Voblint: End-to-End Soundness of a Context-Sensitive Goblint-Style Analyzer in Isabelle/HOL

Manuel Lerchner
Technical University of Munich, Germany
manuel.lerchner@tum.de

Preprint

hexagon:2611.00109v1

Abstract

Goblint-style analyzers solve side-effecting equations, where evaluating one unknown can contribute to other unknowns. An existing Isabelle/HOL proof establishes solver partial correctness without connecting the equations to execution. We mechanize this connection for a scalar imperative language with recursive procedures but no pointers or heap memory.

We adapt the local traces of Schwarz et al. to record procedure activations and the callers resumed by returns. A context policy relates concrete calls to one or more analysis contexts. A cover assigns sets of stores to program points and contexts as candidate overapproximations of reachability. We prove that five local obligations suffice for a cover to contain every valid trace's final store in every admitted context, on any control-flow graph. A post-solution assigns abstract values that bound each equation's result and side effects. When the abstract operations are sound and the routing agrees with the context policy, every post-solution induces a cover. The verified solver computes a post-solution when it returns.

For every accepted configuration, whenever the Isabelle-defined analyzer returns a report, it overapproximates the stores reached by finite source executions from the prescribed initial states. Whenever execution reaches a check, PROVED guarantees that its condition holds, and REFUTED guarantees that it fails. Neither verdict establishes reachability. This source-level guarantee applies to the Isabelle-defined analyzer; solver termination, parsing, and correctness of generated code are not established.

Keywords: abstract interpretation, interprocedural analysis, calling contexts, side-effecting constraint systems, verified static analysis, Isabelle/HOL.

Related version: Master's thesis [19].
Software: Isabelle/HOL formalization and analyzer [18], https://doi.org/10.5281/zenodo.23260696.
© Manuel Lerchner. Licensed under CC BY 4.0.

1 Introduction

A context-sensitive static analyzer can distinguish abstract states for different calling contexts at the same program point. A context groups calls, for example by their call sites or abstract argument values. In the program of fig. 1, `main` calls `bump(5)` and `bump(4)`, and `bump` returns its argument plus one. An interval analysis without calling contexts must describe both calls at the return of `bump` and therefore cannot prove either check. With contexts distinguishing the arguments, the analysis handles each call separately and proves both checks. Goblint analyzes C [38, 32] using such contexts through a *side-effecting constraint system* [31, 1]. Its inequations relate unknowns indexed by program point and context, and the right-hand side of

one unknown may also contribute to other unknowns. A generic top-down solver starts from a queried unknown and discovers dependencies as it evaluates right-hand sides. A defect in any of these layers can produce an unsound verdict. Section 7 examines one such defect in Goblint's remainder operation.

Stade et al. proved the solver partially correct in Isabelle/HOL [34], and Tilscher et al. extended the proof to side effects [35]. A valuation assigns an abstract value to each unknown. When the solver terminates, its valuation satisfies the inequations for all solved unknowns. It is a *post-solution* on that set. The solved set is closed under right-hand-side reads of local unknowns: reading a local unknown requires it to be solved too. These properties form the solver's certificate. To derive program correctness from it, we must prove that the equations soundly approximate execution and that the report preserves their guarantee. The semantic connection requires associating reachable stores (assignments of integer values to variables) with calling contexts. Such semantics are established in the literature [33, 16, 14, 12] (section 8). To our knowledge, no mechanization connects a source semantics through such a context-indexed semantics to a side-effecting constraint system and its verified solver.

We mechanize this connection in Voblint (verified Goblint), an Isabelle/HOL formalization.

- We adapt the local traces of Schwarz et al. [29] to procedure activations, each representing one invocation of a procedure. A context policy, which may admit several contexts for one call, relates these activation traces to contexts. A candidate overapproximation, or *cover*, assigns sets of stores to nodes and contexts. We give five local obligations on covers for arbitrary control-flow graphs. We prove that any cover satisfying them contains the final store of every valid trace in every admitted context (section 3).

- We prove that every post-solution of the generated, context-routed equations meets these obligations, assuming sound abstract operations and agreement between the context policy and the routing that selects callee contexts. The verified solver enters only through its certificate (section 4).

- We prove an end-to-end theorem about the analyzer function that Isabelle exports to OCaml. For every accepted configuration, whenever the analyzer returns a report, the report covers every store reached by a finite source execution at its program point. Every definite verdict correctly classifies a check as true or false whenever execution reaches it (section 5).

- We exhibit concrete instances of the theorem, show that the faulty remainder operation behind a Goblint defect violates a domain obligation, and compare the verdicts obtained with different calling contexts on recursive programs (section 7).

The solver, constraint systems, local traces, and separation of analyses and solvers are inherited. Our context-indexed collecting semantics groups reachable stores by node and context. The report guarantees cover both individual contexts and their union at each node (section 5).

VIMP supports local and global integer variables and recursive procedures with parameters and return values, but no pointers, arrays, or heap. The correctness results are conditional on solver termination, which is not proved for every program. The parser and report printer are unverified handwritten code; Isabelle's code generator, the OCaml toolchain, and the solver-tracing hook used in traced runs are trusted.

2 Overview: One Call Through the Pipeline

We follow the program of fig. 1 through the analyzer, with intervals and entry-state contexts: calls are grouped by the abstract values of the callee's parameters at entry. The source program is compiled to a procedure-aware control-flow graph (CFG). Each procedure p has an entry node $\mathsf{FunctionEntry}\ p$ and a result node $\mathsf{FunctionResult}\ p$. The graph g has two kinds of edges. An

```
fun bump(n) {
  return n + 1;
}

fun main() {
  a = bump(5);
  b = bump(4);
  __voblint_check(a == 6);
  __voblint_check(b == 5);
}
```

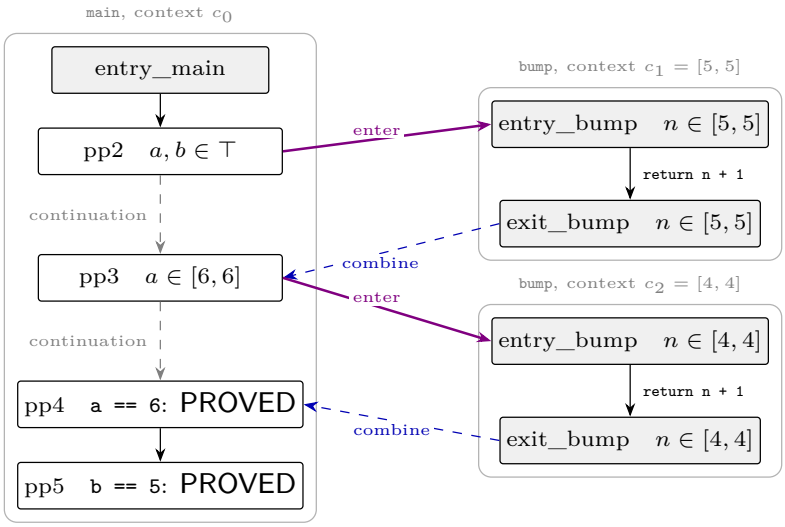


Figure 1: The running example. Left: the VIMP program. Right: its graph as the analyzer solves it with intervals and entry-state contexts. Each frame is one procedure in one context; `bump` gets one copy per entry state. Solid violet arrows enter callees; dashed blue arrows combine caller and callee states on return, and dashed gray arrows link a call site to its continuation. Both checks are PROVED. Without contexts, the single copy of `bump` joins both calls and both checks are UNKNOWN: the analysis cannot establish whether their conditions hold or fail. The states shown are analyzer output using warrowing, an update rule combining widening and narrowing for global unknowns (section 4). Here $\top$ allows any integer. The drawing omits the intermediate node of `bump`'s body and the unreachable node after `return`.

intraprocedural edge $(u, a, v) \in \mathit{intra}\ g$ carries an action a, such as an assignment or a guard. A *call* edge $(u, e, \mathsf{FunctionEntry}\ p, k) \in \mathit{calls}\ g$ leads from the call site u into p and names the continuation k at which the caller resumes. Here e describes the call, including its arguments and result destination. A predicate $\mathcal{G}$ tells which variables are global. A proved forward simulation relates every source execution to a graph execution with the same store (section 5).

