#)). Reduction also proves $v == 51$ in Table 11.1. Inside the loop, “odd” shrinks $[1, 50]$ to $[1, 49]$, so the loop head settles at $[1, 51]$ and only 51 remains after the loop.

Int thus combines its components differently from the cooperating analyses of Chapter 7. There, separate analyses keep separate states and exchange facts only through answers to queries, where a transfer asks one. Int is a single domain whose value holds all four components, and it reduces after every arithmetic operation and guard. A fact such as “odd” therefore sharpens the interval stored in the value itself, inside expression evaluation, and not just the answer to one query. Goblint draws the same line. Its integer domains form one value domain inside its base analysis, whereas its MCP combines whole analyses.

A reduction round passes information in two steps. It first combines the range information of Sign and Interval and uses it to sharpen Sign, Interval, and Parity. It then intersects the residue class with the parity and uses the result to sharpen Interval, Parity, and Congruence. The user chooses how often these rounds run: never, once, or until the value stops changing, which is the default. Every mode keeps the integers a value stands for and only makes the value smaller ([refine_exact](#), [refine_reductive](#)).

The design follows Goblint’s integer domain [IntDomTuple](#) at revision 5503dec. Goblint combines up to six optional components, each enabled by an option. Int always carries its four components, and Sign and Parity have no counterpart there. Goblint’s option `ana.int.refinement` offers the same three modes, but Goblint defaults to no reduction, whereas Voblnt defaults to repeating rounds until nothing changes.

In the generated OCaml, the default mode iterates without a termination proof, so on some inputs the analyzer might not return. Soundness is unaffected, since the theorem only speaks about returned reports (Figure 1.4).

Int reduces the result of every arithmetic operation and of the guard filter, but not the result of a join, a widening, or a narrowing, which all work componentwise. For join and widening, Goblint makes the same choice. After a narrowing, however, Goblint’s [IntDomTuple](#) performs a reduction that Voblnt cannot mirror. The verified TD solver requires the narrowing $a \Delta b$ of values $b \sqsubseteq a$ to lie between b and a (Section 2.5). Reduction only makes a value smaller, so it can push the narrowed value below b and break this law ([post_narrow_refinement_would_violate_narrow_ge](#)). Goblint is not bound by the law, because no proof constrains its lattices, and its answers may thus be more precise.

11.3. Order constraints as a relational state thy

None of the earlier domains can express that $x \leq y$, because each describes the variables one at a time (Figure 5.2). A relational state records such relations, so its concretization maps the whole state to a set of stores at once. A value of type `relc` is a set P of variable pairs, where (x, y) stands for $x \leq y$, and describes the stores that satisfy every pair (`gamma_relc`):

$$\gamma(\text{RelC}(P)) = \{s \mid \forall (x, y) \in P. s(x) \leq s(y)\}, \quad \gamma(\text{RelBot}) = \emptyset.$$

Below, the pair (x, y) is written $x \leq y$.

More pairs describe fewer stores, so the order is reverse inclusion: $\text{RelC}(P) \sqsubseteq \text{RelC}(Q)$ exactly when $Q \subseteq P$ (`less_eq_relc`), with `RelBot` below every value and $\top = \text{RelC}(\emptyset)$, which constrains nothing. The join keeps the pairs both operands share, $\text{RelC}(P) \sqcup \text{RelC}(Q) = \text{RelC}(P \cap Q)$ (`sup_relc`).

The carrier does not close its pairs under transitivity (Figure 11.2). The values $\{x \leq y, y \leq z\}$ and $\{x \leq y, y \leq z, x \leq z\}$ therefore denote the same stores but are different values. The join can also forget an implied relation. The join of $\{x \leq y, y \leq z\}$ and $\{x \leq y, x \leq z\}$ keeps only their common pair and is $\{x \leq y\}$, although both operands imply $x \leq z$. The order analysis is therefore not as precise as a closed relational domain could be. It demonstrates that the framework supports relational states.

Pairs enter only at guards, and an assignment to x removes every pair that mentions x , so P is finite in every value the analysis constructs, and generated code can store it as a list.

The emptiness test only compares with `RelBot` (`is_empty_relc_gamma`). Unlike the pointwise state, the carrier has a single representation of unreachability, so it needs no lifting.

A relational state stores no value per variable, so it cannot evaluate an expression to an abstract value, and its transfers work on the pairs directly. An assignment to x invalidates

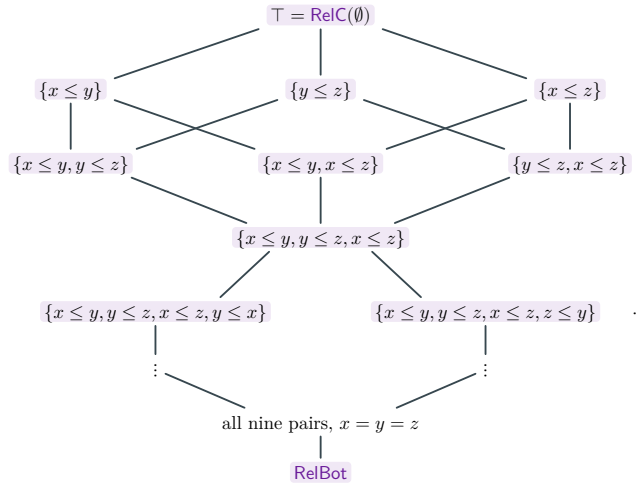


Figure 11.2: The relational states over the variables x , y , and z , ordered bottom to top by reverse inclusion. Each `RelC` value is drawn as its set of pairs. The upper part shows all values built from $x \leq y$, $y \leq z$ and $x \leq z$. Values with other pairs, such as $y \leq x$, are drawn only below the lowest of these, and the dots stand for the rest. Nine pairs exist, including $x \leq x$, so the values are finitely many, and the lowest `RelC` value holds all nine and denotes the stores with $x = y = z$.

every relation that involves x , so it forgets every pair that mentions x . New relations come from guards. On the true arm of `if (x < y)`, the analysis knows that x is smaller than y and records this by meeting its state with `RelC{x ≤ y}` (`assume_step`). Since the carrier stores only non-strict pairs, it keeps $x \leq y$ and loses the strictness, which costs precision but not soundness. This guard filter meets the same soundness statement as the numeric guard filter of Section 5.6 (`relc_branch_step_sound`). The type instantiates the solver's classes and `executable_domain` of Figure 5.6, but it is not a numeric domain.

11.4. Proving order relations between variables thy

Section 11.3 introduced the relational state as an abstract domain. This section builds the order analysis on top of it and demonstrates it on a program. After an assignment has forgotten the pairs involving the changed variable, it asks the shared channel how the new value relates to each variable, and every selected analysis, the order analysis included, may give an answer (Section 7.3). In the program below, the guard `x < y` tells Interval nothing about either variable, so it keeps $x = \top$, $y = \top$ and reports `UNKNOWN` at the first check. The order analysis records $\{x \leq y\}$ and reports `PROVED`.

```

1 fun main() {
2   x = __voblint_nondet_int();
3   y = __voblint_nondet_int();
4   z = __voblint_nondet_int();
5   if (x < y) {
6     __voblint_check(x <= y);
7     if (y <= z) {
8       __voblint_check(x <= z);
9     }
10    y = y + 1;
11    __voblint_check(x <= y);
12  }
13  w = x;
14  __voblint_check(w == x);
15 }
```

Because the analysis does not combine pairs transitively, it cannot prove $x \leq z$ under $y \leq z$. It holds $\{x \leq y \wedge y \leq z\}$ there and answers `UNKNOWN` (Section 11.3). The assignment `y = y + 1` throws away every pair involving y , leaving $\top$. A copy, in contrast, yields pairs in both directions. At `w = x`, the assignment asks, for the variable x itself, whether $x \leq x$ holds in each direction. The order analysis answers yes to both, because it treats a variable as ordered with itself, so it records $\{w \leq x \wedge x \leq w\}$ and proves `w == x`, where Interval answers `UNKNOWN`.

The base domains record different properties, Int lets them share these facts, and the order analysis proves relations that no non-relational domain can express. A user does not have to choose between them. The product state and the channel of Chapter 7 let Int and the order analysis run together. They exchange facts through queries, and every check is decided by the meet of their answers. Selecting both gets the most out of the analyses Voblint ships, although no theorem guarantees that a combination is more precise than its parts (Section 7.3).

12. The Executable Analyzer

Chapter 10 proved that every report `run_voblint` returns is sound. A user, however, does not evaluate a HOL function. The user writes source text and reads a report in a terminal or a browser tab. Between the two, the proved function has to become a program. Its abstract states are mathematical functions, which code cannot compute with, so they are first replaced by finite representations (Section 12.1). Isabelle’s code generator then exports `run_voblint` to OCaml (Section 12.2), and an unverified shell of handwritten OCaml parses the source text and displays the report (Section 12.4). The change of representation is proved to preserve the theorem. The code generator and the shell are not proved correct, so the guarantee for the running tool also rests on them, as Figure 1.4 shows.

12.1. Finite representations of abstract states

To export `run_voblint` as code, every function it calls must be computable. Most of the development already is, since its datatypes and recursive functions translate directly into a functional program, but the concretizations and the pointwise states do not. The concretization γ maps an abstract value to a possibly infinite set of integers, and in general no program can compute such a set. The analyzer does not need to, because γ only states what a value means. The domain interface therefore separates `executable_domain` from the classes and locales below it in Figure 5.6, which add γ and its laws. Code generation exports the executable part together with each domain’s operation record, and the rest serves only the proofs.

The pointwise states of Chapter 5 are total functions on infinitely many variable names, so the solver could not even test whether two of them are equal. Following Concrete Semantics [6, §13.6], an *executable state* (`default_st`) instead stores a default value and a finite list of overrides. Voblint keeps two such pairs, one for locals with default $\top$ and one for globals with default $0^\sharp$, because VIMP initializes globals to zero as C does for static storage (§6.7.9p10). After `x = 0` in `main`, for example, the local part holds the default $\top$ and the single override $x \mapsto [0, 0]$, and every other local still reads $\top$. Since several lists describe the same state, a quotient type [56] identifies the lists that answer every lookup alike. Order and equality then depend only on the defaults and the finitely many listed names, and every operation is computed on exactly these. Each operation therefore agrees with its counterpart on pointwise states, and the soundness results of Chapter 6 carry over to the code that runs (`exec_local_spec_sound`).

With concretizations left to the proofs and pointwise states replaced, nothing else stands in the way. The verified TD solver already comes with an executable version, `solve_c`, which Tilscher et al. refine from its definition [4], and Section 9.3 shows that every run of it that returns satisfies the solver’s correctness theorem. The values of the numeric domains (Section 11.1) and the states of the order analysis (Section 11.3) are already finite data.

12.2. Exporting the analyzer to OCaml and WebAssembly [thy](#)

With these replacements, every function that `run_voblint` uses has code equations, and Isabelle’s code generator [26] can export it. One `export_code` declaration emits `run_voblint`, `render_report`, and the constructors and selectors a caller needs into one OCaml file, [Voblint Generated.ml](#). The command-line tool and the browser playground both call this module through the facade [cli/voblint.ml](#), which re-exports the generated module unchanged. Every analysis result is therefore computed by generated code.

The generated module has 13,109 lines and follows the proofs rather than the habits of OCaml programmers, which makes it hard to read. Type classes become records of operations passed as arguments, finite sets and maps become lists that every lookup scans, and the solver keeps its valuation as a chain of closures, one per update.

The code that runs is a small part of the development (Table 12.1). Code generation keeps only the definitions that `run_voblint` depends on and drops every proof, so the theories and the imported part of the vendored TD solver shrink to the generated module. The handwritten OCaml around this module is smaller still. Together with the playground’s JavaScript, which draws the answer, it is the code that has to be checked by hand, while Zarith, the code generator, the compilers and the browser are trusted (Section 12.4).

part	size	status
generated OCaml analyzer	13,109 lines	generated from the proved <code>run_voblint</code>
Isabelle theories	59,477 lines	proved, 213 theories
vendored TD solver, imported part	9,533 lines	proved by Tilscher et al. [4]
handwritten OCaml	4,922 lines	outside the proof (Section 12.4)
regression programs	386 programs	tested, not proved (Section 12.5)

Table 12.1: The parts of the development by size and status. Line counts are measured on the repository at build time.

The proofs say nothing about running time, and the generated code can be inefficient. In one case, a sampling profile of 256 chained assignments attributed most of the CPU time to a single function. The generated code of `init_publications` recompiled the whole program at every equation evaluation, because its definition names the program’s entry through the compiler from VIMP to control-flow graphs (Section 3.4.1). A proved code equation, `init_publications_code`, that names the entry directly as `FunctionEntry prog_main_name` fixed this without changing any theorem, and the chain of 256 assignments then ran 13 times faster and a program of 512 assignments in procedures of eight statements 18 times faster, in one measurement each ([recorded](#)). Finding further inefficiencies and improving the generated code is future work.

The command-line tool links the module with the handwritten OCaml into a native program. For the playground, `wasm_of_ocaml` compiles the same code to WebAssembly, which a web worker in the reader’s browser calls (Figure 12.1). Both tools thus run the generated code of `run_voblint` up to the code generator and the compilers (Figure 1.4).

A browser worker, however, has a much smaller stack than a native program. Deep recursions such as the one in [04-function_chain_self_recursive_tail](#) can therefore overflow it. The bundle attempts to avoid this with the continuation-passing translation of `wasm_of_ocaml` (`--effects=cps`). With it, the browser build needs $3.3\times$ the native solve time on the median run (Section 13.5, [profiling page](#)).

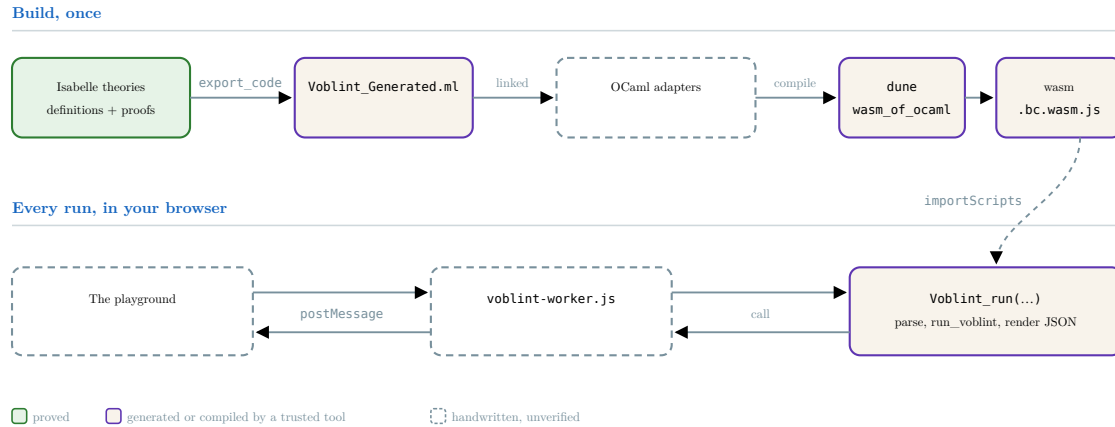


Figure 12.1: How the generated `run_voblint` reaches the browser. Build: `export_code` writes `run_voblint` into `Voblint_Generated.ml`, which `dune` and `wasm_of_ocaml` compile with the handwritten OCaml into a WebAssembly bundle. Run: a web worker calls the bundle and returns the answer as JSON. Adapted from the project site.