From the graph, the analyzer generates inequations over *local unknowns* (v, c) for a node v and a context c, and over *global unknowns* without a program point. These global unknowns hold shared analysis information, including callee entry states; they are distinct from program global variables. At `a = bump(5)`, the analysis's *enter* operation binds `n` to $[5, 5]$, and this entry state is *routed* to the callee context $c_1 = [5, 5]$; the second call routes to $c_2 = [4, 4]$. Section 4 gives the equations of a call. For these two calls, the analyzer maintains separate context states and proves both checks.

To interpret the abstract state in c_1, we need the concrete stores associated with that context. Activation traces and a context policy define this association (section 3). The trace of the activation `bump(5)` records its caller, the policy admits c_1 for that call, and the *activation collecting semantics* $\mathcal{A}(v, c)$ collects the final stores of the traces that end at v and carry c. The *node collecting semantics* $\mathcal{C}(v)$ collects them regardless of context. An analyzer report *res* defines two sets at each node v. Its report set $\mathcal{R}_{res}(v)$ is the union of the stores represented by its abstract states at v across all contexts. The verdict set $\mathcal{V}_{res}(v)$ contains the stores in which every definite verdict at v holds: the check condition is true for PROVED and false for REFUTED. Neither verdict asserts reachability (section 5). The soundness argument is a chain of inclusions at every node:

$$\text{reached stores} \subseteq \mathcal{C}(v) = \bigcup_c \mathcal{A}(v, c) \subseteq \mathcal{R}_{res}(v) \subseteq \mathcal{V}_{res}(v). \tag{1}$$

Each inclusion may add unreached stores, which costs precision, but none may lose a reached

store. Section 5 proves the outer inclusions.

3 Activation Traces and the Coverage Contract

An abstract state at node v in context c needs a concrete interpretation. We adapt local traces [29] to retain activations and callers, then relate them to contexts, including call histories [33]. The coverage contract is independent of abstract domains, equations, and solvers.

A graph execution with a plain frame stack does not record contexts. A frame holds only what is needed to resume the caller and disappears at the return. Frames could carry a ghost context, an annotation used only in the proof (section 6). We keep contexts out of the base trace semantics and relate them to traces separately. Each call starts a new activation linked to its caller; a return combines the completed callee trace with the caller trace and resumes the caller.

3.1 Activation traces

An *activation path* is the list of node–store pairs visited by one activation. The constructors of an activation trace describe how an activation begins or continues (`activation_trace`). $\mathsf{Root}\ \pi$ records the activation of `main`. $\mathsf{Call}\ t'\ \pi$ records a callee activation and retains the caller trace t' at the call site. $\mathsf{Resume}\ t'\ t''\ \pi$ records the caller's continuation after the callee trace t'' has completed. The observers *path_of* t, *sink_node* t, and *sink_store* t return the path, its last node, and its last store, respectively. The observer *caller_of* t returns the creating caller trace wrapped in *Some*, or *None* for the main activation. The operation *extend* $t\ (v, s)$ appends a step to the path.

The datatype admits terms that no execution produces. Let $entry_g$ be the entry node of `main`. The set $\mathcal{T}_{\mathcal{G},g,S}$ of *valid* traces of graph g from initial stores S is defined by four rules mirroring graph execution (`valid_activation_trace`). We omit the fixed subscripts below; @ concatenates lists and # prepends an element. A rule derives the conclusion below its line from the premises above it:

$$\frac{s \in S}{\mathsf{Root}\,[(entry_g, s)] \in \mathcal{T}}$$

$$\frac{t \in \mathcal{T} \quad (sink_node\ t, a, v) \in intra\ g \quad s' \in step_a(sink_store\ t)}{extend\ t\ (v, s') \in \mathcal{T}}$$

$$\frac{t \in \mathcal{T} \quad (sink_node\ t, e, \mathsf{FunctionEntry}\ p, k) \in calls\ g}{\mathsf{Call}\ t\ [(\mathsf{FunctionEntry}\ p,\ enter_e(sink_store\ t))] \in \mathcal{T}}$$

$$\frac{\begin{array}{c} t'' \in \mathcal{T} \quad caller_of\ t'' = Some\ t' \quad sink_node\ t'' = \mathsf{FunctionResult}\ p \\ (sink_node\ t', e, \mathsf{FunctionEntry}\ p, k) \in calls\ g \end{array}}{\mathsf{Resume}\ t'\ t''\ \big(path_of\ t' \mathbin{@} [(k, combine_e(sink_store\ t', sink_store\ t''))]\big) \in \mathcal{T}}$$

Here $step_a$ is the concrete meaning of an action, $enter_e$ binds the callee's formals to the call's arguments and resets its locals, and $combine_e$ keeps the caller's locals, takes the callee's globals, and assigns the return value to the call's destination (`combine_collect`). A return resumes the activation identified by *caller_of*. The traces are defined for arbitrary graphs, so the return rule takes *some* call edge from the caller's call site into p. It does not require the edge through which the callee was entered. On compiled graphs every call site has exactly one call edge (`calls_source_unique`), so the two coincide.

Every finite source execution of a well-formed program yields a valid trace that ends at the corresponding node with the same store, and whose chain of callers represents the runtime frame stack (`source_run_has_activation_trace`). The final stores of valid traces, collected per node, form the node collecting semantics $\mathcal{C}_{\mathcal{G},g,S}(v)$ (`node_collect`).

The constructors follow patterns from the local traces of Schwarz et al. [29]: initial traces, thread creation, and binary lock acquisition incorporating a second trace. Voblint adapts these patterns to procedure entry and return (section 8).

3.2 Calling contexts as a relation

Call-string contexts record the last k call sites; entry-state contexts record the abstract values of the formals at entry. A *context policy* specifies which callee contexts a concrete call may belong to. It maps a call site u, a caller context c, a description of the call, the caller's store, and the entered store to the *set* of admitted callee contexts (`context_policy`). The result is a set because Goblint's enter may return several alternatives for one call, each routed separately. The set may be empty, for instance at a call the analysis considers unreachable. The supplied analyses return a single entry alternative, so a call is admitted in at most one context per caller context. A Sign-analysis instance demonstrates the relational case: its enter returns two overlapping alternatives, the plain entry state and one that forgets the formals. One concrete call is then admitted in two contexts (`ov_two_contexts_admitted`).

For a call edge e into p, we write $adm_{u,c}(e, s, s')$ for the set the policy returns for the call described by e and p, caller store s and entered store s'. We write $c' \in adm^{\exists}_{u,c}(p, s, s')$ when some call edge from u into p maps s to the entered store s' and the policy admits c' for it (`admits_call_context`). For a fixed initial context c_0, the contexts a trace carries are defined inductively (`activation_context_rel`):

$$\frac{}{\mathsf{Root}\ \pi\ \text{carries}\ c_0} \qquad \frac{t'\ \text{carries}\ c}{\mathsf{Resume}\ t'\ t''\ \pi\ \text{carries}\ c}$$

$$\frac{t\ \text{carries}\ c \quad c' \in adm^{\exists}_{sink_node\, t,\, c}(p, sink_store\ t, s')}{\mathsf{Call}\ t\ ((\mathsf{FunctionEntry}\ p, s') \# \pi)\ \text{carries}\ c'}$$

That is, the root carries the initial context c_0, a callee carries every context admitted for its call from some context of its caller, and a resumed activation keeps the contexts it had before the call. Traces store no context, so the policy is a parameter of every theorem below. The activation collecting semantics collects the final stores of the valid traces that end at v and carry c (`activation_collect`):

$$\mathcal{A}_{\mathcal{G},adm,c_0,g,S}(v, c) \;=\; \{\, sink_store\ t \mid t \in \mathcal{T}_{\mathcal{G},g,S},\ sink_node\ t = v,\ t\ \text{carries}\ c \,\}.$$

In the running example, the final store of `bump(5)` lies in $\mathcal{A}(\mathsf{FunctionResult}\ \texttt{bump}, [5,5])$ and that of `bump(4)` in $\mathcal{A}(\mathsf{FunctionResult}\ \texttt{bump}, [4,4])$, whereas $\mathcal{C}$ at that node contains both.

3.3 The coverage contract

A *cover* assigns a set of stores to each node and context. The four rules for valid traces give rise to four preservation obligations on a cover. A fifth obligation ensures that each call from a covered store is admitted in at least one context. We express this contract in the locale `activation_coverage`; an Isabelle locale packages named parameters and assumptions [2]. Write $cover(v, c)$ for the assigned set. The obligations are as follows, with the call obligations

quantified over every call edge $(u, e, \mathsf{FunctionEntry}\ p, k)$:

INIT	$s \in S \implies s \in \mathit{cover}(\mathit{entry}_g, c_0)$
INTRA	$(u, a, v) \in \mathit{intra}\ g,\ s \in \mathit{cover}(u, c),\ s' \in \mathit{step}_a(s) \implies s' \in \mathit{cover}(v, c)$
CALL	$s \in \mathit{cover}(u, c),\ c' \in \mathit{adm}_{u,c}(e, s, \mathit{enter}_e(s))$ $\implies \mathit{enter}_e(s) \in \mathit{cover}(\mathsf{FunctionEntry}\ p, c')$
RETURN	$s \in \mathit{cover}(u, c),\ c' \in \mathit{adm}^{\exists}_{u,c}(p', s, s'),$ $t \in \mathit{cover}(\mathsf{FunctionResult}\ p, c') \implies \mathit{combine}_e(s, t) \in \mathit{cover}(k, c)$
TOTAL	$s \in \mathit{cover}(u, c) \implies \mathit{adm}_{u,c}(e, s, \mathit{enter}_e(s)) \neq \emptyset$

RETURN ranges over every procedure p' and entered store s' admitted at the call site. As in the return rule, the call edge e used to resume the caller can differ from the edge used to enter the callee. On a graph with one call edge per call site, $p' = p$ and both operations use the same edge.

If no context admits a call from a covered store, CALL and RETURN hold vacuously. They therefore do not ensure that the cover at the continuation includes the returned store, even though the resumed caller still carries its contexts there. Without TOTAL, a cover that meets the other four obligations can miss a reached store (`total_dropped_unsound`). TOTAL asks only about covered stores, because an entry-state policy admits no context for stores the analysis has excluded. Call strings and the unit policy, which admits one context for every activation, meet it for every cover (`call_context_total_on_of_fun`).

Theorem 1 (Context-indexed soundness, `activation_collect_sound`)**.** *If a cover meets the five obligations of* `activation_coverage`*, then* $\mathcal{A}(v, c) \subseteq \mathit{cover}(v, c)$ *for every node* v *and context* c*.*

Proof idea. The proof inducts over valid traces. The invariant states that a trace carries some context and its final store is covered in every context it carries. A single-trace invariant does not suffice for the return rule, which needs the invariant for the caller recovered through *caller_of*. The induction therefore carries the invariant along the whole chain of callers (`caller_chain_covered`, an instance of the induction principle `caller_chain_closure`).

TOTAL is used twice. In the call case, it supplies a context for the new callee. In the return case, it supplies an admitted callee context c' for each context c of the resumed caller, using the callee's own entry edge. The callee then carries c' (`activation_context_rel_caller_entryI`), so its exit store is covered at c'. RETURN ensures that combining the caller store and callee exit store yields a store in the cover at the continuation in context c. This is why RETURN ranges over the admitted entry (p', s'), which can differ from the entry specified by the call edge used for resumption.

Under the contract, every valid trace therefore carries at least one context, so taking the union over contexts recovers the node collecting semantics (`node_collect_eq_Union_activation_collect`):

$$\mathcal{C}(v) \;=\; \bigcup_c \mathcal{A}(v, c).$$

4 From Equations to a Cover

The equations follow Goblint's `Spec` interface [38, 32] and side-effecting constraint systems [31, 1]. We show how sound abstract operations, context routing, and the solver certificate [35] establish the five coverage obligations.

4.1 The analysis interface

An analysis describes the stores at a node by an abstract state d. The concretization $\gamma(d, \rho)$ is the set of stores it represents, parameterized by ρ, the solved values of the analysis's global unknowns. We suppress ρ when it plays no role. The analysis supplies the record `dg_spec`, the counterpart of Goblint's `Spec`: one transfer per kind of edge action, an enter operation, and two combine operations. A transfer computes an abstract successor state. A call transfer retains the caller's state alongside the callee's entry state. After the return, the caller's locals come from before the call, while the globals and the result come from the callee. Enter therefore returns a list of entry pairs (q, d): a resume state q describing the caller's store and an entry state d describing the store in which the callee starts. After the return, *combine_env* merges the callee's exit state into q, taking the callee's globals, and *combine_assign* then writes the return value to the call's destination. Together they mirror the concrete $combine_e$ of section 3. A transfer may also publish contributions, meaning abstract values that must be included in the analysis's global unknowns. We interpret its output using the join of the previous global values and these contributions.

The local soundness laws of transfers and combine form the locale `analysis_contract`: the concretization is monotone, each transfer covers every concrete successor store, and combine covers $combine_e(s, t)$ whenever q covers s and the exit state covers t. The routing conditions below require enter to return a pair covering the caller's store and the entered store (`entry_pairs_cover`). Each context policy must satisfy these conditions, which are separate from `analysis_contract`. A numeric domain proves only its abstract operations sound. The framework derives the transfers once for all domains, following Nipkow and Klein's generic interpreter [25, Sect. 13.5]. The implemented domains are Sign, Interval, Parity, Congruence, and Int, a reduced product that combines all four and propagates information between them.