12.3. Tracing solver steps without changing the proofs [thy](#)

To explain a run, the command-line tool and the playground can trace the solver's steps. The solver trace records when each unknown is queried, which update destabilizes which unknown, and where widening applies. Goblin's tracing of its top-down solvers reports the same events. Tracing must leave the definitions and proofs of the vendored TD solver [4] untouched. Voblint therefore adds tracing only to the equations that code generation translates. Every traced equation reports each step through one constant, `trace_event`, which is defined as the unit value `()` and so does nothing in the logic. Each traced equation is thus provably equal to the vendored one it copies (for example, `solve_rec_c_traced`). The traced copies make up a second entry point, `run_voblint_traced`, which equals `run_voblint` in the logic (`run_voblint_traced_eq`). Code generation exports both. A `code_printing` declaration then tells the code generator to emit, in place of `trace_event`, a call to the handwritten OCaml function `Solver_trace_hook.emit`. The hook receives a suspended event and builds it only when tracing is on. Only the traced entry point calls the hook, so an untraced run executes no handwritten code inside the analysis. For a traced run, the hook joins the trusted base and must return, raise nothing, and leave the solver's values alone.

The command-line tool prints the events with `--trace`, and the playground replays them step by step on the solved graph (Figure 12.3b). The replay shows the order of the solve, which the solved values alone hide, such as when a call creates a new callee context. The solver trace itself is unverified, but a traced run returns the same report as an untraced one, provided the hook meets its obligations.

12.4. Unverified parsing, rendering, and playground

The generated `run_voblnt` works on a syntax tree and returns a typed answer, whereas a user writes source text and reads a report. Unverified OCaml bridges both ends. On the input side, an `ocamllex` lexer and a Menhir parser turn the source text into an `imp_prog` and record each check's source position in its label. Both are generated from the grammar manifest `vimp-grammar.yaml` by `gen_vimp_menhir.py`. On the output side, the generated `render_report` turns the report's states into text, and handwritten OCaml either prints the command-line report, placing each check at the position its label carries, or serializes the answer as JSON, from which the playground draws the graph and marks the verdicts and diagnostics in the editor. The command-line tool and the playground share all of this code and differ only in how they receive a request. The code generator maps HOL's integers to arbitrary-precision Zarith integers, as VIMP requires (Section 3.6).

Figure 12.2 shows, for the program below, the call that `run_voblnt` receives and the answer it returns once `render_report` has turned the states into text, serialized as for the playground.

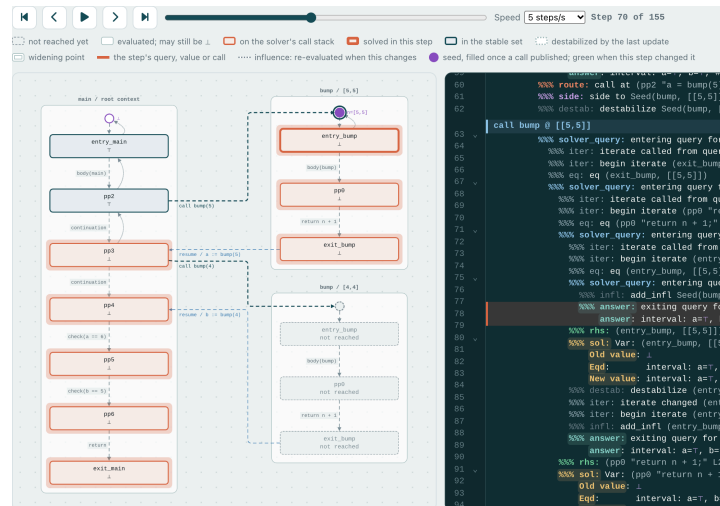


Figure 12.2: The input and output of the generated analyzer for the program above (claim [shell-json](#)). The call carries the configuration and the syntax tree `p` the parser built, of the kind Figure 3.3 shows in full; the answer carries the report with its states rendered as text, here with its one check in full. Everything before the call, and the rendering from the answer on, is unverified. `{...}` and `[...x $n$ ]` fold parts of the JSON.

The [playground](#) runs the analyzer in the browser. Its toolbar sets the configuration of `run_voblnt` and toggles the tracing of Section 12.3, and the end-to-end soundness theorem covers every configuration it can select (`run_voblnt_source_sound`). Figure 12.3a shows one run.



(a) one run in three views: the check's verdict in the editor, the state inspector at the check, and the solved graph. Settings analysis interval, globals warrow, context none [open this run](#).



(b) one step of the solver replay for the calls of `bump`: the graph shows each unknown's value at this step, and the solver trace beside it marks the step's line. Settings analysis interval, globals warrow, context entry-state [open this run](#).

Figure 12.3: The playground. Both panes run the generated analyzer of Figure 12.1 in the reader's browser. The screenshots are illustrative.

12.5. Testing the unverified components

The handwritten OCaml around the generated analyzer (Table 12.1) is tested. A [regression corpus](#) of 386 VIMP fixtures pins the expected verdicts: a definite one in `precision/`, **UNKNOWN** in `soundness/`, and a documented loss in `known-imprecision/`. [Golden files](#) pin solver traces (Section 12.3), and property tests check that the parser reads back random syntax trees.

13. Evaluation of Strength and Precision

Chapter 10 proved that every report `run_voblint` returns is sound. A machine-checked theorem can still hold for uninteresting reasons. Its premises may never hold together, its obligations may exclude no error a real analyzer makes, and an analyzer that answers `UNKNOWN` everywhere satisfies it as well (`unknown_everywhere_sound`). Following J. Kim [57, §6], this chapter first checks that the premises can hold and that the obligations exclude real errors. It then shows, on concrete programs, what precision the analyzer reaches and where it ends. Several facts below concern one fixed program and are proved by `eval`, which runs generated code and therefore also trusts the code generator (Figure 1.4).

13.1. The soundness theorems are not vacuous `thy`

A theorem whose premises can never hold together is true but says nothing. The end-to-end soundness theorem has three premises: an initial store, a source execution from it, and an `Analysed` answer. The last one needs care, because the analyzer answers only if its solve terminates, and termination is not proved in general (Section 9.3). The direct way to show that the premises are consistent is to exhibit a program that meets all of them, as D. Marmosier [58, §9] does for the locale assumptions of a formal model.

Here, the two-call program of Figure 3.1 is analyzed with Interval, entry-state contexts, and the *warrow* update rule. The theory builds a source execution that returns from `bump(5)` and `bump(4)` and stops before the first check. Evaluation shows that the analyzer answers and that both checks are `PROVED`. With these facts, `nv_source_certified` instantiates `run_voblint_source_sound` with every premise discharged. Its counterpart for single checks, `nv_check_proved_sound`, does the same for `run_voblint_check_sound` at the check `a == 6`.

A `DEAD` verdict needs its own instance, since its theorem concludes that no run reaches a node rather than that a condition holds there. A second program sets `x = 1` and guards a check by `x < 0`. The analyzer marks the check `DEAD`, and `nv_dead_unreached` instantiates `run_voblint_dead_check_unreached` to show that the collecting semantics at that node is empty. In all three instances, the source executions are proved by rule application, while `eval` computes the compiled graphs and the analyzer’s answers. The end-to-end soundness theorem and its companions therefore do not hold vacuously, since each applies to at least one program on which the solve actually terminates.

13.2. The obligations reject a real Goblint bug

Non-vacuity does not show that the obligations are strong enough to reject a wrong analysis. One way to test them is to check them against a real error. Chapter 1 opened with [issue 1156](#), in which Goblint claimed `c % 2 == 1` for $c \in \{-5, -7\}$, although C’s truncating remainder gives `-1` for both values. [Pull request 1161](#) restricted when the

congruence domain returns a constant remainder and added the program as regression test 37-congruence/14-negative.c [14].

In Voblint, the defective operator could not have been proved sound. Every Congruence remainder must meet `congruence_mod_sound`, which requires the remainder `c_mod` of any two concrete values to lie in the concretization of the abstract remainder. Since `c_mod` truncates toward zero, as C11 specifies (Section 3.6), the pre-fix answer, the constant 1 for $1 + 2\mathbb{Z} \bmod 2$, violates the obligation at -5 (`prefix_congruence_mod_unsound`). This theorem covers every abstract remainder that returns the constant, so it also rules out Goblint’s pre-fix operator. Figure 13.1 runs the regression test transliterated into VIMP. Congruence alone knows only $c \in \gamma(1 + 2\mathbb{Z})$ and decides neither check. Int also knows that c lies in $\gamma([-7, -5])$, so it refutes the first check and proves the second.

check	Congruence	Int
<pre> 1 fun main() { 2 top = __voblint_nondet_int(); 3 c = -5; 4 if (top) { 5 c = -7; 6 } 7 __voblint_check(c % 2 == 1); 8 __voblint_check(c % 2 == -1); 9 } </pre>	UNKNOWN $c = 1 + 2\mathbb{Z}$	REFUTED $c = \text{signs: } -$ $\text{intervals: } [-7, -5]$ parities: odd $\text{congruences: } 1 + 2\mathbb{Z}$
		PROVED $c = \text{signs: } -$ $\text{intervals: } [-7, -5]$ parities: odd $\text{congruences: } 1 + 2\mathbb{Z}$

Figure 13.1: Goblint’s regression test for [pull request 1161](#) in VIMP, and the verdict and state at each check with Congruence alone and with Int, both under the unit context policy. Concretely, $c \% 2$ evaluates to -1 on both executions.

13.3. Measuring precision on concrete programs

The theorem says nothing about precision, so the examples below measure it on single programs whose concrete behavior is known.

Earlier chapters already showed precise answers. Narrowing recovers the bound of the counting loop after widening (Section 2.5), entry-state contexts keep the two calls of `bump` apart (Figure 8.3), and only Int proves $v == 51$ on the stride-2 loop (Table 11.1). Analyses that cooperate through queries decide checks that each alone leaves unknown (Chapter 7, `order_asks_needs_both`). Beyond these, the corpus holds 261 fixtures that pin a definite verdict (Section 12.5). Searching the [playground](#)’s examples for “precision” lists them.

13.3.1. Precision and cost of calling contexts

Contexts trade analysis time for precision. In the program of Figure 13.2, `f` passes its argument on to `g`, and `main` calls `f` with a positive and a negative value. With call strings of length 1, a context of `g` records only its call site in `f`, so both activations of `g` share one context. Sign joins `+` and `-` there to `T`, and neither check is proved. Call strings of length 2 also record the call in `main`, keep the two activations apart, and prove both checks. The price is a second copy of `g`. The theories prove the same separation for `run_voblint` itself (`run_voblint_call_string_2_decides`, `cs_nest_checks_hold`) and at the entry of `g` on hand-built equation systems (`sign_k2_strictly_more_precise_than_k1_at_g`).

```

1 fun g(p) { return p + p; } VIMP
2 fun f(p) { t = g(p); return t; }
3 fun main() {
4   x = f(3);
5   y = f(-10);
6   __voblnt_check(0 < x);
7   __voblnt_check(y < 0);
8 }

```

check	length 1	length 2
$0 < x$	UNKNOWN $x = \top$	PROVED $x = +$
$y < 0$	UNKNOWN $y = \top$	PROVED $y = -$

Figure 13.2: Sign with call strings of length 1 and 2 (claims [ctx-nest-k1](#) and [ctx-nest-k2](#)). Concretely, $x = 6$ and $y = -20$.

More contexts also cost more, since every context of a procedure gets its own copy of the procedure’s unknowns. The program `down(100)`, which recurses down to `down(0)` and checks $n \geq 0$ at every level, shows that this cost does not buy precision gradually. With call strings of length 100, every level gets its own context, and the check is **PROVED** with 102 procedure copies. With length 99, the analyzer still builds 101 copies, but the deepest calls share one context. Widening there loses the lower bound of n , and the check is **UNKNOWN** (claims [cost-down-k99](#) and [cost-down-k100](#)). Nearly the whole cost is thus paid before any precision is gained.

With entry-state contexts, the number of contexts can even be infinite. Under the unit context policy and bounded call strings, a compiled program has finitely many unknowns ([compiled_unit_vars_finite](#), [compiled_call_strings_finite](#)). Entry-state contexts under Interval have no such bound. A recursion that changes its argument at every level demands a fresh context at every level, and widening bounds the values of unknowns but not their number ([01-unbounded context chain diverges](#)). The solve exceeds its time limit, and it is expected, without proof, to diverge. Goblin’s configuration for the SV-COMP competition avoids this explosion by omitting intervals from the contexts of recursive functions [27, §3.4.3].

13.3.2. Comparing the update rules for global unknowns

Table 13.1 runs the five update rules of Section 9.2 on four programs. Every check holds in every execution, so each **UNKNOWN** in the table is lost precision. The first two rules only join contributions to a global unknown, and the last three also widen them.

In the first row, `p(1)` and `p(2)` publish to the same seed. *Warrow* joins the two contributions before widening, so the second looks like growth, and the upper bound of x jumps to $+\infty$. The per-origin rules widen each call site’s contribution separately, and since each site writes one constant, nothing grows. The joining rules never widen and keep `[1, 2]` as well. The call in a loop shows the same pattern, and again, only *warrow* loses the bound.

The two recursions separate the joining rules from the warrowing ones. On the growing recursion, the argument grows without bound, and the joining rules give no answer within the limit. Their solve is expected to diverge, although the timeout alone does not show this. The warrowing rules stop at `[0,  $+\infty$ ]`, which suffices for $x \geq 0$. On the shrinking recursion, in contrast, *warrow per origin* widens the recursive call’s own contribution to $-\infty$, so `a + 2` may be zero and the division loses all information. *Warrow* keeps the lower bound, which the author attributes to its joining in `main`’s argument `-1` first, and the joining rules keep `[-1, 9]`. No rule is therefore best on every row. *Bounded narrowing*, at its default bound 5, matches *warrow per origin* on all four programs.

program	join	join per origin	warrior	warrior per origin	bounded narrowing
two call sites: $p(1); p(2);$, check $x \leq 2$ in p	PROVED VIMP ↗ $x = [1, 2]$	PROVED VIMP ↗ $x = [1, 2]$	UNKNOWN VIMP ↗ $x = [1, +\infty]$	PROVED VIMP ↗ $x = [1, 2]$	PROVED VIMP ↗ $x = [1, 2]$
call in a loop: $g(x)$ for x from 0 to 9, check $x < 10$ in g	PROVED VIMP ↗ $x = [0, 9]$	PROVED VIMP ↗ $x = [0, 9]$	UNKNOWN VIMP ↗ $x = [0, +\infty]$	PROVED VIMP ↗ $x = [0, 9]$	PROVED VIMP ↗ $x = [0, 9]$
growing recursion: $f(x)$ calls $f(x + 1)$ from $f(0)$, check $x \geq 0$	no answer in 5 s VIMP ↗	no answer in 5 s VIMP ↗	PROVED VIMP ↗ $x = [0, +\infty]$	PROVED VIMP ↗ $x = [0, +\infty]$	PROVED VIMP ↗ $x = [0, +\infty]$
shrinking recursion: $f(a)$ calls $f(9 / (a + 2))$ while $a < 4$, from $f(-1)$, check $a \leq 9$	PROVED VIMP ↗ $a = [-1, 9]$	PROVED VIMP ↗ $a = [-1, 9]$	PROVED VIMP ↗ $a = [-1, 9]$	UNKNOWN VIMP ↗ $a = \top$	UNKNOWN VIMP ↗ $a = \top$

Table 13.1: Update rules on four programs (Interval, unit context policy, bounded narrowing at bound 5), as on the [project site](#). Each cell gives the verdict and state at the row’s check, and its button opens the run. “No answer in 5 s”: stopped by --timeout 5.

13.4. Comparing precision with Goblint

The programs of Section 13.3 were selected because a mechanism succeeds on them, which biases them toward Voblint’s strengths. Its limits are better measured on programs selected by someone else. Monniaux’s tutorial on the flavors of abstract interpretation walks through finite lattices, relational domains, widening and narrowing, and alternatives to iterative solving. For each, it gives a small program on which a standard analysis gains or loses precision [59]. Every program of the tutorial that VIMP can express is transliterated into a regression fixture of [28-flavors](#), and Voblint’s verdicts on them are registered as claims. Each check holds in every execution, so each UNKNOWN below is lost precision.