4.2 Context-indexed equations and routing

Write $\sigma(v, c)$ for the value assigned to local unknown (v, c) by the solved valuation. The unknown's right-hand side joins four sources (`routed_node_rhs_parts`):

- the initial state at the program entry;
- the transfer along each intraprocedural edge into v, in the same context c;
- the combined result of each call returning to v;
- at a procedure entry, the entry states published to it.

At a call, a *routing function route* chooses the callee context from the call site, the caller context, the entry state and the call. Consider a call edge $(u, e, \mathsf{FunctionEntry}\ p, k)$ in caller context c. For each entry pair (q, d) returned by enter for $\sigma(u, c)$, the continuation's right-hand side includes a contribution (`routed_call_alternative_program`). If d is bottom ($\bot$, representing no stores), the contribution is $combine(q, \bot)$. Otherwise, let $c' = route(u, c, d, e)$; the contribution is the program

$$\text{publish } d \text{ to } seed(p, c'); \quad r \leftarrow \text{read } (\mathsf{FunctionResult}\ p, c'); \quad combine(q, r).$$

This program is a sequence of solver operations, not source-language code. Here $seed(p, c')$ is a global unknown accumulating entry states for p in c'. The entry equation of $(\mathsf{FunctionEntry}\ p, c')$ reads it. In the running example, the call `bump(5)` in c_0 publishes $n \in [5, 5]$ to $seed(\texttt{bump}, [5, 5])$, reads the result of `bump` in $[5, 5]$, and combines it with the resume state of `main`. The same equation construction handles recursive calls, which can introduce cyclic dependencies.

The locale `routed_context` connects abstract routing to the concrete policy *adm* through adequacy and totality.

Hypothesis	Reading
`analysis_contract`	transfers and combine are sound
`pp` (`post_bounded`)	the query is in W, and at each unknown of W, σ bounds the right-hand side and every side effect it publishes
`sg_cov`, `sg_uncov`	the cover is $\gamma(\sigma(v,c),\rho)$ on W and $\emptyset$ elsewhere
`fwd`	an intraprocedural edge from an unknown of W with a non-empty state leads into W
`comb_fwd`	a call's continuation is in W in the caller's context
`routing_adequate`, `routing_total`	the policy agrees with the routing
`resolve_sound`	callee resolution lists every procedure a covered store can call
`calls_unique`	each call site has one call edge (`calls_source_unique`)
`is_bot_sound`, `is_bot_bot`	a state classified as bottom describes no store, and $\bot$ is classified as bottom
`cmb_wf`, `extra_wf`	the generated call and seed contributions are well-formed programs over unknowns
`finC`, `finE`	the graph has finitely many edges
seed keys	seed unknowns differ from all other global unknowns

Table 1: Hypotheses of theorem 2, stated over a set W of local unknowns: all assumptions of the locales `routed_context` and `dg_context_activation`. The theorem adds three premises: the program entry is in W in context c_0, the initial state describes the initial stores under initial global values ρ_0, and $\rho_0 \leq \rho$.

Adequacy (`routing_adequate`). If *adm* admits c' for a call from a store that the caller's solved state describes, then enter produces a pair (q,d) routed to c'. The pair's resume state covers the caller's store, and its entry state covers the entered store. The callee's entry unknown in c' is solved.

Totality (`routing_total`). *adm* admits at least one context for every call from a store that the caller's solved state describes.

For the unit policy and for call strings, *adm* admits exactly the context the routing function computes from the call site and the caller context. Totality is then immediate, and adequacy reduces to entry coverage and the routed entry unknown being solved.

For entry-state contexts, the policy `routed_entry_context_rel` is defined from the solved valuation σ:

$$adm_{u,c}(e,s,s') \;=\; \{\, route(u,c,d,e) \mid (q,d) \in enter_e(\sigma(u,c)),\; s \in \gamma(q),\; s' \in \gamma(d) \,\}.$$

In words, a context is admitted when an entry pair of the solved caller state covers both concrete stores and is routed there. We first fix σ and then define the policy. This policy determines which contexts contain each store, while their union $\mathcal{C}$ remains unchanged. Section 8 compares this with a posteriori soundness [22]. For entry-state contexts, $\mathcal{A}(v,c)$ therefore depends on the analysis that produced σ.

Theorem 2 (Routed collecting soundness, `activation_collect_dg_sound`)**.** *Under the hypotheses of table 1, if* $(entry_g, c_0) \in W$*, the initial state describes the initial stores under initial global values* ρ_0*, and* $\rho_0 \leq \rho$*, then* $\mathcal{A}(v,c) \subseteq cover(v,c)$ *for all* v *and* c*, where* $cover(v,c) = \gamma(\sigma(v,c),\rho)$ *on* W *and* $\emptyset$ *elsewhere.*

The proof checks the five obligations of theorem 1 for this cover. The initial state and $\rho_0 \leq \rho$ give Init. The edge transfers and `fwd` give Intra. Call follows from adequacy and entry coverage. Return follows from adequacy and combine soundness. Total follows from totality. Both bounds of `pp` are needed because a value reaches its target unknown as an answer or as a published side effect, such as an entry state published to a seed. Here `calls_unique` identifies the admitted and returning call edges.

4.3 The solver certificate

Theorem 2 requires closure under execution steps. The verified top-down solver [35] instead provides `part_post_solution` for its query x, solved set V, and result σ. This certificate states that $x \in V$ and V is closed under right-hand-side reads of local unknowns. The result σ bounds the right-hand side and every side effect of each solved unknown.

The solver works backward from its query, the result of `main`, while theorem 2 follows executions forward from the entry. The bridge takes W to be the *live* part of V: the unknowns (u, c) such that u belongs to a procedure q whose result ($\mathsf{FunctionResult}\ q, c$) is in V (`live_unknowns`). The compiler ends every body with a return edge, so every node of a procedure reaches the procedure's result through intraprocedural edges and call-to-continuation steps. The result's right-hand side reads its predecessors, so closure under reads puts all nodes of a procedure into V once its result is in V in that context.

The same closure argument applies to callees. For a non-bottom routed entry state, $(k, c) \in V$ reads ($\mathsf{FunctionResult}\ p, c'$) (`sol_vars_callee_result`). The callee's nodes are therefore live in c', including its entry. If the entry state is bottom, no concrete store enters the callee, so this case requires no callee coverage. `live_unknowns_cover` proves, from a well-formed program and a completed solver run, that the live set contains the program entry and satisfies the closure hypotheses of table 1.

Structural reachability also holds for an activation that never returns, because the argument follows the equations, not executions. Theorem 2 applies to the live subset W of the solved unknowns V. The report publishes the states associated with all unknowns in V, so it can include more states than the proof requires. This preserves soundness because the report only has to contain the cover.

The rest of the proof never inspects the solver. The analysis locale takes it as a parameter with four assumptions (`certified_solver`). A run satisfying its domain predicate, which identifies terminating inputs, yields `part_post_solution` and a finite solved set. A returned executable run lies in that domain and agrees with the logical solver. These facts are proved for the vendored solver under each of its five update rules for global unknowns (`td_certified_solver`). An update rule decides how contributions to a global unknown accumulate: by join, by join per contributing right-hand side, or by *warrowing*, which combines widening and narrowing [35]. Any solver satisfying this certificate can reuse the same semantic soundness proof.

5 The End-to-End Theorem

The solved valuation bounds the context-indexed collecting semantics of the CFG. A forward simulation connects source and graph executions, following the CompCert methodology [20]. Report construction then preserves coverage and gives sound verdicts.