For comparison, Goblint at revision [6127869](#) ran on C versions of the same programs (Table 13.2). VIMP’s integers are unbounded (Section 3.6), whereas Goblint analyzes C’s `int`. On these programs, the difference does not change what the programs compute, since no execution leaves the range of `int`. It can still change an analysis result, because an

	finite lattices		paths and convexity	widening and narrowing			narrowing foiled	
	rule of signs VIMP ↗ $z < 0$	not inductive VIMP ↗ $i > 0$	absolute value VIMP ↗ $x \neq 0$	bounded loop VIMP ↗ $i == 10$	reset at bound VIMP ↗ $i \leq 9$	reset at bound VIMP ↗ $i \geq 0$	skipped increment VIMP ↗ $i \leq 9$	reset at equality VIMP ↗ $i \leq 9$
Voblint	Sign	PROVED	UNKNOWN	UNKNOWN	UNKNOWN	PROVED	UNKNOWN	UNKNOWN
	Interval	PROVED	UNKNOWN	PROVED	PROVED	PROVED	UNKNOWN	UNKNOWN
	Int	PROVED	PROVED	PROVED	PROVED	PROVED	UNKNOWN	UNKNOWN
	Int, Order	PROVED	PROVED	PROVED	PROVED	PROVED	UNKNOWN	UNKNOWN
Goblint	Interval	PROVED	UNKNOWN	PROVED	PROVED	UNKNOWN	UNKNOWN	UNKNOWN
	delay 10	PROVED	UNKNOWN	PROVED	PROVED	PROVED	PROVED	PROVED
	Int, delay 3	PROVED	PROVED	PROVED	PROVED	PROVED	UNKNOWN	UNKNOWN
	thresholds	PROVED	UNKNOWN	PROVED	PROVED	PROVED	PROVED	PROVED
	comparisons	PROVED	UNKNOWN	PROVED	PROVED	PROVED	UNKNOWN	UNKNOWN
	octagons	PROVED	UNKNOWN	PROVED	PROVED	UNKNOWN	UNKNOWN	UNKNOWN
	octagons, paths	PROVED	UNKNOWN	PROVED	PROVED	UNKNOWN	UNKNOWN	UNKNOWN

Table 13.2: The flavors programs [59] under each analysis setting of Voblint ([28-flavors](#)) and of Goblint ([C versions](#), [results.tsv](#)). Each VIMP button opens the program in the playground. Goblint’s “will succeed” is shown as PROVED, “is unknown” as UNKNOWN.

abstract value may reach the limit of `int` although no execution does. Goblint then applies its default setting `assume_top` of [sem.int.signed_overflow](#), which gives a signed operation that may overflow the whole range of its type. This happens for one check, discussed in Section 13.4.3.

The settings of Table 13.2 are chosen such that each analyzer has its best chance on each flavor. Voblint runs Sign, Interval, Int, the reduced product of its four non-relational domains (Section 11.2), and Int with the order analysis, its only relational information. No other Voblint option affects these programs. Its solver has neither widening delays nor thresholds, and calling contexts and the update rules for global unknowns need procedures, which a second benchmark compares (Table 13.3). The Goblint rows “delay 10”, “thresholds”, and “comparisons” apply the repairs the literature proposes for failed narrowing, and 10 is the smallest delay that proves a check Interval leaves unknown. In “Int, delay 3”, Goblint’s closest counterpart of Voblint’s Int, Interval and Congruence refine each other. The last two rows add Apron’s octagons [60], the second together with Goblint’s [branchSet](#) analysis, which keeps a separate state for each set of branches taken.

The following subsections discuss the column groups of Table 13.2 from left to right.

13.4.1. Finite lattices

The tutorial opens with the rule of signs, which says that the sum of two negative numbers is negative [59, slide 8]. Over unbounded integers, the rule is sound, and Sign proves $z < 0$ after $z = x + y$ with $x = -5$ and $y = -10$ (claim [flavors-vanilla-sign](#)).

A set of states at a loop head is an *inductive* invariant if it contains the states that first reach the head, and if one more iteration of the loop from any of its states stays inside it. In the terms of Section 2.2, it is a post-fixpoint of the loop’s transfer function that contains the initial states. The abstract value that an analyzer computes at a loop head describes such an invariant, so a check that the analyzer proves follows from an inductive invariant that its domain can express. The converse fails: widening may miss an invariant the domain can express (Section 13.4.4).

```

1 fun main() {
2   i = 6;
3   i = i - 2;
4   while (i != 0) {
5     __voblint_check(i > 0);
6     i = i - 2;
7   }
8 }

```

In this loop from the tutorial [59, slide 27], i takes the values 4 and 2 at the check, so $i > 0$ holds. The property is not inductive, however. If all that is known at the loop head is that i is positive, i may be 1, and the next iteration then yields -1 . Sign can express $i > 0$ but nothing stronger about i , so the loop head reaches every sign, and the check is **UNKNOWN** (claim [flavors-not-inductive-sign](#)). Int, in contrast, can express a stronger property that is inductive: at the loop head, i is nonnegative, at most 4, and even. A positive i with these bounds is 2 or 4, so the next iteration yields 0 or 2 and stays inside the property.

Int finds this invariant through the reduction of its components. Widening drops Interval’s lower bound at the loop head, but Sign keeps $i \geq 0$. The guard $i \neq 0$ then makes the sign positive, and the reduction (Section 11.2) moves the interval to $[1, 4]$ and,

since i is even, to $[2, 4]$. From there, $i - 2$ returns to $[0, 2]$, which the head value already covers, and the check is **PROVED**. Without the reduction, the check sees $i \in [-\infty, 4]$ and is **UNKNOWN** (claims [flavors-not-inductive-int](#) and [flavors-not-inductive-int-never](#)). Goblint proves the check only with a widening delay of 3 or with thresholds at the constants of comparisons (Table 13.2, [results.tsv](#)).

13.4.2. Paths and convexity

The tutorial’s coffee example computes an absolute value and checks that it is positive only for a nonzero argument [59, slide 38]:

```

1 fun main() {
2   x = __voblnt_nondet_int();
3   y = 0;
4   if (-1000 <= x && x <= 2000) {
5     if (x < 0) {
6       y = -x;
7     } else {
8       y = x;
9     }
10    if (y >= 1) {
11      __voblnt_check(x != 0);
12    }
13  }
14 }

```

The check needs $y \leq |x|$ after the branch, which holds only as the disjunction of $y = x$ on one path and $y = -x$ on the other. A non-relational domain loses both relations. A convex relational domain joins the branches into their convex hull [61, §4.4], which keeps $y \geq |x|$ but loses $y \leq |x|$, so polyhedra fail as well [59, slides 42–43]. Voblnt’s order analysis ends with $x \leq y$, which does not exclude $x = 0$, and the check is **UNKNOWN** (claim [flavors-coffee-int-order](#)). Proving the check requires keeping the two branches apart, as trace partitioning and path partitioning do [29, §6.3.4], [62, §1.1], [63, §7.1.5]. In Goblint, neither octagons nor path sensitivity alone proves the check, but octagons partitioned by the branches taken do. Voblnt has no such partitioning, which would need its own soundness proof.

13.4.3. Widening and narrowing

Voblnt’s Interval proves both widening examples that narrowing repairs: the bounded loop, which counts i to 10 (claim [flavors-fudge-interval](#)), and the reset at bound:

```

1 fun main() {
2   i = 0;
3   while (true) {
4     i = i + 1;
5     if (i >= 10) {
6       i = 0;
7     }
8     __voblnt_check(i <= 9);
9     __voblnt_check(i >= 0);
10  }
11 }

```

Widening sends the upper bound of i to $+\infty$, and narrowing through the test $i \geq 10$ brings it back to 9 [59, slide 59]. Both checks are **PROVED** over $[0, 9]$ (claim [flavors-reset-bound-interval](#)). Goblint’s Interval proves $i \leq 9$ but loses $i \geq 0$ because its widening stops at the largest int, where $i + 1$ may overflow and `assume_top` gives the sum the whole range of int. With `assume_none`, which assumes that no signed operation overflows, Goblint proves $i \geq 0$ as well ([results.tsv](#)).

13.4.4. Narrowing foiled

Two variants of the same counter defeat narrowing. In the skipped increment (left), a nondeterministic branch may skip the increment [59, slide 60]. In the reset at equality (right), the counter resets on $i == 10$ instead of $i \geq 10$ [59, slide 77]:

```

1 fun main() {
2   i = 0;
3   while (true) {
4     c = __voblnt_nondet_int();
5     if (c) {
6       i = i + 1;
7       if (i >= 10) {
8         i = 0;
9       }
10    }
11    __voblnt_check(i <= 9);
12  }
13 }

```

skipped increment

```

1 fun main() {
2   i = 0;
3   while (true) {
4     i = i + 1;
5     if (i == 10) {
6       i = 0;
7     }
8     __voblnt_check(i <= 9);
9   }
10 }

```

reset at equality

In the skipped increment, the skipping path carries the loop-head value back unchanged. Every interval $[0, c]$ with $c \geq 9$ is therefore a fixpoint, the widened value included, and a decreasing iteration cannot descend below a fixpoint [29, §4.7.3, Ex. 4.7], [64]. In the reset at equality, the test removes no bound from an interval, so nothing stops the widened bound at 9. Both checks are **UNKNOWN** in Voblnt, with $i \in [0, +\infty]$ (claims [flavors-skipped-interval](#) and [flavors-reset-eq-interval](#)). Goblint’s octagons do not help either, since the loss comes from the iteration.

The known repairs change how the solver iterates. Goblint proves both checks when it delays widening by 10 rounds [63, §7.1.3], but not by 9, so the delay must exceed the loop bound. It also proves them with threshold widening [63, §7.1.2], but only when the thresholds include the 9 of the check itself. Policy iteration [65, §3.2] and guided static analysis [66] replace the decreasing iteration altogether. In Voblnt, each repair would change the vendored solver and its partial-correctness proof.

The flavors benchmark has eight checks on small programs, so it compares the mechanisms of the two analyzers and says little about their precision on larger programs. On these checks, Voblnt loses no precision to verification on the mechanisms both analyzers share. With Int and the order analysis, it proves five of the eight checks of Table 13.2, and Goblint’s plain Interval proves three. Across all its settings, Goblint proves all eight, though no single setting does, using widening delays, thresholds, and path partitioning. Voblnt lacks these techniques, and each would bring its own proof obligation.

13.4.5. Procedures

The second benchmark compares calling contexts and the update rules for global unknowns on five programs from Goblint’s own regression suite, adapted to VIMP (Table 13.3). Goblint runs on C versions whose checks all hold in a native run ([goblint-contexts](#)). Voblint runs Int, and Goblint runs Interval with its default domain of excluded values, so both refine a guard such as $i \neq 0$.

In *three contexts*, a loop steps i by $f(i) = i + 1$ and then checks $i == 10$. The *deep call chain* consists of recursive procedures that call each other up to 21 levels deep, and its ten checks ask for their results. In the *chain with guards*, f calls g , g calls h , and h counts its argument down to 0 behind the guards $i == 0$ and $0 < i$, and all seven calls, with arguments up to 300, return 3. In the *mutual recursion*, $f(i) = g(i - 2) + g(i - 3)$ and $g(i) = f(i + 1)$, and twelve checks ask for exact values. The *two-formal countdown* $f(x, y)$ decrements both arguments until x reaches 0 and returns y , so its three calls return 0.

Goblint’s context gas 0, call strings, full contexts, and narrowing of global unknowns ([solvers.td3.narrow-globs](#)) correspond to Voblint’s policy without contexts, its call strings, entry-state contexts, and bounded narrowing (Section 9.2). Context gas 10 ([ana.context.gas_value](#)), which keeps full contexts only for the first ten recursive calls, has no counterpart.

With entry-state contexts, Voblint proves every check of all five programs. Goblint’s full contexts prove every check except the one of the three-contexts program. After its loop, an interval only bounds i to $[10, 11]$. Voblint’s Int also knows that i is even, which leaves 10, and Goblint’s configuration has no parity or congruence component. With bounded contexts, Goblint proves more on the deep call chain and Voblint more on the mutual recursion. The two-formal countdown recurses up to 300 levels deep and needs a context per level, which only entry-state contexts provide.

On this benchmark, Voblint’s entry-state contexts prove every check that Goblint’s full contexts prove, and one more. With call strings, each analyzer proves checks the other loses, and without contexts Voblint proves more. Goblint’s widening then loses the lower bound 0 of i in h unless it narrows global unknowns ([results.tsv](#)), whereas Voblint keeps it and turns $i \neq 0$ into $i > 0$.

		three contexts 	deep call chain 	chain with guards 	mutual recursion 	two-formal countdown 
Voblint	none	0/1	4/10	7/7	0/12	0/3
	bounded narrowing	1/1	4/10	7/7	0/12	0/3
	call string, $k = 1$	0/1	4/10	7/7	2/12	0/3
	call string, $k = 5$	0/1	4/10	7/7	2/12	0/3
	entry state	1/1	10/10	7/7	12/12	3/3
Goblint	none	0/1	0/10	0/7	0/12	0/3
	narrowing globals	0/1	4/10	7/7	0/12	0/3
	call string, $k = 1$	0/1	0/10	0/7	0/12	0/3
	call string, $k = 5$	0/1	6/10	0/7	0/12	0/3
	full	0/1	10/10	7/7	12/12	3/3
	gas 10	0/1	9/10	0/7	12/12	0/3

Table 13.3: Checks proved out of those that hold, on procedure programs adapted from Goblint’s regression suite (Voblint with Int, Goblint with Interval). Voblint: no contexts, also with bounded narrowing; call strings of length k ; entry-state contexts. Goblint ([results.tsv](#)): context gas 0, also with narrowing of global unknowns; call strings of length k ; its default full contexts; context gas 10.

13.5. Running time

A sound result is of little use if the analyzer takes too long to produce it. Speed, however, was not a goal of this thesis. Apart from the defect fixed in Section 12.2, the generated code runs as Isabelle emits it. The measurements below therefore ask only whether this unoptimized code runs in acceptable time on the regression corpus.

The [profiling page](#) reports the measurements in full. The profiler runs the 369 well-formed regression fixtures that set no timeout of their own under all 15 combinations of domain and context policy, with the default update rule warrow. Each pair of a fixture and a configuration is one case.

Of the 5535 cases, 21 pass the limit of 30 s, all of them under entry-state contexts. The other 5514 cases finish, and each is timed as the median of 3 runs. Most of them are solved within a few milliseconds, except for recursive programs under entry-state contexts, where successive calls can create distinct contexts. A procedure h that calls itself n times with arguments $n, n-1, \dots, 0$ shows the effect. Sign and Parity have only a few abstract values, so the number of contexts stays bounded and the solve time is nearly constant. Congruence, Interval and Int tell the successive arguments apart and create one context per level. Their solve time grows about quadratically, although the number of solver evaluations grows only about linearly.

The same gap appears without recursion. The profiling page grows 8 sizes of a program independently, from variables and statements to call chains. In all of them the number of evaluations grows about linearly, while solve time grows as $n^{2.0}$ to $n^{2.9}$ (Figure 13.3). Each evaluation therefore gets more expensive as the program grows, along every size measured. The CPU profiles trace this cost to the data structures of the generated code. Its finite sets and maps are lists (Section 12.2), so a membership test scans the list, a union inserts element by element, and removing duplicates compares every pair. Balanced trees or hash tables would avoid these scans. The Collections Framework [67] refines abstract sets and maps into such structures, and the Light-weight Containers [68] can replace the list-based default of code generation altogether. Either route needs only targeted replacements in the code setup and no change to the soundness proofs.

The generated analyzer therefore handles the regression corpus in acceptable time, while its data structures limit how far it scales. These measurements cover small programs on one machine and give no performance guarantee beyond them.

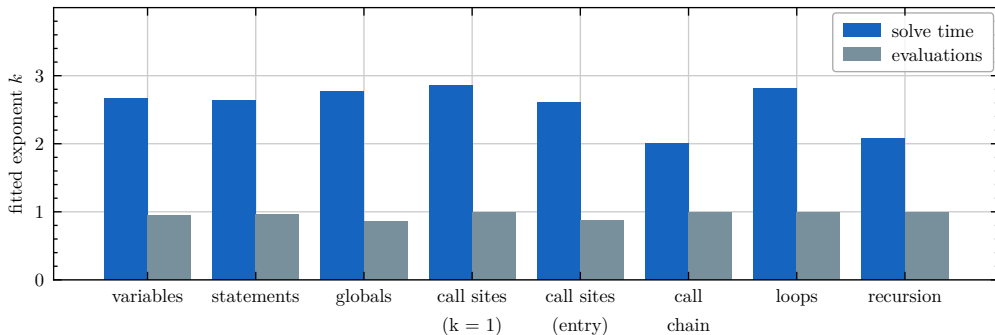


Figure 13.3: Fitted exponents k of n^k for solve time and solver evaluations, each family growing one program size n under Interval. The fits describe the measured ranges and are no complexity bounds.

PART VI

Context and Conclusion

14. Related Work

Most parts of Voblint build on earlier work, from verified analyzers and solvers to Goblint and its local-trace semantics. The sections below compare each line of work with Voblint and separate what Voblint inherits from what it adds. The literature was found through web and OpenAlex searches. AI agents supported this work heavily, from the search to checking each claim against its source (statement on the use of generative AI).

14.1. Unverified analyzers for C

Most analyzers applied to real C programs have no machine-checked soundness proof, and they differ from Voblint most in how they treat procedures. Table 14.1 compares six of them with Voblint. Five of them claim to be sound. Pulse-X is a bug finder instead, and its under-approximating soundness theorem is proved on paper [69, pp. 81:2–81:3]. Four of them inline the callee at every call and therefore need no context policy [63, §5.3], [70, §5.3], [71, §5.1], [72, §3]. Inlining cannot handle unbounded recursion, and these tools exclude recursion, unroll it to a bounded depth, rely on user-written contracts, or assume that a recursive call may change any value [63, §5.3], [70, §6.3.9], [73, Limitations] ([IKOS README](#)). Goblint, in contrast, keeps one abstract state per program point and calling context and analyzes recursive programs.

Without proof, these tools are tested against independent evidence, such as compiled execution, other analyzers, or the known verdicts of benchmark programs. Such testing keeps finding errors. On eight analyzers for C and C++, metamorphic testing found 24 issues, 16 of them related to soundness, and seven of the analyzers had soundness issues, MOPSA among them [74, p. 326, Tab. 2; p. 329].

14.2. Verified analyzers in the CompCert line

CompCert and Verasco state their theorems about the tool that users run, and Voblint follows their pattern for a language with recursive procedures and context-sensitive calls. CompCert is a compiler from Clight, a large subset of C, to PowerPC assembly, programmed and proved correct in Coq, now Rocq [21]. Its theorem is semantic preservation, so safety properties proved of the source hold for the executable. The compiler is extracted to Caml and linked with unverified parts. Leroy lists what remains trusted: the semantics of Clight and of the PowerPC assembly, the CIL-based parser, the assembler and the linker, and Coq’s extraction together with the Caml compiler and run-time system [21, §5]. Voblint’s guarantee has the same shape, a theorem about the function the tool runs, code generated from the prover, and an explicit list of trusted components (Figure 1.4). Analyzer soundness is weaker than compiler correctness, however, since it only over-approximates the behavior of a program, so a coarser result remains sound.

Tool	Language	Stance	Calls	Recursion	Relational	Combination	Mechanized
Astrée [52]	C subset	sound	inlining	not used	octagons	channels	none
Eva [70]	C	sound	inlining	contract	Apron	via values	none
MOPSA [71]	C, Python	sound	inlining	unsupported	Apron	products, queries	none
IKOS [72]	C/C++	sound	inlining	assumptions	Apron	fixed products	none
Pulse-X [69]	C	bug finding	summaries	unrolled	pure conditions	one domain	none
Goblint [75]	C	sound	per context	analyzed	Apron	products, queries	TD algorithm
Voblint	VIMP	proved sound	per context	covered	order analysis	products, queries	end to end

Table 14.1: Unverified analyzers and Voblint, as described by each tool’s papers and documentation. *Stance* is the tool’s own soundness claim. Under *calls*, inlining reanalyzes the callee at every call, a summary is computed once per procedure, and *per context* means one abstract state per program point and context. *Recursion* says how a recursive call is treated. *Analyzed* means the tool analyzes recursive calls, and *covered* means that Voblint’s theorem includes them. *Relational* names the domains that relate variables. Apron is a library of relational numeric domains, and Voblint’s order analysis tracks which variable is at most which. *Combination* names how domains exchange facts. Astrée’s domains send each other constraints through reduction channels, Eva’s domains communicate through the abstract values of expressions, a product runs domains side by side on one state, and a query lets one analysis ask another for a fact at the current point. *Mechanized* records what a proof assistant has checked. *End to end* means from source executions to the verdicts of the exported function, for every answer it returns (partial correctness) and with the trusted components of Figure 1.4.

Verasco is a Coq-verified analyzer for most of ISO C99, excluding recursion and dynamic allocation, that proves the absence of run-time errors [22]. Its abstract interpreter iterates over the structure of C#minor, a CompCert intermediate language, and CompCert’s semantic preservation extends the guarantee to compiled code. Verasco covers a far larger language, so the comparison concerns design only. Verasco reanalyzes functions at every call site and raises an alarm at a possible recursive call (Table 14.2, [22, §4]). In Voblint, calls instead become equations over unknowns indexed by program point and context, which a generic solver computes. Verasco’s extracted analyzer runs with unbounded fuel, and termination is proved only on paper [76, §§3.1.2, 4.3.1], so its theorem, like Voblint’s (Section 9.3), covers only the results it returns.

The earlier value analysis of Blazy et al. [77] does not verify its fixpoint iterator. An untrusted OCaml implementation of Bourdoncle’s iteration strategy [41] computes a candidate, and a verified checker accepts it as a post-fixpoint or falls back to top [77, §4]. Voblint instead runs a solver whose partial correctness is proved [4]. The checker leaves the iteration heuristics free and tests every result at run time. A verified solver needs no such test, but its proof holds for that one algorithm.

14.3. Mechanized abstract interpreters

Abstract interpreters have been mechanized in several provers. Table 14.2 compares those most closely related to Voblint. Their proofs cover small or intermediate languages, from While and IMP to C#minor, and the differences that matter for Voblint lie in recursion, contexts, and the fixpoint computation. Three of the works have no procedures at all. In every work of the table, a call is analyzed in at most one context. An analysis that splits a call into several alternatives, however, needs one context for each. Voblint’s context policy

Work	Prover	Language	Recursion	Contexts	Fixpoint	Executable
Verasco [22]	Coq	C#minor	alarm	per call site	structural	yes
Blazy et al. [77]	Coq	Cminor CFG	covered	intraprocedural	checked	yes
Cachera et al. [78]	Coq	Java Card	covered	none	solver	yes
Cachera, Pichardie [79]	Coq	While	no procedures	none	structural	yes
Dabrowski, Pichardie [80]	Coq	Java-like	covered	object-sensitive	specified	no
Nipkow, Klein [6]	Isabelle	IMP	no procedures	none	iterator	in prover
Lammich, Müller-Olm [81]	Isabelle	flowgraphs	exact	summaries	specified	no
Franceschino et al. [82]	F*	IMP	no procedures	none	structural	yes
Voblint	Isabelle	VIMP	covered	set per call	TD solver	yes

Table 14.2: Mechanized abstract interpreters with machine-checked soundness proofs, as described in each work. *Prover* is the proof assistant and *Language* the analyzed language. *Recursion* says how a recursive call is treated: *alarm* raises an alarm, *covered* means the theorem includes recursive programs, *exact* interprets calls and returns exactly, and *no procedures* means the language has none. *Contexts* says how calls are distinguished: *per call site* reanalyzes a function at every call site, *object-sensitive* computes contexts by a function, *summaries* computes one summary per procedure, *set per call* lets the context policy return several contexts for one call, and *none* and *intraprocedural* mean that calls are not distinguished or not analyzed. *Fixpoint* says how the result is computed: *structural* iterates over the program’s syntax, *iterator* over the whole annotated program, *checked* verifies the result of an untrusted iterator, *solver* and *TD solver* use a verified solver, and *specified* only states the constraints. *Executable* says whether the analyzer runs outside the prover.

may therefore admit several contexts for one call, and its semantics and proofs cover this case, although the shipped analyses do not use it (Section 4.3). Only Cachera et al. compute the fixpoint with a verified solver, and that solver handles ordinary inequations, not side-effecting ones [78].

Nipkow and Klein develop abstract interpretation in Isabelle/HOL over annotated commands of the While language IMP [6], [34]. A collecting semantics annotates each program point with a set of states, and an abstract interpreter annotates it with an abstract state. Their interpreter is parametric in a domain of abstract values. The domain supplies abstract and inverse operations with their soundness laws, and the interpreter derives forward evaluation, the backward filtering of guards, and the step of each command generically [6, Sects. 13.5, 13.7]. Voblint’s non-relational domains follow this pattern (Section 5.7). Its collecting semantics, however, ranges over activation traces of a procedure-aware control-flow graph (Chapter 4), because a return must know which caller resumes.

Other works make soundness proofs reusable by sharing one interpreter between the concrete and the abstract semantics [83], [84], [85]. Voblint instead splits analysis, context policy, and solver for a constraint-based analyzer and composes them in one theorem, `activation_collect_dg_sound`. Executable verified analyzers are not new either. Cachera and Pichardie extract theirs to OCaml [79, §§1, 6], and the analyzer of Franceschino et al. runs in a browser [82]. Interprocedural analyses have also been verified in Isabelle before [81], [86], [87]. None of these entries claims an extracted analyzer, however, and none of their semantics represents one activation with its caller chain, the object Voblint’s context relation `activation_context_rel` reads.

14.4. Verified solvers and termination results

A verified solver guarantees a property of its result for arbitrary equations, and the analyzer must give the equations their meaning. Hofmann et al. verify the local generic solver RLD in Coq [42], and de Vilhena et al. verify in Iris a local solver that discovers dependencies dynamically [88]. Stade et al. prove the top-down solver partially correct in Isabelle/HOL [3]. Tilscher et al. [4] extend this line to TD_{side} , which accepts contributions to global unknowns, verify it against a generic interface for update rules, prove five update rules of Stemmler et al. [53] sound, and refine the solver to executable code. For updates without widening and narrowing, they also prove that a terminating solve returns the least partial post-solution, minimal in both valuation and solved set. CompCert similarly hides its intraprocedural dataflow solvers behind a module type whose clients rely only on fixpoint properties of the result [89], [90].

Termination results exist, but none applies to Voblint. Tilscher et al. prove termination for top-down solvers without side effects over a finite type of unknowns [43, Thm. 2, Cor. 1], whereas Voblint’s unknowns pair nodes of an infinite type with contexts (Section 9.3). Seidl and Vogler prove on paper that their side-effecting solver terminates when side effects target only unknowns without a right-hand side and only finitely many unknowns are encountered [40, §9, Thm. 5]. Their result assumes widening and narrowing operators whose iterations stabilize [40, §3], so it does not carry over to the joining rules, which never widen. Schulze Frielinghaus et al. give a local solver that terminates on every non-recursive program [91], and both programs whose solves exceed their time limits in Section 13.3 are recursive.

Voblint includes a copy of the solver of [4], whose algorithm, update rules, and proofs belong to that work, and connects its certificate `part_post_solution` to the program. The certificate implies coverage of the concrete traces, and correctness of the VIMP-to-CFG translation transfers the coverage to source executions. The solver Voblint runs warrows every local unknown at a widening point, so the least-solution theorem does not apply, and Voblint uses only partial correctness and claims no optimality (Section 9.2).

14.5. Calling contexts and trace partitioning

Of the two classical approaches of Sharir and Pnueli (Section 2.6), Voblint’s call-string policy follows the call-string approach, and its entry-state policy resembles the functional approach tabulated per input [39]. At the return, Voblint combines the caller’s state before the call with the callee’s exit state (Section 6.1), the shape of the return function of J. Knoop and B. Steffen [92, §4.1]. Reps et al. solve such problems precisely when the facts are finite and the transfer functions distributive [93], whereas Voblint requires neither and proves no precision result. Seidl and Fecht model procedures by a pushdown automaton that pushes an activation record at each call and pops it at each return [54, §4], and Voblint’s activation traces keep this structure per activation (Chapter 4). The entry-state policy’s unknowns, a node paired with the abstract entry state of its procedure, resemble the unknowns $\langle v, d \rangle$ of their forward accumulation [54, §7.1].

Trace partitioning abstracts a set of traces by indexing it with control history before abstracting states [62]. Rival and Mauborgne define both partitions and coverings, in which one trace may belong to several indices, and present calling-context sensitivity as one such indexing. They also note that the function mapping tokens to tokens in a covering may be

replaced by a relation [62, Rem. 3.2.4]. Voblint’s activation collecting semantics is a covering of this relational kind, since a context policy may admit one activation in several contexts. Voblint mechanizes the indexing over activation traces and proves that the solved, routed result bounds the activation collecting semantics of every admitted context. Contexts also determine how many unknowns a solve creates. The context lifters of Erhard et al. [94] guarantee finitely many contexts whenever the solver updates each unknown finitely often [94, Abstract]. Voblint’s entry-state policy has no such bound, which is one reason the termination of the solve is not proved (Section 9.3).

14.6. Goblint and its local-trace semantics

Voblint follows Goblint’s architecture and adapts the local traces of Schwarz et al. to activations.

14.6.1. Correspondence with Goblint’s architecture

Side-effecting constraint systems let one equation both define a local unknown and publish contributions to global unknowns [23], [24]. Seidl et al. [2] describe how Goblint uses them to decouple a mixed flow-sensitive analysis from the solver, with digests on the analysis side and update rules on the solver side, recovering precision. Voblint adopts the split. An analysis supplies the transfer behavior, and the solver organizes the interaction.

The correspondence with Goblint’s implementation is architectural. Voblint was compared with the framework interface of Goblint’s [constraint generator](#) at revision 5320a6b7 by source inspection. There, a call runs the analysis’s `enter`, selects the callee context from the entry state, publishes it, and combines the callee’s exit state with the caller. Voblint’s equations have the same four steps, with `enter` and `combine` from the analysis (`dg_spec`) and the context chosen by the context policy. The project site’s side-by-side comparison ([site](#)) lists 19 Goblint constructs. Of these, 3 are modeled, 12 simplified, and 4 not modeled. No row claims that the two compute the same fixpoint.

Where Voblint simplifies Goblint, its results do not carry over directly. For instance, Voblint publishes the entry state to a global seed, which the entry’s local unknown reads back, whereas Goblint publishes it to the local entry unknown directly. Under the warrowing rules the seed itself can be widened, so widening applies at a different place than in Goblint, and it is not proved which of the two is more precise. Voblint also resolves call targets statically, and its analyses share one product state and one query kind without globals, whereas Goblint’s MCP also passes events, spawns, and per-analysis globals. No agreement rate between the verdicts of the two analyzers is reported, since the corpus has not been run through Goblint and the adapted fixtures record Goblint’s annotations only as prose.