From source to graph. VIMP has a small-step semantics `pstep` over configurations (r, s, frs) of a residual command (the code left to execute), a store, and a frame stack. A call pushes a frame holding the caller's store and destination and continues with the callee's body followed by a *Restore* marker; the marker pops the frame and combines the stores as in section 3 (`pstep`, rules `Call` and `RestoreStep`). A call whose arity does not match, or whose callee is undeclared, is stuck; well-formedness excludes such programs.

The compiler uses continuations to translate each procedure body into a procedure-aware CFG. A forward simulation matches each source step with graph steps that reach the same store (`csim_star`).

A source configuration does not determine a graph node. The residual command can occur in both branches of `if (b) {x = 1;} else {x = 1;}`. The proof therefore annotates every command of the program with its nodes. The relation $p, s_0 \vdash (r, s, \mathit{frs}) \rhd v$ (`at_node`) holds when some run of the annotated program from s_0 reaches an annotated configuration that erases

to (r, s, frs), names v, and is related to v by the simulation. In the example, the configuration after either branch is observed at both branch nodes. The theorem quantifies over every node at which a configuration is observed.

Combining the simulation with the result of section 3 shows that every store reached by a finite source execution lies in $\mathcal{C}$ at a node related to that execution (`source_reaches_node_collect`). For the annotated program, this holds at every node where the execution is observed (`csim_point_compile_prog`).

From valuation to report. The analyzer function `run_voblint` takes a configuration and a program. The configuration selects the analyses, the update rule for global unknowns, the context policy and call-string length, and the placement of program globals. The function rejects an invalid configuration or an ill-formed program, and otherwise runs the executable solver and returns Analysed *res* when the solver returns a result (`run_voblint`). At each node, the report lists the states computed for its solved contexts and classifies each check as PROVED, REFUTED, UNKNOWN (undetermined), or DEAD (unreachable). The sets $\mathcal{R}_{res}(v)$ and $\mathcal{V}_{res}(v)$ interpret these states and verdicts as in section 2.

In the Isabelle statement below, `fixes` declares typed parameters, `defines` introduces abbreviations, `assumes` lists premises, and `shows` gives the conclusion. Here Π is the procedure table of p, and $\mathcal{G}, \Pi \vdash (c, s, frs) \rightarrow_p^* (c', s', frs')$ is a finite execution of `pstep` from residual command c, store s and frame stack *frs*. $\mathcal{C}_{\mathcal{G},g,S}\, v$ is the node collecting semantics of the compiled graph g from the initial stores S.

Theorem 3 (End-to-end soundness, `run_voblint_source_sound`)**.**

```
theorem run_voblint_source_sound:
  fixes p :: imp_prog and s0 s :: store
  defines G_def: "𝒢 ≡ declared_global p"
      and Pi_def: "Π ≡ prog_table p"
      and g_def: "g ≡ prog_cfg p"
  assumes s0: "s0 ∈ cinit_stores 𝒢"
      and run: "𝒢, Π ⊢ (main_body Π, s0, []) →ₚ* (residual, s, frs)"
      and ans: "run_voblint config p = Analysed res"
  shows "(∃v. p, s0 ⊢ (residual, s, frs) ▷ v)
       ∧ (∀v. p, s0 ⊢ (residual, s, frs) ▷ v ⟶
                s ∈ 𝒞_{𝒢,g,cinit_stores 𝒢} v
              ∧ s ∈ ℛ_res v
              ∧ s ∈ 𝒱_res v)"
```

The theorem assumes an initial store with zero-valued globals and unconstrained locals, a finite execution from the body of `main`, and a returned analyzer report. This includes finite prefixes of nonterminating executions. The reached configuration is observed at some node; at every such node, its store is collected, covered by the report, and consistent with every definite verdict. The proof combines contexts using $\mathcal{C}(v) = \bigcup_c \mathcal{A}(v, c)$ and theorem 2.

Under the configured policy, the report's states for c at v also cover $\mathcal{A}(v, c)$ (`run_voblint_context_covers`). The underlying `run_voblint_contexts` theorem additionally ensures that every valid activation trace has an admitted context.

A returned report entails solver termination, program well-formedness, and coverage via `live_unknowns_cover`. No separate premises restrict the accepted configurations.

Meaning of verdicts. PROVED states that whenever a covered execution reaches the check, the condition holds; REFUTED that it fails (`run_voblint_check_sound`). Neither asserts that the check is reachable, and REFUTED is not a verified counterexample. DEAD establishes that the check's node is unreachable (`run_voblint_dead_check_unreached`). Analogous theorems cover arithmetic diagnostics. If a node has no diagnostic, divisors in its outgoing edge expressions and call arguments are nonzero in every collected store (`run_voblint_arithmetic_safe`). An error diagnostic establishes that its divisor is zero in every collected store at that node (`run_voblint_arithmetic_refuted`).

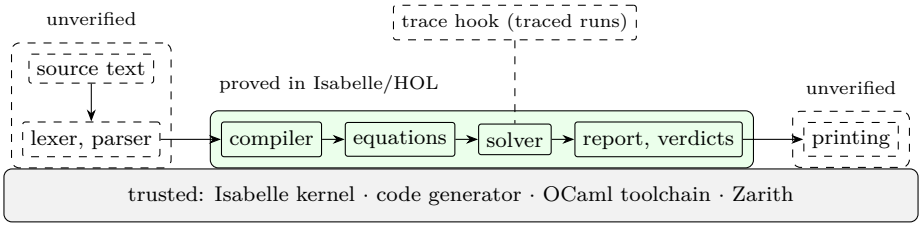


Figure 2: The trust boundary of the delivered analyzer, for programs for which the solver terminates. Dashed boxes are handwritten code outside the proof. For traced runs, the solver, routing functions, and report construction also call a trusted hook, which must return without exceptions and leave their results unchanged.

Trusted components. Figure 2 separates what is proved from what is trusted. The theorem is about the constant that Isabelle's code generator [10] exports to OCaml and that the command-line tool calls. The lexer and parser, which construct the program, and the handwritten OCaml that prints the report lie outside the proof. The Isabelle kernel, the code generator, the OCaml toolchain, and the arbitrary-precision integer library Zarith are trusted. When tracing is requested, the command-line tool calls `run_voblint_traced`, which is equal to `run_voblint` in Isabelle (`run_voblint_traced_eq`). Its exported code maps `trace_event`, whose logical result is the unit value (), to the handwritten hook `Solver_trace_hook.emit`. The solver, routing functions, and report construction call this hook. The traced executable additionally trusts it to return without exceptions and leave the computed results unchanged; untraced runs do not call it.

We must also justify that `pstep` models the intended language; the soundness theorem does not establish this. Some of VIMP's choices differ from C. Integers are unbounded. Division by zero yields 0 and the remainder by zero yields the dividend. Expressions are pure and total, so operand evaluation order does not affect the values of `&&` and `||`. A failed check does not stop the execution, so a verdict for one check does not depend on an earlier one. The locals of `main` start unconstrained, while a callee's locals start at 0. `min`, `max`, and the nondeterministic input are built-in calls, and `main` has no explicit `return`. A verdict therefore does not always hold for a C program with the same text.