Goblint’s MCP (Section 7.3) also answers a query cycle with the top element and caches answers per transfer ([mCP.ml](#) at revision 5320a6b7). Voblint’s combination follows this design and adds a proof obligation per analysis that quantifies over every channel that holds. Unlike a reduced product [17, §5.1.2], [32], which Int uses (Section 11.2), the product state keeps separate carriers and exchanges facts only through answers, so no reduction operator has to be proved for a pair of analyses. Verasco threads its channels so that the second domain can query the state the first has just computed [22, §7], whereas Voblint’s answers describe the predecessor state only, as Goblint’s do.

14.6.2. Digests and the context relation

Local traces give each thread of a concurrent program a semantics against which thread-modular analyses are proved sound and compared [5], and they were later extended to relational analyses [49, Ch. 6], [95]. Voblint adapts them to procedure activations (Section 4.2.1), which do not interfere, so no concurrency result of that work transfers.

Schwarz refines the local-trace semantics through *digests* [49, §2.3]. A digest is a total function on local traces, and the concrete semantics already splits each unknown $[u]$ of a program point into unknowns $[u, A]$, one per digest value A . The collecting semantics thus keeps one set of traces per program point and digest value, much as Voblint’s activation collecting semantics keeps one per node and context. Voblint’s context relation `activation_context_rel` (Section 4.3) plays the role of the digest for procedure activations, and it differs in two ways. First, a digest is a function, so each trace has exactly one digest value and the traces split into disjoint groups [49, §2.3, Def. 4, Thm. 4]. The context relation, in contrast, may admit one activation in several contexts, so Voblint’s groups may overlap. Second, a digest is updated at every step of a trace, whereas an activation’s contexts are fixed when it is called.

14.7. Closest precedents and inherited parts

Two mechanized analyzers come closest to Voblint’s combination of mixed flow sensitivity and calling contexts. Cachera et al. [78] combine a flow-insensitive heap with flow-sensitive local state for Java Card bytecode, but their analysis is context-insensitive. F. Dabrowski and D. Pichardie [80] prove in Coq that a context-sensitive points-to analysis with a flow-insensitive heap is sound. They instrument the concrete semantics with contexts through a Coq functor over a module type of contexts, whose call-context function computes the context of a callee from the call site, the caller’s context, and the receiver. Their specification is not executable, however, since the analyses are given only as constraints that no solver computes [80, §7]. Their concrete semantics, parameterized by a context policy, is the closest precedent for Voblint’s, which differs in three respects. Its contexts are admitted by a relation, the totality condition `call_context_total_on` makes the activation collecting semantics exhaustive (`node_collect_eq_Union_activation_collect`), and the indexing is connected to an executable solver.

Voblint thus takes its trust pattern from CompCert and Verasco, its domain interface from Nipkow and Klein [6, Sects. 13.5, 13.7], its solver from Tilscher et al. [4], its update rules from Stemmler et al. [53], side-effecting constraint systems from Seidl et al. [23] and Apinis et al. [24], widening, narrowing, and the reduced product from Cousot and Cousot [28], [32], its activation traces from the local traces of Schwarz et al. [5], and its architecture from Goblint. As far as the author is aware, the connection between these parts has not been mechanized before, namely a soundness proof that leads from a source semantics, through the activation collecting semantics, which indexes activation traces by context, to a side-effecting constraint system and its verified TD solver.

15. Conclusion

Static analyzers such as Goblint report properties intended to hold for every execution of a program, but their implementations generally lack machine-checked end-to-end soundness proofs. This thesis shows that the soundness of a Goblint-style analyzer with calling contexts, side effects, and a verified TD solver can be machine-checked from source executions to the verdicts of the analysis function. The result holds for a scalar language with recursive procedures. It is a partial-correctness result, so its guarantee holds whenever the analyzer returns an answer. For a user, a **PROVED** verdict therefore establishes the reported property for every covered execution that reaches the check, instead of resting on the tests the analyzer has passed.

The current implementation of Voblint lies between existing formalized analyzers, such as Verasco [22] and the abstract interpreter of Nipkow and Klein [6], and Goblint. The formalized analyzers state their theorem about the analyzer they run, but avoid the mechanisms Goblint relies on for interprocedural precision. Verasco reanalyzes a function at every call site and raises an alarm at a possible recursive call, and the interpreter of Nipkow and Klein analyzes IMP, which has no procedures. Goblint instead solves equations over unknowns indexed by program point and calling context, publishes side effects to global unknowns, and computes the solution with a top-down solver. The TD solver itself has been verified [4], but no proof connects Goblint’s solved equations to the executions of the program. Voblint keeps Goblint’s architecture and states a CompCert-style theorem about the analyzer it runs. As far as the author is aware, a proof that connects context-indexed, side-effecting equations and their solver to source executions has not been mechanized before (Section 14.7).

The end-to-end soundness theorem holds for every report `run_voblint` returns, under every analysis configuration it offers (Section 10.4). Activation traces give contexts a concrete meaning, and splitting the collected stores by context loses no execution (Section 4.4). Domains, context policies, cooperating analyses, and the solver each meet their own obligations, and generic theorems compose them (Section 8.3). The command-line tool and the playground execute code generated from `run_voblint` through an unverified harness (Chapter 12). The theorem therefore applies to the analysis function itself. Carrying it over to the displayed reports additionally relies on correct parsing, code generation, and presentation (Figure 1.4).

Concrete analysis configurations satisfy the premises of the end-to-end theorem, and the remainder obligation rejects a real Goblint defect (Chapter 13). On the external benchmarks of Section 13.4, Voblint proves every check that Goblint proves with its plain interval domain or with full calling contexts, and 261 fixtures of the regression corpus record the precise verdicts it computes on further programs (Section 12.5). The analyzer runs in acceptable time on that corpus, although its list-based sets and maps limit how far it scales (Section 13.5).

15.1. Lessons from the formalization

Formalizing Voblint required more than proving the components of an analyzer sound. Much of the work went into finding definitions and interfaces that let these proofs compose, and into checking that the theorems describe the intended analyzer. The following observations summarize this experience and suggest what a similar formalization might do differently.

Reuse the architecture, but reconsider the model. Goblint’s architecture is documented in the literature and has been exercised on real programs. Following it closely settled the analyzer’s design, so the work could concentrate on the soundness proof. The concrete semantics, however, had to be designed for the proof. The first prototype used whole-program executions with calls inlined. To account for one call, the soundness proof built a separate execution starting at the callee’s entry and spliced it into the caller’s trace. This handled individual calls, but it did not reflect the program’s call stack and was hard to extend to recursion. Activation traces replaced it after two weeks (Chapter 4). Each procedure activation is its own trace, a call starts a nested one, and a return resumes exactly the suspended caller, as a [design record](#) explains. The idea came from the local traces of Schwarz et al. [5], against which Goblint’s thread-modular analyses are proved sound, and not from Goblint’s implementation.

Design the interfaces before completing the proofs. The development began by following Goblint’s architecture, with every proof obligation left open as `sorry`. This made it inexpensive to try definitions, find missing assumptions, and move interfaces without repairing finished proofs. The largest revisions indeed concerned the semantics rather than individual arguments. Because Voblint followed Goblint’s structure, most interfaces were settled before the proofs were written, and later changes seldom reached finished proofs. The end-to-end theorem itself, however, took shape only as the components were connected. Stating it first and deriving each component’s obligations from it would likely have saved some rework.

Separate independent soundness obligations. Voblint separates the soundness of abstract operations, the coverage of executions by calling contexts, and the correctness of the solver, which enters only through its certificate (Section 9.3). Because each has its own obligation, a new domain, context policy, or cooperating analysis needed proofs of its own obligations but reused the generic soundness arguments unchanged. Revising the concrete semantics, by contrast, affected every layer whose obligations depend on it.

Validate the executable analyzer. The main theorem concerns the exported function `run_voblint`, and the command-line tool, the playground, and the regression tests all run code generated from this function. Regression tests committed to the repository record the expected verdicts on representative programs and detect changes in the analyzer’s observed precision. Generated theorem statements, registered claims, and repository checks keep the thesis and documentation from drifting away from the theories (Appendix C).

Make the solver observable. The [command-line tool](#) and the browser [playground](#) were also important for understanding the analyzer’s behavior. The solver trace of the command-line tool and the playground’s rendered graph and trace replay (Section 12.3) show where an analysis loses precision or where the solver does needless work. The trace hook lies outside the proof. Only traced runs execute it, and for them the trusted base assumes that it leaves the analysis results unchanged (Section 12.3). Because the analyzer

is exported to OCaml, standard profiling and debugging tools also remain available. The author used them to profile the generated code and to attach a debugger to solves that did not terminate. The trace of [issue 251](#), for instance, showed every newly entered context solved once from $\perp$ before its seed was published, and [pull request 256](#) reordered the two, as Goblint does.

Proof correctness and proof quality require different checks. Isabelle checks that the theorems follow from the definitions and assumptions, but not that the definitions model the intended analyzer. Once the definitions and interfaces had stabilized, generative AI tools produced many of the proofs. Isabelle checked their validity relative to the formal assumptions, while human review focused on the definitions, the assumptions, and the intended meaning of the theorems. Acceptance by Isabelle does not make a proof readable or reusable, however. Unused and duplicated lemmas accumulated during the development, and some arguments stayed more specialized than necessary. To find such problems, the author developed [isar-tools](#) alongside the formalization (Section C.4). It detects maintenance issues such as duplicated or subsumed lemmas, unused lemmas, leftover proof-search and diagnostic commands, redundant brackets, and inconsistent indentation. Full Isabelle builds and these maintenance checks complement each other. A full build of the development takes several minutes on the author’s laptop (an Apple M4 Pro), whereas the checks of [isar-tools](#) are deterministic and complete in seconds. Agents that edit the theories can therefore run them after individual changes, while full Isabelle builds remain necessary to establish correctness.

In retrospect, three questions required different kinds of evidence. The soundness theorem establishes that the analysis results satisfy their formal specification. The design of the semantics, the comparison with Goblint, and executable examples support the adequacy of that specification, which proved the most consequential challenge. Benchmarks, regression tests, tracing, and profiling revealed the analyzer’s practical behavior, including precision losses and needless solver work, which the soundness theorem does not address.

15.2. Relevance to Goblint and limitations

The proofs do not show that Goblint is correct, since they concern Voblint’s definitions and the correspondence with Goblint is architectural (Section 14.6). They show, however, that Goblint’s architecture admits a soundness proof that composes along its own boundaries, provided each analysis and each context policy meets its obligations.

A documented Goblint defect can be replayed as a regression fixture and as a counterexample theorem, as the congruence remainder was (Section 13.2). A proposed domain operation with a counterpart in Voblint could be checked against the obligation that its counterpart meets before the change lands in Goblint. Conversely, a verified component exported as OCaml could replace its unverified counterpart in Goblint.

For Goblint’s developers, however, the current formalization is still largely a proof of concept. It lacks many concepts that their work depends on, such as pointers, a heap, threads, machine integers, and most of Goblint’s analyses, so its immediate applicability to Goblint development is limited, although it already provides a reference for selected components and proof obligations (Section 15.3). Among the limitations identified in Section 1.4, the gap between VIMP and C is the largest. VIMP has no pointers, heap, or threads and computes with unbounded integers, so a verdict about a VIMP program does

not carry over to a C program with the same text (Section 3.6), and the adequacy of `pstep` itself is argued rather than proved. As far as the author is aware, no commonly accepted C11 semantics exists in Isabelle against which `pstep` could be checked [48], [96], [97], [98].

A second limitation concerns the evaluation of precision. No precision or completeness theorem is proved, so the evidence for precision rests on examples. The regression corpus is small and its expected verdicts were set by hand (Section 12.5), and beyond the small comparison of Section 13.4, no agreement rate with Goblint is measured (Section 14.6).

15.3. Extending Voblint towards C and Goblint

Addressing these limitations requires extending Voblint towards the language features and analyses that Goblint supports. Some analyses and concepts could be added to the current proof of concept without much change, since each soundness proof stays within its own obligations. For example, relational domains such as octagons or polyhedra [29] could follow the order analysis (Section 11.4), and path-sensitive analyses could split a call into several entry pairs, which the interface already admits (Section 6.1). Other features of Goblint, such as machine integers, pointers, or threads, would require extensive changes to Voblint. Machine integers illustrate the scale of such an extension. [Pull request 357](#), developed separately from the version this thesis describes, introduces machine-integer kinds from `i8` to `u64` and a typed core language elaborated from VIMP. Integrating it requires revisiting several layers and their soundness proofs, which is left as future work. With machine integers, Goblint’s bitfield domain, which tracks each bit separately and which the author co-developed ([Goblint pull request 1623](#)), is a natural next analysis.

Whether a further C feature needs such a rewrite depends on how deeply it reaches into the formalization. Several central results are independent of the details of VIMP. The solver certificate is stated over unknowns and right-hand sides, and the coverage contract refers to the language only through a few step functions (Section 4.4), so a feature that changes only these step functions leaves the rest of the proof intact. The trace semantics, however, assumes that a callee cannot touch its caller’s locals, so a return can resume the caller with the locals it had before the call (Section 6.1). Features that keep this assumption, such as machine integers, change the step functions and the proofs that unfold them, while the contract and its composition keep their shape. Pointers, shared mutable memory, and threads would invalidate this assumption. With a pointer into the caller’s frame or into the heap, a callee can modify data the caller reads after the return, and with threads, other code can run between a call and its return. Such features would require extending the trace semantics [5], [99], and the soundness arguments that depend on its current separation of caller and callee state would need substantial revision.

Revisions of this scale may become more feasible as AI-assisted tools improve. Such tools have already made large-scale mathematical formalization practical [13, Sec. “Auto-formalization is a practical reality”] and played an important role in the formalization presented in this thesis. Over the course of this project, the models behind these tools, including Anthropic’s Claude, OpenAI’s ChatGPT and Codex, and Cursor, handled the formalization consistently better, and the surrounding Isabelle tooling improved as well. Continued progress in both areas may make ambitious formalization projects and large-scale refactorings increasingly practical.

Bibliography

- [1] V. Vojdani, K. Apinis, V. Rötov, H. Seidl, V. Vene, and R. Vogler, “Static Race Detection for Device Drivers: The Goblin Approach,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, 2016, pp. 391–402. doi: [10.1145/2970276.2970337](https://doi.org/10.1145/2970276.2970337).
- [2] H. Seidl, V. Vojdani, J. Erhard, and M. Schwarz, “Mixed Flow-Sensitive Static Analysis: Engineering Modularity,” in *FM 2026: Formal Methods*, 2026, pp. 446–470. doi: [10.1007/978-3-032-26220-2_22](https://doi.org/10.1007/978-3-032-26220-2_22).
- [3] Y. Stade, S. Tilscher, and H. Seidl, “The Top-Down Solver Verified: Building Confidence in Static Analyzers,” in *Computer Aided Verification*, 2024, pp. 303–324. doi: [10.1007/978-3-031-65627-9_15](https://doi.org/10.1007/978-3-031-65627-9_15).
- [4] S. Tilscher, A. Graß, and H. Seidl, “Verifying a Solver for Mixed Flow-Sensitive Analyses,” in *NASA Formal Methods (NFM 2026)*, in LNCS. Springer, 2026, pp. 48–69. doi: [10.1007/978-3-032-28079-4_3](https://doi.org/10.1007/978-3-032-28079-4_3).
- [5] M. Schwarz, S. Saan, H. Seidl, K. Apinis, J. Erhard, and V. Vojdani, “Improving Thread-Modular Abstract Interpretation,” in *Static Analysis*, in LNCS, vol. 12913. 2021, pp. 359–383. doi: [10.1007/978-3-030-88806-0_18](https://doi.org/10.1007/978-3-030-88806-0_18).
- [6] T. Nipkow and G. Klein, *Concrete Semantics with Isabelle/HOL*. Springer, 2014. doi: [10.1007/978-3-319-10542-0](https://doi.org/10.1007/978-3-319-10542-0).
- [7] ISO/IEC JTC 1/SC 22/WG 14, “ISO/IEC 9899:2011, Information technology – Programming languages – C.” Accessed: Sep. 21, 2026. [Online]. Available: <https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>
- [8] Leiden Declaration Working Group, “Leiden Declaration on Artificial Intelligence and Mathematics.” [Online]. Available: <https://leidendeclaration.ai/>
- [9] AutoCorrode contributors, “AutoCorrode: Verification Infrastructure for the Isabelle/HOL Interactive Proof Assistant.” Accessed: Sep. 21, 2026. [Online]. Available: <https://github.com/aws-labs/AutoCorrode>
- [10] K. Kappelmann, “Isabelle PIDE MCP.” Accessed: Oct. 21, 2026. [Online]. Available: <https://github.com/kappelmann/isabelle-pide-mcp>
- [11] K. Kappelmann, M. Schäffeler, L. Stevens, M. Abdulaziz, A. Popescu, and D. Traytel, “Just Type It in Isabelle! AI Agents Drafting, Mechanizing, and Generalizing from Human Hints.” [Online]. Available: <https://arxiv.org/abs/2604.15713v3>
- [12] D. Bryant, J. J. H. y Munive, C. Kaliszyk, and J. Urban, “Munkres' General Topology Autoformalized in Isabelle/HOL.” Accessed: Sep. 22, 2026. [Online]. Available: <https://arxiv.org/abs/2604.07455v1>
- [13] T. Hales, “What Mathematicians Should Know About the Lean Theorem Prover: Questions of Reliability and AI.” Accessed: Oct. 21, 2026. [Online]. Available: <https://terrytao.wordpress.com/2026/10/09/what-mathematicians-should-know-about-the-lean-theorem-proverquestions-of-reliability-and-ai/>

-
- [14] M. Schwarz, “Congruences: Make % Sound by Restricting Cases Where We Return a Constant.” Accessed: Sep. 21, 2026. [Online]. Available: <https://github.com/goblint/analyzer/pull/1161>
- [15] L. C. Paulson, T. Nipkow, and M. Wenzel, “From LCF to Isabelle/HOL,” *Formal Aspects of Computing*, vol. 31, no. 6, pp. 675–698, 2019, doi: [10.1007/s00165-019-00492-1](https://doi.org/10.1007/s00165-019-00492-1).
- [16] G. Klein *et al.*, “seL4: Formal Verification of an OS Kernel,” in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, ACM, 2009, pp. 207–220. doi: [10.1145/1629575.1629596](https://doi.org/10.1145/1629575.1629596).
- [17] X. Rival and K. Yi, *Introduction to Static Analysis: An Abstract Interpretation Perspective*. MIT Press, 2020. [Online]. Available: <https://mitpress.mit.edu/9780262043410/introduction-to-static-analysis/>
- [18] H. G. Rice, “Classes of Recursively Enumerable Sets and Their Decision Problems,” *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953, doi: [10.1090/S0002-9947-1953-0053041-6](https://doi.org/10.1090/S0002-9947-1953-0053041-6).
- [19] B. Livshits *et al.*, “In Defense of Soundness: A Manifesto,” *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015, doi: [10.1145/2644805](https://doi.org/10.1145/2644805).
- [20] Goblint developers, “Assumptions.” Accessed: Oct. 17, 2026. [Online]. Available: <https://goblint.readthedocs.io/en/latest/user-guide/assumptions/>
- [21] X. Leroy, “Formal Verification of a Realistic Compiler,” *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009, doi: [10.1145/1538788.1538814](https://doi.org/10.1145/1538788.1538814).
- [22] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie, “A Formally-Verified C Static Analyzer,” in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM, 2015, pp. 247–259. doi: [10.1145/2676726.2676966](https://doi.org/10.1145/2676726.2676966).
- [23] H. Seidl, V. Vene, and M. Müller-Olm, “Global Invariants for Analysing Multi-threaded Applications,” *Proceedings of the Estonian Academy of Sciences. Physics. Mathematics*, vol. 52, no. 4, pp. 413–436, 2003, doi: [10.3176/phys.math.2003.4.05](https://doi.org/10.3176/phys.math.2003.4.05).
- [24] K. Apinis, H. Seidl, and V. Vojdani, “Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis,” in *Programming Languages and Systems (APLAS 2012)*, in LNCS, vol. 7705. Springer, 2012, pp. 157–172. doi: [10.1007/978-3-642-35182-2_12](https://doi.org/10.1007/978-3-642-35182-2_12).
- [25] M. Lerchner, “Voblint: A Verified Goblint-Style Static Analysis Pipeline in Isabelle/HOL.” Zenodo, 2026. doi: [10.5281/zenodo.23260696](https://doi.org/10.5281/zenodo.23260696).
- [26] F. Haftmann and T. Nipkow, “Code Generation via Higher-Order Rewrite Systems,” in *Functional and Logic Programming (FLOPS 2010)*, in LNCS, vol. 6009. Springer, 2010, pp. 103–117. doi: [10.1007/978-3-642-12251-4_9](https://doi.org/10.1007/978-3-642-12251-4_9).
- [27] S. Saan, “Correctness Witnesses for Thread-Modular Program Analysis,” Doctoral dissertation, 2026. [Online]. Available: <https://dspace.ut.ee/server/api/core/bitstreams/00ec3799-947c-48eb-99e3-f43c9646bbcb/content>
- [28] P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, ACM, 1977, pp. 238–252. doi: [10.1145/512950.512973](https://doi.org/10.1145/512950.512973).
-

-
- [29] A. Miné, “Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation,” *Foundations and Trends in Programming Languages*, vol. 4, no. 3–4, pp. 120–372, 2017, doi: [10.1561/25000000034](https://doi.org/10.1561/25000000034).
- [30] A. Tarski, “A Lattice-Theoretical Fixpoint Theorem and Its Applications,” *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 285–309, 1955, doi: [10.2140/pjm.1955.5.285](https://doi.org/10.2140/pjm.1955.5.285).
- [31] H. Seidl, R. Wilhelm, and S. Hack, *Compiler Design: Analysis and Transformation*. Berlin, Heidelberg: Springer, 2012. doi: [10.1007/978-3-642-17548-0](https://doi.org/10.1007/978-3-642-17548-0).
- [32] P. Cousot and R. Cousot, “Systematic Design of Program Analysis Frameworks,” in *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, ACM, 1979, pp. 269–282. doi: [10.1145/567752.567778](https://doi.org/10.1145/567752.567778).
- [33] P. Cousot and R. Cousot, “Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation,” in *Programming Language Implementation and Logic Programming (PLILP 1992)*, in LNCS, vol. 631. Springer, 1992, pp. 269–295. doi: [10.1007/3-540-55844-6_142](https://doi.org/10.1007/3-540-55844-6_142).
- [34] T. Nipkow, “Abstract Interpretation of Annotated Commands,” in *Interactive Theorem Proving (ITP 2012)*, in LNCS. Springer, 2012, pp. 116–132. doi: [10.1007/978-3-642-32347-8_9](https://doi.org/10.1007/978-3-642-32347-8_9).
- [35] K. Apinis, H. Seidl, and V. Vojdani, “How to Combine Widening and Narrowing for Non-Monotonic Systems of Equations,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, 2013, pp. 377–386. doi: [10.1145/2491956.2462190](https://doi.org/10.1145/2491956.2462190).
- [36] K. Apinis, H. Seidl, and V. Vojdani, “Enhancing Top-Down Solving with Widening and Narrowing,” in *Semantics, Logics, and Calculi*, vol. 9560, in LNCS, vol. 9560., Springer, 2016, pp. 272–288. doi: [10.1007/978-3-319-27810-0_14](https://doi.org/10.1007/978-3-319-27810-0_14).
- [37] A. Graß, “Towards the Verification of Top-Down Solvers,” Master's thesis, Technische Universität München, 2024. Accessed: Sep. 22, 2026. [Online]. Available: <https://www.tcs.ifi.lmu.de/staff/alexandra-grass/master-thesis-alexandra-grass.pdf>
- [38] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2006.
- [39] M. Sharir and A. Pnueli, “Two Approaches to Interprocedural Data Flow Analysis,” in *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds., Prentice-Hall, 1981, ch. 7.
- [40] H. Seidl and R. Vogler, “Three improvements to the top-down solver,” *Mathematical Structures in Computer Science*, vol. 31, no. 9, pp. 1090–1134, 2021, doi: [10.1017/S0960129521000499](https://doi.org/10.1017/S0960129521000499).
- [41] F. Bourdoncle, “Efficient Chaotic Iteration Strategies with Widenings,” in *Formal Methods in Programming and Their Applications*, in LNCS, vol. 735. Springer, 1993, pp. 128–141. doi: [10.1007/BFb0039704](https://doi.org/10.1007/BFb0039704).
- [42] M. Hofmann, A. Karbyshev, and H. Seidl, “Verifying a Local Generic Solver in Coq,” in *Static Analysis (SAS 2010)*, in LNCS. Springer, 2010, pp. 340–355. doi: [10.1007/978-3-642-15769-1_21](https://doi.org/10.1007/978-3-642-15769-1_21).
- [43] S. Tilscher, A. Graß, H. Seidl, and Y. Stade, “Proving Total Correctness of Top-Down Solvers with Widening and Narrowing,” *Journal of Automated Reasoning*, vol. 70, no. 2, 2026, doi: [10.1007/s10817-026-09756-x](https://doi.org/10.1007/s10817-026-09756-x).
-

-
- [44] P. Lammich and S. Wimmer, “IMP2 – Simple Program Verification in Isabelle/HOL,” *Archive of Formal Proofs*, Jan. 2019, [Online]. Available: <https://isa-afp.org/entries/IMP2.html>
- [45] N. Schirmer, “A Sequential Imperative Programming Language Syntax, Semantics, Hoare Logics and Verification Environment,” *Archive of Formal Proofs*, Feb. 2008, Accessed: Sep. 22, 2026. [Online]. Available: <https://isa-afp.org/entries/Simpl.html>
- [46] G. D. Plotkin, “A Structural Approach to Operational Semantics,” *Journal of Logic and Algebraic Programming*, vol. 60–61, pp. 17–139, 2004, doi: [10.1016/j.jlap.2004.05.001](https://doi.org/10.1016/j.jlap.2004.05.001).
- [47] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, “CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs,” in *Compiler Construction (CC 2002)*, in LNCS, vol. 2304. Springer, 2002, pp. 213–228. doi: [10.1007/3-540-45937-5_16](https://doi.org/10.1007/3-540-45937-5_16).
- [48] M. Brecknell *et al.*, “AutoCorres2,” *Archive of Formal Proofs*, Apr. 2024, Accessed: Oct. 04, 2026. [Online]. Available: <https://isa-afp.org/entries/AutoCorres2.html>
- [49] M. B. Schwarz, “Thread-Modular Abstract Interpretation: The Local Perspective,” Doctoral dissertation, Technische Universität München, 2025. [Online]. Available: <https://mediatum.ub.tum.de/doc/1765098/n8cop1l4luxwzq7hmzpv376a6.tmai-local-perspective-schwarz.pdf>
- [50] P. Cousot, “The Calculational Design of a Generic Abstract Interpreter,” in *Calculational System Design*, vol. 173, M. Broy and R. Steinbrüggen, Eds., in NATO ASI Series F, vol. 173., IOS Press, 1999, pp. 421–505.
- [51] K. Apinis, “Frameworks for Analyzing Multi-threaded C,” Doctoral dissertation, 2014. [Online]. Available: <https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20140606-1189191-0-9>
- [52] P. Cousot *et al.*, “Combination of Abstractions in the ASTRÉE Static Analyzer,” in *Advances in Computer Science – ASIAN 2006. Secure Software and Related Issues*, in LNCS, vol. 4435. Springer, 2007, pp. 272–300. doi: [10.1007/978-3-540-77505-8_23](https://doi.org/10.1007/978-3-540-77505-8_23).
- [53] F. Stemmler, M. Schwarz, J. Erhard, S. Tilscher, and H. Seidl, “Taking Out the Toxic Trash: Recovering Precision in Mixed Flow-Sensitive Static Analyses,” *Proceedings of the ACM on Programming Languages*, vol. 9, no. PLDI, pp. 1167–1193, 2025, doi: [10.1145/3729297](https://doi.org/10.1145/3729297).
- [54] H. Seidl and C. Fecht, “Interprocedural Analyses: A Comparison,” *The Journal of Logic Programming*, vol. 43, no. 2, pp. 123–156, 2000, doi: [10.1016/S0743-1066\(99\)00058-8](https://doi.org/10.1016/S0743-1066(99)00058-8).
- [55] P. Granger, “Static Analysis of Arithmetical Congruences,” *International Journal of Computer Mathematics*, vol. 30, no. 3–4, pp. 165–190, 1989, doi: [10.1080/00207168908803778](https://doi.org/10.1080/00207168908803778).
- [56] B. Huffman and O. Kunčar, “Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL,” in *Certified Programs and Proofs (CPP 2013)*, in LNCS. Springer, 2013, pp. 131–146. doi: [10.1007/978-3-319-03545-1_9](https://doi.org/10.1007/978-3-319-03545-1_9).
- [57] J. Kim, “Mechanizing Typed Regulatory Actions for Security Tokens: Semantics, Falsification, and Bounded EVM Evidence.” [Online]. Available: <https://arxiv.org/abs/2608.29134v3>
- [58] D. Marmsoler, “Isabelle/STARK: A Formalization of zk-STARK in Isabelle/HOL.” [Online]. Available: <https://arxiv.org/abs/2608.01965v1>
-

-
- [59] D. Monniaux, “Flavors of Abstract Interpretation.” [Online]. Available: <https://hal.science/hal-04750222>
- [60] B. Jeannet and A. Miné, “Apron: A Library of Numerical Abstract Domains for Static Analysis,” in *Computer Aided Verification (CAV 2009)*, in LNCS, vol. 5643. Springer, 2009, pp. 661–667. doi: [10.1007/978-3-642-02658-4_52](https://doi.org/10.1007/978-3-642-02658-4_52).
- [61] P. Cousot and N. Halbwachs, “Automatic Discovery of Linear Restraints Among Variables of a Program,” in *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, ACM, 1978, pp. 84–96. doi: [10.1145/512760.512770](https://doi.org/10.1145/512760.512770).
- [62] X. Rival and L. Mauborgne, “The Trace Partitioning Abstract Domain,” *ACM Transactions on Programming Languages and Systems*, vol. 29, no. 5, 2007, doi: [10.1145/1275497.1275501](https://doi.org/10.1145/1275497.1275501).
- [63] B. Blanchet *et al.*, “A Static Analyzer for Large Safety-Critical Software,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, ACM, 2003, pp. 196–207. doi: [10.1145/781131.781153](https://doi.org/10.1145/781131.781153).
- [64] N. Halbwachs and J. Henry, “When the Decreasing Sequence Fails,” in *Static Analysis (SAS 2012)*, in LNCS, vol. 7460. Springer, 2012, pp. 198–213. doi: [10.1007/978-3-642-33125-1_15](https://doi.org/10.1007/978-3-642-33125-1_15).
- [65] A. Costan, S. Gaubert, É. Goubault, M. Martel, and S. Putot, “A Policy Iteration Algorithm for Computing Fixed Points in Static Analysis of Programs,” in *Computer Aided Verification (CAV 2005)*, in LNCS, vol. 3576. Springer, 2005, pp. 462–475. doi: [10.1007/11513988_46](https://doi.org/10.1007/11513988_46).
- [66] D. Gopan and T. Reps, “Guided Static Analysis,” in *Static Analysis (SAS 2007)*, in LNCS, vol. 4634. Springer, 2007, pp. 349–365. doi: [10.1007/978-3-540-74061-2_22](https://doi.org/10.1007/978-3-540-74061-2_22).
- [67] P. Lammich and A. Lochbihler, “The Isabelle Collections Framework,” in *Interactive Theorem Proving (ITP 2010)*, in LNCS, vol. 6172. Springer, 2010, pp. 339–354. doi: [10.1007/978-3-642-14052-5_24](https://doi.org/10.1007/978-3-642-14052-5_24).
- [68] A. Lochbihler, “Light-Weight Containers for Isabelle: Efficient, Extensible, Nestable,” in *Interactive Theorem Proving (ITP 2013)*, in LNCS, vol. 7998. Springer, 2013, pp. 116–132. doi: [10.1007/978-3-642-39634-2_11](https://doi.org/10.1007/978-3-642-39634-2_11).
- [69] Q. L. Le, A. Raad, J. Villard, J. Berdine, D. Dreyer, and P. W. O’Hearn, “Finding Real Bugs in Big Programs with Incorrectness Logic,” *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, p. 81:1–81:27, 2022, doi: [10.1145/3527325](https://doi.org/10.1145/3527325).
- [70] D. Bühler, P. Cuoq, and B. Yakobowski, “The Eva Plug-in, for Frama-C 33.0 (Arsenic).” Accessed: Sep. 29, 2026. [Online]. Available: <https://www.frama-c.com/download/frama-c-eva-manual.pdf>
- [71] M. Journault, A. Miné, R. Monat, and A. Ouadjaout, “Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer,” in *Verified Software. Theories, Tools, and Experiments (VSTTE 2019)*, in LNCS. Springer, 2020, pp. 1–18. doi: [10.1007/978-3-030-41600-3_1](https://doi.org/10.1007/978-3-030-41600-3_1).
- [72] G. Brat, J. A. Navas, N. Shi, and A. Venet, “IKOS: A Framework for Static Analysis Based on Abstract Interpretation,” in *Software Engineering and Formal Methods (SEFM 2014)*, in LNCS. Springer, 2014, pp. 271–277. doi: [10.1007/978-3-319-10431-7_20](https://doi.org/10.1007/978-3-319-10431-7_20).
-