6 Mechanization

Table 2 lists the size of the development [18] by layer. The development reuses part of the verified solver of Tilscher et al. [35]. The solver is vendored with four changes made for this work and proposed upstream. They restore the build under Isabelle2025-2, remove from the widening and narrowing classes the stabilization assumptions, which require update sequences to stop changing and are used only in termination proofs, rename clashing names, and add facts such as `solve_dom_of_solve_c`. None changes the algorithm. Neither Voblint nor the imported solver theories contain `sorry` or `oops`, commands that admit or abandon a proof. Code generation emits 13,109 lines of OCaml. The handwritten OCaml implementation contains 4,922 lines for parsing, report printing, and the trace hook.

Design choices and their alternatives. Besides the relational context policy (section 3) and the solver as a parameter (section 4), two choices shaped the proof.

- *Activation traces as the soundness target.* A first prototype used whole-program executions with calls inlined. The prototype's proof spliced a callee execution into the caller's run, which was hard to extend to recursion. An unimplemented alternative would annotate graph-semantics stack frames with ghost contexts and maintain an invariant over the stack.

Layer	Contents	Theories	Lines	Proved
`Program_Model`	source semantics, CFG, compiler, traces, contract	27	9,996	466
`Abstract_Interpreter`	domain interfaces, D/G framework, routing, solver bridge	67	21,228	1,033
`Analyses`	Sign, Interval, Parity, Congruence, Int, relational witness	70	15,800	619
`Executable_Surface`	analyzer function, report, end-to-end theorems, export	15	5,222	180
`Examples`	non-vacuity, precision witnesses, regressions	34	7,231	397
Voblint total		213	59,477	2,695
Solver (imported)	Tilscher et al. [35]	7	9,533	168

Table 2: Size of the development by layer: lines of Isabelle/HOL including comments, and proved statements, which are `lemma`, `theorem` and `corollary` commands. The counts exclude one theory generated from the grammar manifest (`VIMP_Grammar_Generated`).

With activation traces, the invariant concerns the caller chain recovered by *caller_of*; contexts remain a separate policy parameter.

- *Two-default executable states.* Soundness is stated over total functions on variable names; the solver runs on a quotient type [13] of a finite override list with separate defaults for locals and zero-initialized globals. The list stores exceptions to the defaults; the quotient identifies representations of the same function (`exec_local_spec_sound`). Separate defaults refine the single default of Concrete Semantics [25, §13.6].

Feedback on the proof assistant. Connecting the executable solver to its logical definition required a separate proof. Functions in Isabelle/HOL are total, so the potentially diverging solver has a domain predicate. For a terminating execution, we must show that the logical solver is defined. The added lemma `solve_dom_of_solve_c` establishes this implication. Code generation also turns finite sets and maps into lists that every lookup scans, and CPU profiles attribute most of the solving time to these scans. Replacing these list representations through the Containers library is a possible optimization that we have not evaluated.

7 Evaluation

A soundness theorem may have unsatisfiable premises or allow an analyzer that answers UNKNOWN everywhere (`unknown_everywhere_sound`). We therefore exhibit programs for which the theorem applies and the analyzer proves checks, and show that a domain obligation rejects a known defect. We then compare the precision of context policies on a recursive program. Facts about one fixed program are proved by `eval`, which runs generated code and therefore also trusts the code generator. Recorded analyzer outputs are checked against fresh runs by `pixi run thesis-claims`; table 3 is generated from these claims.

Non-vacuity (machine-checked, one program each). For the program of fig. 1, analyzed with Interval, entry-state contexts, and the warrowing update rule, we construct a source execution in Isabelle that returns from both calls. Evaluation shows that the analyzer answers with both checks PROVED, and `nv_source_certified` instantiates theorem 3 with every premise discharged. Companion instances do the same for a single check (`nv_check_proved_sound`) and for a DEAD check (`nv_dead_unreached`).

Necessity of the soundness obligations (machine-checked). The theorem `total_dropped_unsound` exhibits a cover that meets the other four coverage obligations but omits a reachable store, showing why TOTAL is required.

Context policy	Contexts	`res1 == 2`		`res2 == 1`	
one context per procedure	2	UNKNOWN	$[1, 12]$	UNKNOWN	$[1, 12]$
call strings, length 1	4	UNKNOWN	$[1, 12]$	UNKNOWN	$[1, 12]$
call strings, length 2	6	PROVED	$[2, 2]$	PROVED	$[1, 1]$
entry states	6	PROVED	$[2, 2]$	PROVED	$[1, 1]$

Table 3: Callee contexts of `f` and `g`, verdicts and computed values for the two checks of the regression fixture `09-depth2_mutual_recursion_in_loop` (group `17-call-string`) under Interval, per context policy. Concretely, `res1` is always 2 and `res2` always 1. Generated from reproducible analyzer runs.

A separate example tests a domain obligation against a real Goblint defect. Goblint once claimed `c % 2 == 1` for $c \in \{-5, -7\}$, although C's truncating remainder gives -1 (issue 1156, fixed in [28]). The defect concerns the domain's arithmetic operation. Each domain's abstract remainder operation must satisfy `mod_sound` in the locale `sound_arith_ops`: the concretization of its result must contain the truncating remainders of all concrete operands represented by its arguments. The theorem `congruence_mod_sound` establishes this property for the congruence domain. `prefix_congruence_mod_unsound` proves that every abstract remainder returning the constant 1 for $(1 + 2\mathbb{Z}) \bmod 2$ violates the obligation at -5, so the defective operator could not be registered.

Calling contexts on recursive programs (executable). Table 3 reports one regression program's results under each context policy. Mutually recursive procedures `f` and `g` return 2 and 1 at the bottom of the recursion. `main` calls `f(2)` and `g(2)` inside a loop of 200 iterations and checks both results. Each call chain is three call sites deep. The table also counts the copies of `f` and `g` the analysis creates. With one context per procedure, or with call strings of length 1, the activations of different depths share a state and both results widen to $[1, 12]$. With call strings of length 2 or with entry-state contexts, activations at different depths have separate abstract states, and both checks are proved. The loop adds no contexts, because the callee contexts do not depend on the loop counter. A related program has a machine-checked precision comparison (`sign_k2_strictly_more_precise_than_k1_at_g`). At one callee entry, the routed solution for call-string length 2 is strictly more precise than for length 1. That theorem compares two equation-system solutions computed by `eval` for this program. It establishes neither a corresponding result about `run_voblint` nor a general precision ordering between call-string lengths.

The thesis [19] also reports Voblint's results on the programs of Monniaux's tutorial on the flavors of abstract interpretation [23] and compares the verdicts with Goblint's.

Running time (empirical). We use the OCaml code generated by Isabelle without hand-written optimization. A proved code equation, `init_publications_code`, avoids recompiling the program during equation evaluation. The profiling run at repository revision `17fcb363d` analyzed the 380 regression fixtures that set no timeout of their own. The 15 configurations combine five domains with three context policies (none, call strings of length 1, entry states). Excluding the eleven ill-formed programs leaves 369 fixtures and 5,535 cases. A case's time is the median of three repetitions on an Apple M4 Pro; a case that times out (30 s) or fails in any repetition counts as failed. Most fixtures are small regression programs, so low solve times on this corpus do not establish performance on larger programs. All 21 timeouts are entry-state cases with Interval, Int or Congruence on seven recursive fixtures, where each recursion level creates a new context. One is `24-site-figures/precision/13-recursion_grows_warrow.vimp`: `f(x)` checks `x >= 0` and calls `f(x + 1)`. With Interval and one context per procedure, the analyzer proves the check; with entry-state contexts, the analysis exceeds the 30 s timeout. Call strings of length 2 were not profiled.