-
- [73] Mopsa developers, “Mopsa User Manual.” Accessed: Sep. 29, 2026. [Online]. Available: <https://mopsa.gitlab.io/mopsa-analyzer/user-manual/>
- [74] D. Kaindlstorfer, A. Isychev, V. Wüstholtz, and M. Christakis, “Interrogation Testing of Program Analyzers for Soundness and Precision Issues,” in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, 2024, pp. 319–330. doi: [10.1145/3691620.3695034](https://doi.org/10.1145/3691620.3695034).
- [75] S. Saan *et al.*, “Goblint: Autotuning Thread-Modular Abstract Interpretation (Competition Contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2023)*, in LNCS, vol. 13994. Springer, 2023, pp. 547–552. doi: [10.1007/978-3-031-30820-8_34](https://doi.org/10.1007/978-3-031-30820-8_34).
- [76] J.-H. Jourdan, “Verasco: a Formally Verified C Static Analyzer,” Doctoral dissertation, 2016. [Online]. Available: <https://theses.hal.science/tel-01327023>
- [77] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie, “Formal Verification of a C Value Analysis Based on Abstract Interpretation,” in *Static Analysis*, 2013, pp. 324–344. doi: [10.1007/978-3-642-38856-9_18](https://doi.org/10.1007/978-3-642-38856-9_18).
- [78] D. Cachera, T. Jensen, D. Pichardie, and V. Rusu, “Extracting a Data Flow Analyser in Constructive Logic,” *Theoretical Computer Science*, vol. 342, no. 1, pp. 56–78, 2005, doi: [10.1016/j.tcs.2005.06.004](https://doi.org/10.1016/j.tcs.2005.06.004).
- [79] D. Cachera and D. Pichardie, “A Certified Denotational Abstract Interpreter,” in *Interactive Theorem Proving (ITP 2010)*, in LNCS. Springer, 2010, pp. 9–24. doi: [10.1007/978-3-642-14052-5_3](https://doi.org/10.1007/978-3-642-14052-5_3).
- [80] F. Dabrowski and D. Pichardie, “A Certified Data Race Analysis for a Java-like Language,” in *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, in LNCS. 2009, pp. 212–227. doi: [10.1007/978-3-642-03359-9_16](https://doi.org/10.1007/978-3-642-03359-9_16).
- [81] P. Lammich and M. Müller-Olm, “Formalization of Conflict Analysis of Programs with Procedures, Thread Creation, and Monitors,” *Archive of Formal Proofs*, Dec. 2007, Accessed: Sep. 22, 2026. [Online]. Available: <https://isa-afp.org/entries/Program-Conflict-Analysis.html>
- [82] L. Franceschino, D. Pichardie, and J.-P. Talpin, “Verified Functional Programming of an Abstract Interpreter,” in *Static Analysis (SAS 2021)*, in LNCS. Springer, 2021, pp. 124–143. doi: [10.1007/978-3-030-88806-0_6](https://doi.org/10.1007/978-3-030-88806-0_6).
- [83] S. Michelland, Y. Zakowski, and L. Gonnord, “Abstract Interpreters: A Monadic Approach to Modular Verification,” *Proceedings of the ACM on Programming Languages*, vol. 8, no. ICFP, 2024, doi: [10.1145/3674646](https://doi.org/10.1145/3674646).
- [84] S. Keidel, C. B. Poulsen, and S. Erdweg, “Compositional Soundness Proofs of Abstract Interpreters,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, 2018, doi: [10.1145/3236767](https://doi.org/10.1145/3236767).
- [85] D. Darais, M. Might, and D. V. Horn, “Galois Transformers and Modular Abstract Interpreters: Reusable Metatheory for Program Analysis,” in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*, ACM, 2015, pp. 552–571. doi: [10.1145/2814270.2814308](https://doi.org/10.1145/2814270.2814308).
- [86] D. Wasserrab, “Backing up Slicing: Verifying the Interprocedural Two-Phase Horwitz-Reps-Binkley Slicer,” *Archive of Formal Proofs*, Nov. 2009, Accessed: Sep. 22, 2026. [Online]. Available: <https://isa-afp.org/entries/HRB-Slicing.html>
-

-
- [87] J. Breitner, “Shivers' Control Flow Analysis,” *Archive of Formal Proofs*, Nov. 2010, Accessed: Sep. 22, 2026. [Online]. Available: <https://isa-afp.org/entries/Shivers-CFA.html>
- [88] P. E. de Vilhena, F. Pottier, and J.-H. Jourdan, “Spy Game: Verifying a Local Generic Solver in Iris,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, p. 33:1–33:28, 2020, doi: [10.1145/3371101](https://doi.org/10.1145/3371101).
- [89] CompCert developers, “CompCert Module Kildall: Solvers for Dataflow Inequalities.” Accessed: Sep. 22, 2026. [Online]. Available: <https://compcert.org/doc/html/compcert.backend.Kildall.html>
- [90] R. L. Spina, D. Demange, and S. Blazy, “Formal Verification of WTO-based Dataflow Solvers,” in *Programming Languages and Systems (ESOP 2025)*, in LNCS. Springer, 2025, pp. 91–118. doi: [10.1007/978-3-031-91121-7_4](https://doi.org/10.1007/978-3-031-91121-7_4).
- [91] S. Schulze Frielinghaus, H. Seidl, and R. Vogler, “Enforcing Termination of Interprocedural Analysis,” *Formal Methods in System Design*, vol. 53, no. 2, pp. 313–338, 2018, doi: [10.1007/s10703-017-0288-5](https://doi.org/10.1007/s10703-017-0288-5).
- [92] J. Knoop and B. Steffen, “The Interprocedural Coincidence Theorem,” in *Compiler Construction (CC '92)*, in LNCS, vol. 641. Springer, 1992, pp. 125–140. doi: [10.1007/3-540-55984-1_13](https://doi.org/10.1007/3-540-55984-1_13).
- [93] T. Reps, S. Horwitz, and M. Sagiv, “Precise Interprocedural Dataflow Analysis via Graph Reachability,” in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM, 1995, pp. 49–61. doi: [10.1145/199448.199462](https://doi.org/10.1145/199448.199462).
- [94] J. Erhard, J. F. Schinabeck, M. Schwarz, and H. Seidl, “Context Gas and friends: taming context-sensitivity on the fly,” *International Journal on Software Tools for Technology Transfer*, vol. 27, no. 2, pp. 289–307, 2025, doi: [10.1007/s10009-025-00803-3](https://doi.org/10.1007/s10009-025-00803-3).
- [95] M. Schwarz, S. Saan, H. Seidl, J. Erhard, and V. Vojdani, “Clustered Relational Thread-Modular Abstract Interpretation with Local Traces,” in *Programming Languages and Systems (ESOP 2023)*, in LNCS, vol. 13990. 2023, pp. 28–58. doi: [10.1007/978-3-031-30044-8_2](https://doi.org/10.1007/978-3-031-30044-8_2).
- [96] F. Tuong and B. Wolff, “Deeply Integrating C11 Code Support into Isabelle/PIDE,” in *Proceedings of the Fifth Workshop on Formal Integrated Development Environment (F-IDE 2019)*, in EPTCS, vol. 310. 2019, pp. 13–28. doi: [10.4204/EPTCS.310.3](https://doi.org/10.4204/EPTCS.310.3).
- [97] R. Krebbers, “The C Standard Formalized in Coq,” Doctoral dissertation, 2015.
- [98] K. Memarian, “The Cerberus C Semantics,” PhD thesis, Technical Report UCAM-CL-TR-981, 2023. [Online]. Available: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-981.html>
- [99] P. Sotin and B. Jeannet, “Precise Interprocedural Analysis in the Presence of Pointers to the Stack,” in *Programming Languages and Systems (ESOP 2011)*, in LNCS. Springer, 2011, pp. 459–479. doi: [10.1007/978-3-642-19718-5_24](https://doi.org/10.1007/978-3-642-19718-5_24).
- [100] F. Haftmann and M. Wenzel, “Constructive Type Classes in Isabelle,” in *Types for Proofs and Programs (TYPES 2006)*, in LNCS. Springer, 2007, pp. 160–174. doi: [10.1007/978-3-540-74464-1_11](https://doi.org/10.1007/978-3-540-74464-1_11).
- [101] C. Ballarín, “Locales: A Module System for Mathematical Theories,” *Journal of Automated Reasoning*, vol. 52, no. 2, pp. 123–153, 2014, doi: [10.1007/s10817-013-9284-7](https://doi.org/10.1007/s10817-013-9284-7).
-

-
- [102] L. C. Paulson, “Why not just use Lean?.” Accessed: Sep. 27, 2026. [Online]. Available: https://lawrencecpaulson.github.io/2026/04/23/Why_not_Lean.html
- [103] A. Chlipala, *Certified Programming with Dependent Types*. MIT Press, 2013. [Online]. Available: <http://adam.chlipala.net/cpdt/>

A. Reading the Isabelle Listings

This appendix explains the Isabelle/HOL syntax [6] of the listings that the chapters lift from the theories. Readers who know Isabelle can skip it.

HOL is a typed logic of total functions. The thesis writes $x :: \tau$ for “ x has type τ ”, $\tau_1 \Rightarrow \tau_2$ for the type of functions from τ_1 to τ_2 , **a** for a type variable, and $\lambda x.t$ for the function that maps x to t . A function is applied by juxtaposition, as in $f x y$. A *datatype* declares a type by its constructors, such as **gcom** (Section 3.1). A *record* is a tuple with named fields, such as **dg_spec**, and a **type_synonym** gives a new name for an existing type.

A **definition** gives a new constant by one equation whose right-hand side does not mention the constant, so it only names an existing term and needs no proof. The right-hand side may be any term, for instance, a set comprehension $\{s.P\}$ over an infinite type, so a definition need not be computable. Below, $\forall x$ ranges over all variables:

```
1 (* src/Abstract_Interpreter/Domain/State/Nonrelational_State.thy *)
2 definition gamma_state :: "('a::numeric_domain) abs_state => store set" where
3   "gamma_state d = {s.  $\forall x. s\ x \in \gamma\ (d\ x)}$ "
```

Isabelle

A **fun** may instead give one equation per constructor pattern and may call itself. The function below has one equation for the empty list `[]` and one for a list `q # qs`, and its result type **'b option** has the values `None` and `Some y`:

```
1 (* src/Analyses/Int/Int_Arithmetic.thy *)
2 fun first_deciding :: "('a => 'b option) list => 'a => 'b option" where
3   "first_deciding [] x = None"
4 | "first_deciding (q # qs) x = (case q x of Some y => Some y | None => first_deciding qs x)"
```

Isabelle

Since every HOL function is total, Isabelle proves that such a recursion terminates [6, §2.3.4]. A function may also be defined without this proof. Its equations then hold only on its *domain*, the arguments on which the recursion terminates, as for the verified TD solver and its domain `solve_dom`.

A theorem declares its variables and their types after **fixes**, introduces abbreviations local to the statement after **defines**, lists its premises after **assumes**, and states its conclusion after **shows**. The end-to-end soundness theorem of Section 10.4 uses all four:

```
1 (* src/Executable_Surface/CLI/Analysis_Certified.thy *)
2 theorem run_voblint_source_sound:
3   fixes p :: imp_prog and s0 s :: store
4   defines G_def: "G  $\equiv$  declared_global p"
5   and Pi_def: "Pi  $\equiv$  prog_table p"
6   and g_def: "g  $\equiv$  prog_cfg p"
7   assumes s0: "s0  $\in$  cinit_stores G"
8   and run: "G, Pi  $\vdash$  (main_body Pi, s0, [])  $\rightarrow_p^*$  (residual, s, frs)"
9   and ans: "run_voblint config p = Analysed res"
10  shows "( $\exists v. p, s0 \vdash (residual, s, frs) \triangleright v$ )
11     $\wedge$  ( $\forall v. p, s0 \vdash (residual, s, frs) \triangleright v \longrightarrow$ 
12      s  $\in$  CG,g,cinit_stores G v
13       $\wedge$  s  $\in$  Rres v
14       $\wedge$  s  $\in$  Vres v)"
```

Isabelle

Inside a proposition, $A \implies B$ is an implication and $\exists$, $\wedge$, $\vee$ are the usual connectives. Subscripts, superscripts, and symbols such as $\vdash$ and $\triangleright$ are notation for a constant applied to its arguments, and the text names the constant at first use.

A *type class* collects operations and laws that an instance proves once for a type [100]. The class `widening` fixes an operator, written infix as ∇ , after `fixes` and states its laws after `assumes`:

```
1 (* vendor/td-verification/Warrowing.thy *)
2 class widening =
3   fixes widen :: "'a :: order => 'a => 'a" (infixl "\nabla" 65)
4   assumes widen_ge1: "a <= a \nabla b" and widen_ge2: "b <= a \nabla b"
```

Here `'a :: order` restricts the type variable to types with an order, and each assumption carries a name, such as `widen_ge1`, by which proofs cite it. A type has at most one instance of each class.

A *locale* fixes parameters and assumptions in the same syntax, and interpreting it for concrete parameters proves the assumptions and yields its theorems [101]. Unlike a class, a locale can be interpreted several times for one type. The locale below fixes one operation. The attribute `[intro]` after an assumption's name registers it with Isabelle's automatic proof methods:

```
1 (* src/Abstract_Interpreter/Domain/Eval/Backward_Domain.thy *)
2 locale sound_intersection =
3   fixes intersect :: "'a::numeric_domain => 'a => 'a"
4   assumes intersect_sound[intro]:
5     "n \in \gamma a \implies n \in \gamma b \implies n \in \gamma (intersect a b)"
6     and intersect_reductive1[intro]: "intersect a b <= a"
7     and intersect_reductive2[intro]: "intersect a b <= b"
```

An *inductive definition* is the least relation closed under its rules, and `inductive_set` defines a set the same way. Each rule is a named implication, and $A \implies B \implies C$ reads “from A and B , infer C ”. *Rule induction* proves a property of every element with one case per rule. The set below has a base rule and a step rule:

```
1 (* vendor/td-verification/Basics.thy *)
2 inductive_set reach for T \sigma x where
3   base: "x \in reach T \sigma x"
4 | step: "y \in reach T \sigma x \implies z \in dep T \sigma y \implies z \in reach T \sigma x"
```

The chapters define execution relations such as `pstep` and `cstep` by rules and write the reflexive–transitive closure of a step relation $\rightarrow$ as $\rightarrow^*$.