A separate scaling sweep uses Interval on eight generated program families. For each family, we fit solve time t against its size parameter n as $t = an^b$ by least squares on logarithmic axes. We use successful timings from $n = 100$ to the largest completed size (500–2000, depending on the family). The fitted exponents range from 2.01 to 2.86. The profiling snapshot records the measurements. The profiling script defines the families. The profiling page shows the runtime distributions and scaling curves. The scaling sweep exposes costs hidden by the small regression fixtures; it establishes neither an asymptotic bound nor performance on realistic large programs.

8 Related Work

Concrete semantics of calling contexts. Sharir and Pnueli distinguish two approaches [33]. The functional approach computes input–output relations of procedures; the call-string approach tags information with the history of calls. Voblint's call-string policy follows the second, and its entry-state policy, which indexes a callee's exit states by its entry state, resembles a functional approach tabulated per input. Knoop and Steffen handle the locals of recursive procedures with stacks of lattice elements; their maximal-fixpoint strategy uses stacks of length at most two [16]; Voblint's return has the shape of their return function [16, §4.1]. Reps et al. compute precise solutions when the set of dataflow facts is finite and the transfer functions are distributive [26]. Voblint requires neither a finite domain nor distributive transfer functions, but establishes no corresponding general precision theorem. Jeannet and Serwe derive interprocedural analyses by abstract interpretation of a standard stack-based operational semantics and combine the call-string and functional approaches [14]. Van Horn and Might derive sound analyses from abstract machines by bounding their stores [12]. These approaches relate analysis contexts to concrete executions. In the a posteriori soundness framework of Might and Manolios [22], a map from concrete to abstract resources parameterizes the state abstraction. They construct a suitable map after the analysis. Voblint instead derives its entry-state context relation from the solution; Total ensures that every call from a covered store admits a callee context (section 4).

Verified analyzers. Verasco is a Coq-verified analyzer for most of C99, excluding recursion and dynamic allocation [15]. Its interpreter iterates over the structure of C#minor, reanalyzes a function at every call site, and raises an alarm at a possible recursive call. Blazy et al. use a verified checker to validate the candidate fixpoint produced by an untrusted iterator [3]. Nipkow and Klein develop abstract interpretation for IMP in Isabelle/HOL [24, 25], and Voblint's non-relational domains follow their generic interpreter. Like the verified analyzers of Franceschino et al. [9] and Cachera and Pichardie [6], Nipkow and Klein's IMP interpreter targets a language without procedures. Michelland et al. derive concrete and abstract interpreters from a shared monadic denotation and prove the abstract interpreter sound [21]. Cachera et al. compute a context-insensitive dataflow analysis with a verified solver for ordinary inequations [5].

Contexts in mechanized semantics. Dabrowski and Pichardie prove a context-sensitive points-to analysis sound in Coq [7]. Their concrete semantics carries contexts through a module type whose call-context function computes a callee's context. This is the closest precedent for Voblint's context-indexed semantics. Their development specifies the analyses as constraints but does not provide an executable implementation [7, §7]. Voblint's contexts are admitted by a relation, the totality condition makes the indexed collection exhaustive, and the indexing is connected to an executable verified solver. Trace partitioning indexes sets of traces by control history [27]; Rival and Mauborgne allow coverings in which one trace has several indices and present calling contexts as one such indexing. The activation collecting semantics is a covering of this relational kind. Among Isabelle developments, Lammich and Müller-Olm verify the soundness and precision of a conflict analysis for programs with procedures and threads [17], and Breitner formalizes Shivers' control-flow analysis for functional programs [4]. Wasserrab

proves correctness of interprocedural slicing, which removes statements irrelevant to selected observations, by weak simulation [39]. This preserves slicing observations; Voblint instead bounds reachable stores.

Verified solvers. Hofmann et al. verify a local generic solver in Coq [11], and de Vilhena et al. verify one that discovers dependencies dynamically in Iris [8]. Stade et al. prove the top-down solver partially correct in Isabelle/HOL [34], and Tilscher et al. extend it to side effects with a generic interface for update rules [35]. Voblint reuses this solver formalization. Their termination theorem for the solver with widening and narrowing in phases assumes finitely many unknowns [37]. Voblint does not establish this condition for entry-state contexts.

Goblint and local traces. Side-effecting constraint systems [31, 1] and the split between analyses and solvers are Goblint's architecture [38, 32], and Voblint's `dg_spec` follows its `Spec` interface. Voblint's theorems do not establish soundness of Goblint's OCaml implementation. Schwarz et al. prove Goblint's thread-modular analyses sound against local traces on paper [29, 30]. Activation traces adapt this construction to sequential procedure activations; the adaptation establishes no concurrency result.

9 Conclusion

We have connected the source semantics of VIMP to a Goblint-style analyzer based on context-sensitive, side-effecting equations and a verified solver. A returned report for an accepted configuration and a scalar VIMP program covers reached stores at every program point. The report's definite verdicts hold for those stores. Precision improvements have been shown for individual programs, but no general precision guarantee is proved.

The report also preserves coverage within each context (section 5).

Language extensions. The coverage contract abstracts over the language's step functions. Extending VIMP with machine integers would require new concrete operations, abstract transfers, and soundness proofs. A heap without pointers into caller frames could be modeled by extending the store, again with new transfers and proofs. Pointers into caller frames and interference from other threads would also require revisiting the trace semantics, which assumes that a callee cannot modify its caller's locals. These extensions remain future work.

Termination. Known termination results for top-down solvers assume finitely many unknowns [37]. Entry-state context sets can be infinite, and all timeouts in section 7 occur under this policy. Goblint controls context creation through optional widening on contexts (`ana.context.widen`) or *context gas*, a budget for context-sensitive calls along a call stack. Establishing termination for Voblint would require controlling context creation and satisfying the remaining solver termination conditions.

Data availability. The Isabelle/HOL development and regression corpus are public on GitHub. The earlier Zenodo archive [18] lacks the vendored solver, the claims behind table 3, the profiling snapshot, and the table scripts. A reproducible release must archive these dependencies together. The upstream solver artifact is public [36], but Voblint depends on a modified fork that is not publicly accessible. Reproducing this development requires that exact dependency.

Declaration about GenAI/LLM use. The author extensively used Anthropic's Claude, OpenAI's ChatGPT and Codex, and Cursor. Under the author's supervision, these tools produced a large share of the implementation. This included Isabelle proofs after definitions and interfaces stabilized, cross-theory refactorings, and conversions of tactic-style proofs to

structured Isar. They helped adapt the local traces of Schwarz et al. to procedure activations and refine the trace requirements. Agents wrote much of the command-line interface, regression corpus, and consistency tooling. The author chose the research direction, selected and revised designs, and approved definitions and theorem statements; Isabelle checked every proof. The tools also assisted with literature search and drafting and revising the text and figures, which the author reviewed. Grammarly checked spelling and grammar.

References

[1] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. Side-effecting constraint systems: A swiss army knife for program analysis. In *Programming Languages and Systems (APLAS 2012)*, volume 7705 of *LNCS*, pages 157–172. Springer, 2012. doi:10.1007/978-3-642-35182-2_12.

[2] Clemens Ballarin. Locales: A module system for mathematical theories. *Journal of Automated Reasoning*, 52(2):123–153, 2014. doi:10.1007/s10817-013-9284-7.

[3] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Static Analysis*, pages 324–344, 2013. doi:10.1007/978-3-642-38856-9_18.

[4] Joachim Breitner. Shivers' control flow analysis. *Archive of Formal Proofs*, November 2010. URL: https://isa-afp.org/entries/Shivers-CFA.html.

[5] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005. doi:10.1016/j.tcs.2005.06.004.

[6] David Cachera and David Pichardie. A certified denotational abstract interpreter. In *Interactive Theorem Proving (ITP 2010)*, LNCS, pages 9–24. Springer, 2010. doi:10.1007/978-3-642-14052-5_3.

[7] Frédéric Dabrowski and David Pichardie. A certified data race analysis for a Java-like language. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, LNCS, pages 212–227, 2009. doi:10.1007/978-3-642-03359-9_16.

[8] Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. Spy game: Verifying a local generic solver in Iris. *Proceedings of the ACM on Programming Languages*, 4(POPL):33:1–33:28, 2020. doi:10.1145/3371101.

[9] Lucas Franceschino, David Pichardie, and Jean-Pierre Talpin. Verified functional programming of an abstract interpreter. In *Static Analysis (SAS 2021)*, LNCS, pages 124–143. Springer, 2021. Hosted browser version: https://w95psp.github.io/verified-abstract-interpreter. doi:10.1007/978-3-030-88806-0_6.

[10] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming (FLOPS 2010)*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010. doi:10.1007/978-3-642-12251-4_9.

[11] Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. Verifying a local generic solver in Coq. In *Static Analysis (SAS 2010)*, LNCS, pages 340–355. Springer, 2010. doi:10.1007/978-3-642-15769-1_21.

[12] David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 51–62. ACM, 2010. doi:10.1145/1863543.1863553.

[13] Brian Huffman and Ondřej Kunčar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs (CPP 2013)*, LNCS, pages 131–146. Springer, 2013. doi:10.1007/978-3-319-03545-1_9.

[14] Bertrand Jeannet and Wendelin Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *Algebraic Methodology and Software Technology (AMAST 2004)*, volume 3116 of *Lecture Notes in Computer Science*, pages 258–273. Springer, 2004. doi:10.1007/978-3-540-27815-3_22.

[15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 247–259. ACM, 2015. doi:10.1145/2676726.2676966.

[16] Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. In *Compiler Construction (CC '92)*, volume 641 of *LNCS*, pages 125–140. Springer, 1992. doi:10.1007/3-540-55984-1_13.

[17] Peter Lammich and Markus Müller-Olm. Formalization of conflict analysis of programs with procedures, thread creation, and monitors. *Archive of Formal Proofs*, December 2007. URL: https://isa-afp.org/entries/Program-Conflict-Analysis.html.

[18] Manuel Lerchner. Voblint: A verified Goblint-style static analysis pipeline in Isabelle/HOL, 2026. doi:10.5281/zenodo.23260696.

[19] Manuel Lerchner. Voblint: Towards a verified Goblint-style analysis pipeline in Isabelle/HOL. Master's thesis, Technical University of Munich, 2026. Draft. URL: https://manuellerchner.github.io/Voblint-Verification-Master-Thesis/thesis.pdf.

[20] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

[21] Sébastien Michelland, Yannick Zakowski, and Laure Gonnord. Abstract interpreters: A monadic approach to modular verification. *Proceedings of the ACM on Programming Languages*, 8(ICFP), 2024. doi:10.1145/3674646.

[22] Matthew Might and Panagiotis Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, volume 5403 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2009. doi:10.1007/978-3-540-93900-9_22.

[23] David Monniaux. Flavors of abstract interpretation. Invited tutorial, European Joint Conferences on Theory and Practice of Software (ETAPS 2024), Luxembourg, April 2024. Lecture slides, HAL hal-04750222. URL: https://hal.science/hal-04750222.

[24] Tobias Nipkow. Abstract interpretation of annotated commands. In *Interactive Theorem Proving (ITP 2012)*, LNCS, pages 116–132. Springer, 2012. doi:10.1007/978-3-642-32347-8_9.

[25] Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. doi:10.1007/978-3-319-10542-0.

[26] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 49–61. ACM, 1995. doi:10.1145/199448.199462.

[27] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems*, 29(5), 2007. doi:10.1145/1275497.1275501.

[28] Michael Schwarz. Congruences: Make % sound by restricting cases where we return a constant, 2023. Pull request 1161. URL: https://github.com/goblint/analyzer/pull/1161.

[29] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. Improving thread-modular abstract interpretation. In *Static Analysis*, volume 12913 of *LNCS*, pages 359–383, 2021. doi:10.1007/978-3-030-88806-0_18.

[30] Michael Schwarz, Simmo Saan, Helmut Seidl, Julian Erhard, and Vesal Vojdani. Clustered relational thread-modular abstract interpretation with local traces. In *Programming Languages and Systems (ESOP 2023)*, volume 13990 of *LNCS*, pages 28–58, 2023. doi:10.1007/978-3-031-30044-8_2.

[31] Helmut Seidl, Varmo Vene, and Markus Müller-Olm. Global invariants for analysing multi-threaded applications. *Proceedings of the Estonian Academy of Sciences. Physics. Mathematics*, 52(4):413–436, 2003. doi:10.3176/phys.math.2003.4.05.

[32] Helmut Seidl, Vesal Vojdani, Julian Erhard, and Michael Schwarz. Mixed flow-sensitive static analysis: Engineering modularity. In *FM 2026: Formal Methods*, pages 446–470, 2026. doi:10.1007/978-3-032-26220-2_22.

[33] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice-Hall, 1981.

[34] Yannick Stade, Sarah Tilscher, and Helmut Seidl. The top-down solver verified: Building confidence in static analyzers. In *Computer Aided Verification*, pages 303–324, 2024. Formalization: Archive of Formal Proofs entry Top_Down_Solver, https://isa-afp.org/entries/Top_Down_Solver.html. doi:10.1007/978-3-031-65627-9_15.

[35] Sarah Tilscher, Alexandra Graß, and Helmut Seidl. Verifying a solver for mixed flow-sensitive analyses. In *NASA Formal Methods (NFM 2026)*, LNCS, pages 48–69. Springer, 2026. doi:10.1007/978-3-032-28079-4_3.

[36] Sarah Tilscher, Alexandra Graß, and Helmut Seidl. Verifying a solver for mixed flow-sensitive analyses (artifact), 2026. doi:10.5281/zenodo.19259984.

[37] Sarah Tilscher, Alexandra Graß, Helmut Seidl, and Yannick Stade. Proving total correctness of top-down solvers with widening and narrowing. *Journal of Automated Reasoning*, 70(2), 2026. Article 15. doi:10.1007/s10817-026-09756-x.

[38] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: The Goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 391–402. ACM, 2016. doi:10.1145/2970276.2970337.

[39] Daniel Wasserrab. Backing up slicing: Verifying the interprocedural two-phase Horwitz-Reps-Binkley slicer. *Archive of Formal Proofs*, November 2009. URL: https://isa-afp.org/entries/HRB-Slicing.html.