Code equations determine what code generation emits [26]. One `export_code` declaration emits OCaml from them, and the generated code is trusted, not proved (Figure 1.4). Theories are grouped into *sessions*, which Isabelle builds and checks as units, and which the chapters cite as the home of a construction. The control-flow graph, for instance, lives in `Voblint_CFG`, which builds on `Voblint_VIMP`.

B. Notation

The symbols below are declared in the theories and used in both the formal statements and the prose. Each entry names the Isabelle declaration that introduces it and links to that declaration in the rendered theories. The table is generated from the theories, so it cannot drift from them.

Notation	Reads as	Meaning	Declared by
Semantic operators and relations			
$\gamma \ a$	gamma of a	the integers the abstract value a represents	gamma Abstract_Domain
$\llbracket e \rrbracket_e \ s$	the value of e in s	the integer expression e evaluates to in store s	aval VIMP_Expr
$\llbracket e \rrbracket^\sharp \ d$	the abstract value of e in d	the abstract value an evaluator computes for expression e in abstract state d ; it contains the concrete value whenever the store lies in $\gamma \ d$	aval_abs Forward_Domain
$\llbracket a \rrbracket_c^\sharp$	the abstract transfer of action a in component c	the state component c computes from d at an edge with action a , asking its own channel; for a sound component it covers every store the edge produces from a store d describes	abs_step MCP_Spec
$\gamma \ d$	the stores the abstract state d represents	one symbol for every kind of state, selected by the type of d : a pointwise state gives the stores s with $s \ x \in \gamma \ (d \ x)$ for every variable x , a relational state the stores satisfying its pairs, and a lifted state represents no store when it is Bot	gamma_S State_Concretization
$\mathcal{G}, \Pi \vdash c \rightarrow_p c'$ $\mathcal{G}, \Pi \vdash c \rightarrow_p^* c'$	c steps to c'	one small step of a source configuration (command, store, frame stack) under procedure table Π ; the starred arrow is its reflexive-transitive closure	pstep VIMP_Proc psteps VIMP_Proc
$\mathcal{G}, g \vdash c \rightarrow_c c'$ $\mathcal{G}, g \vdash c \rightarrow_c^* c'$	c steps to c' in g	one step of a graph configuration (node, store, frame stack) in control-flow graph g ; the starred arrow is its reflexive-transitive closure	cstep CFG_Exec csteps CFG_Exec
$\Pi, g, p \vdash c \approx c'$	c is matched with c'	the compiler's simulation relation between a source configuration c whose outermost activation runs procedure p and a configuration c' of the compiled graph g	csim Simulation_Relation

Notation	Reads as	Meaning	Declared by
$p, s_0 \vdash c \triangleright v$	from s_0 , c is observed at node v in p	some run of the elaboration of program p from store s_0 reaches an annotated configuration that erases to c , names node v of the compiled graph and is related to v by the simulation; a configuration reachable along two paths can be observed at two nodes	at_node Elaborated_Compilation
$\text{skip}^\sharp S$ $\text{assign}^\sharp S$ $\text{special}^\sharp S$ $\text{branch}^\sharp S$ $\text{body}^\sharp S$ $\text{return}^\sharp S$ $\text{event}^\sharp S$	the <i>op</i> transfer of analysis S	the edge transfers of an analysis S (dg_spec), one field per program construct, each named after the Goblint Spec method it answers to	dgs_skip DG_Spec dgs_assign DG_Spec dgs_special DG_Spec dgs_branch DG_Spec dgs_body DG_Spec dgs_return DG_Spec dgs_event DG_Spec
$\text{enter}^\sharp S$ $\text{combine_env}^\sharp S$ $\text{combine_assign}^\sharp S$	the call-boundary transfers of analysis S	entry into a callee, answering entry pairs (resume state, entry state), and the two stages of the return: the environment stage and the assignment of the result	dgs_enter DG_Spec dgs_combine_env DG_Spec dgs_combine_assign DG_Spec
$\text{combine}^\sharp \mathcal{G} \text{ dst } q \text{ r}$	the abstract return of r into q	the abstract counterpart of the concrete return on abstract states: caller locals, callee globals, and the callee's result written to the destination	combine_collect_abs Transfer_Algebra
$\text{context}^\sharp$	the computed callee context	the routing function: the callee context the equations compute from an abstract entry state	route Routed_Context
$a \nabla b$ $a \triangle b$	a widened by b , a narrowed by b	the solver's widening, an upper bound of both arguments, and its narrowing, which stays between b and a when $b \leq a$	widen Warrowing narrow Warrowing
$\mathcal{G}$	the global-variable classifier	the predicate on variable names that marks the globals; a program supplies it from its global declarations, and the theorems take it as a parameter	declared_global VIMP_Program
Semantic sets A calligraphic capital names a semantic set. Its global form lists every parameter; inside a locale or a theory's context block that fixes them, they are omitted.			
$\mathcal{T}_{\mathcal{G}, g, S}$ within activation_coverage : $\mathcal{T}$	the valid activation traces of g from S	the activation traces of graph g from initial stores S	valid_activation_trace Activation_Trace_Def
$\mathcal{C}_{\mathcal{G}, g, S} \text{ v}$ within activation_coverage : $\mathcal{C}$	the stores collected at v	the stores valid activation traces reach at node v	node_collect Activation_Trace_Collect

Notation	Reads as	Meaning	Declared by
$\mathcal{A}_{\mathcal{G}, \text{adm}, c_0, g, S} \ V \ C$ within <code>activation_coverage</code> : $\mathcal{A} \ v \ c$	the stores collected at v in context c	the stores valid activation traces carrying context c reach at node v ; adm admits callee contexts and the root runs in c_0	<code>activation_collect</code> Activation_Trace_ Context
$\mathcal{R}_{res} \ v$	the stores report res describes at v	the stores some state the analysis report holds at node v , under any of its contexts, describes	<code>report_sem</code> Analysis_Report
$\mathcal{V}_{res} \ v$	the stores the verdicts of res at v hold in	the stores in which every definite verdict the analysis report gives at node v holds: a proved condition is true, a refuted one false	<code>verdict_stores</code> Analysis_Report
Proof-local shorthands These names are local abbreviations available while writing proofs in the indicated locales. They are not new definitions: they unfold when parsed, and outside those locale contexts Isabelle shows the underlying expressions.			
<code>carries t c</code>	t carries context c	the context relation of the locale's policy, between a valid activation trace and the contexts it may run in stands for <code>activation_context_rel</code> $\mathcal{G} \ \text{adm} \ c_0 \ g$	<code>carries</code> Activation_Trace_ Abstract in <code>activation_coverage</code>
<code>admits u ctx p s es</code> <code>ctx'</code>	ctx' is admitted for the call at u	some call edge at u to p enters es from s , and the context relation admits ctx' for it stands for <code>admits_call_context</code> $\mathcal{G}$ $g \ \text{adm}$	<code>admits</code> Activation_Trace_ Abstract in <code>activation_coverage</code>
<code>cover v ctx</code>	the stores covered at v in ctx	proof-local shorthand for applying $\gamma_{\mathcal{M}}$ to the value the published reader holds at (v, ctx) , the cover handed to the coverage locale; it is not a separate semantic operation stands for $\gamma_{\mathcal{M}} \ (\text{sg} \ (\text{Inl} \ (v, \text{ctx})))$	<code>cover</code> Routed_Context in <code>routed_context</code> , input only

C. Repository and Tooling

The theories, the exported analyzer, the project site, and this thesis quote one another. The analyzer is exported from the theories, the thesis cites their names and statements, and both the site and the thesis quote the analyzer’s output. Every change to the theories can therefore make another part stale, and keeping them consistent is engineering work outside the proof. Isabelle offers comparatively little project-level tooling for this beyond its distribution. Users have pointed this out as well [102] ([reader comment](#)). The need grew with AI assistance. Agents wrote many proofs and much of the surrounding code (statement on the use of generative AI), and a reviewer cannot read every line they produce. A change therefore counts as done only when deterministic checks pass, so the checks, not the agent, decide whether the repository is consistent.

C.1. Repository contents and the vendored solver

One repository holds everything this thesis describes: the Isabelle theories, a submodule reference to the vendored TD solver, the exported analyzer with its command-line tool and playground, the regression corpus, and the sources of this thesis. Each release is archived on Zenodo with a citable DOI [25], and the project site hosts the playground, the explainer, and the rendered theories that every Isabelle name in this thesis links to.

The vendored TD solver of Tilscher et al. [4] carries four changes made for this thesis, each proposed upstream and none changing the algorithm. They restore the build under Isabelle2025-2 ([PR 15](#)), remove the stabilization assumptions from the classes `widening` and `narrowing`, which only termination proofs use ([PR 13](#), Section 5.1), rename short field and constructor names that clashed with downstream theories ([PR 14](#)), and add facts about the solver’s definitions, such as `solve_dom_of_solve_c`, that `td_certified_solver` uses ([PR 16](#), Section 9.3). At the time of writing, all four pull requests await review. The solver’s repository is private, so the development cannot yet be built from public sources alone, and there are plans to publish the solver in the Archive of Formal Proofs.

C.2. OCaml packages on conda-forge

The repository relies on several package managers. [Pixi](#) installs Python and the conda-forge packages and runs every task, `opam` installs the OCaml toolchain for the exported analyzer and its browser build, and Isabelle is installed separately. The goal is a single Pixi file from which one `pixi install` sets up everything. For the OCaml side, this requires every package the analyzer builds with on conda-forge for all supported platforms. Until recently, however, conda-forge carried the OCaml compiler and only a few tools, such as Dune and Menhir, and almost none of the libraries below them.

Alongside this thesis, considerable effort went into changing this, together with other conda-forge contributors. A [recipe for ten base libraries](#), from `csexp` to `yojson`, has been

merged, and Apple Silicon builds have been enabled for each of them. Two fixes to the Dune package have been merged, and recipes for `ppxlib` and `sedlex`, as well as an update to Zarith, are under review. `isar-tools` itself [installs from conda-forge](#). Of the 25 packages in the [toolchain manifest](#), 15 are now available on conda-forge, and the others, among them `js_of_ocaml` and `ocamlformat`, still come from opam.

C.3. Derived files and checks

When the same information is maintained by hand in several files, the copies tend to diverge after a change. The repository therefore keeps a few sources and derives everything else from them (Figure C.1). One grammar file yields the Isabelle syntax of VIMP, the analyzer’s parser, and its printer, so the unverified parser (Figure 1.4) and the syntax the proofs use come from one language description. The analysis registrations, in contrast, are written by hand. Each numeric domain registers itself in a few lines per context policy by interpreting a shared locale (Section 11.1). The product state of Chapter 7 is written by hand, with one field per analysis, each of a different type. A dependently typed logic could express such a heterogeneous structure once [103, ch. 9], but HOL’s simple types cannot, so adding an analysis adds a field and one case to each function over the analyses.

The most important derived file is the analyzer itself. Isabelle’s code generator exports `run_voblint` as OCaml, and the repository commits the exported [module](#) so that the command-line tool and the playground build without Isabelle. A committed copy can fall behind the theories, however. The continuous integration therefore regenerates the export from the built session and fails if the result differs from the committed file. Local builds record a hash of the theories the export was generated from and warn when it no longer matches, so a stale analyzer is noticed before it reaches the continuous integration.

This thesis is held to the same discipline. Every Isabelle name in its sources goes through a helper that a check resolves against the theories and links to the rendered theories. Theorem statements are lifted from the theories and compared with what the built session proves, quoted analyzer results are rerun, each VIMP listing’s playground link is decoded from the PDF and compared with the listing, and repository figures come from a measurement script. A rename, a changed statement, or a changed result therefore fails the build instead of reaching the reader.

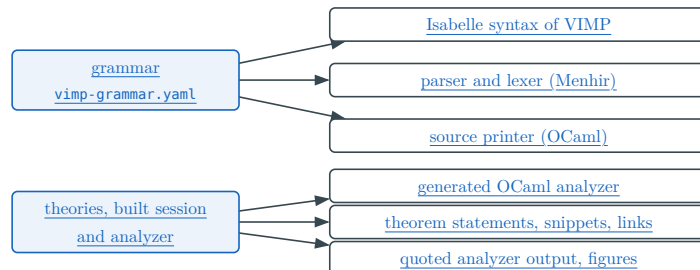


Figure C.1: The sources of the repository (blue) and the files derived from them. Each box links to its file. Each derived file is generated during the build or committed with a check that regenerates it. The link map is also checked against the published site.

C.4. Maintaining the formalization: isar-tools

Isabelle checks that every proof goes through, but a successful build says little about how well a development is maintained. A lemma that no proof cites, an import that another import already brings in, a constant written out next to the notation the project gave it, or a locale assumption that no proof uses all leave the build green. Documentation that quotes formal statements or links to the rendered theories can also fall out of date without affecting the build. In a development of 213 theories, such problems accumulate.

[isar-tools](#), written for this thesis and published separately, addresses them. The command-line tool reads theory and ROOT files the way Isabelle’s build finds them and never runs Isabelle itself, so it answers in seconds, fits a pre-commit hook, and works on theories that do not yet build. Related tools cover parts of this ground. The [Isabelle Linter](#) checks proof style inside Isabelle, and [isabelle-query](#) answers source-level queries about declarations, dependencies, and unfinished proofs.

The tool has three parts. `isar fmt` changes only layout. `isar check` offers about fifty checks in groups. Checks for unfinished proofs and unreachable theories run by default, and checks for unused or duplicated lemmas, unused locale assumptions, or terms written out next to their notation run when named. This repository runs the default groups and several named ones, among them documentation coverage and the locale check below. `isar project`, the part that matters most for this thesis, exposes sessions, theories, and declarations to other tools. `isar project extract` keeps the quoted theorem statements in sync with their sources, `isar project notation` generates the notation tables of the thesis and the explainer, and `isar project anchors` resolves the links into the rendered theories. A change that would leave the thesis inconsistent with the formalization therefore fails the build.

Some findings concern mistakes that Isabelle accepts. Inside a locale header, Isabelle reads an unknown identifier as a free variable and generalizes it. An assumption that cites a renamed or misspelled constant therefore still builds, but it now quantifies over an arbitrary function and states a different specification. In the theory below, the assumption cites a misspelling of the constant defined above it, and `isar check locales` reports the identifier.

```
1 definition gamma_sign :: "int ⇒ int set" where
2   "gamma_sign x = {x}"
3 locale sound_domain =
4   fixes step :: "int ⇒ int"
5   assumes sound: "x ∈ gamma_sgn y ⇒ step x ∈ gamma_sign (step y)"
```

Demo.thy:8:27: locale-free-variable: gamma_sgn in the header of locale sound_domain is no parameter, bound variable, or name used elsewhere; Isabelle generalizes over it as a free variable

A test reruns the check on this [theory](#). The check approximates Isabelle’s inner syntax lexically, so its report is a warning to inspect and does not prove an error.

A second kind of finding concerns build time. A session inherits the heaps of its parent sessions, but a library theory outside that ancestry is elaborated again in every session that imports it. `isar stats build` reads the time each theory took from a verbose build log and reports such repetitions, and continuous integration enforces a budget per library session. In this repository, repeated elaboration once cost 19% of a clean build.

All of these checks work on the sources and complement Isabelle. They point to suspicious patterns and structural problems, and they may report false positives or miss problems. Whether a theory is formally correct is still decided by the Isabelle build alone.