A Formally-Verified C Static Analyzer

Jacques-Henri Jourdan
Inria Paris-Rocquencourt
jacques-henri.jourdan@inria.fr

Vincent Laporte
IRISA and U. Rennes 1
vincent.laporte@irisa.fr

Sandrine Blazy
IRISA and U. Rennes 1
sandrine.blazy@irisa.fr

Xavier Leroy
Inria Paris-Rocquencourt
xavier.leroy@inria.fr

David Pichardie
IRISA and ENS Rennes
david.pichardie@irisa.fr

Abstract

This paper reports on the design and soundness proof, using the Coq proof assistant, of Verasco, a static analyzer based on abstract interpretation for most of the ISO C 1999 language (excluding recursion and dynamic allocation). Verasco establishes the absence of run-time errors in the analyzed programs. It enjoys a modular architecture that supports the extensible combination of multiple abstract domains, both relational and non-relational. Verasco integrates with the CompCert formally-verified C compiler so that not only the soundness of the analysis results is guaranteed with mathematical certitude, but also the fact that these guarantees carry over to the compiled code.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—*Assertion checkers*, *Correctness proofs*; F.3.1 [*Logics and meanings of programs*]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

Keywords static analysis; abstract interpretation; soundness proofs; proof assistants

1. Introduction

Verification tools are increasingly used during the development and validation of critical software. These tools provide guarantees that are always independent from those obtained by more conventional means such as testing and code review; often stronger; and sometimes cheaper to obtain (rigorous testing can be very expensive). Verification tools are based on a variety of techniques such as static analysis, model checking, deductive program proof, and combinations thereof. The guarantees they provide range from basic memory safety to full functional correctness. In this paper, we focus on static analyzers for low-level, C-like languages that establish the absence of run-time errors such as out-of-bound array accesses, null pointer dereference, and arithmetic exceptions. These basic properties are essential both for safety and security. Among the various verification techniques, static analysis is perhaps the one that scales best to large existing code bases, with minimal intervention from the programmer.

Static analyzers can be used in two different ways: as sophisticated bug finders, discovering potential programming errors that are hard to find by testing; or as specialized program verifiers, establishing that a given safety or security property holds with high confidence. For bug-finding, the analysis must be precise (too many false alarms render the tool unusable for this purpose), but no guarantee is offered nor expected that all bugs of a certain class will be found. For program verification, in contrast, *soundness* of the analysis is paramount: if the analyzer reports no alarms, it must be the case that the program is free of the class of run-time errors tracked by the analyzer; in particular, all possible execution paths through the program must be accounted for.

To use a static analyzer as a verification tool, and obtain certification credit in regulations such as DO-178C (avionics) or Common Criteria (security), evidence of soundness of the analyzer must therefore be provided. Owing to the complexity of static analyzers and of their input data (programs written in "big" programming languages), rigorous testing of a static analyzer is very difficult. Even if the analyzer is built on mathematically-rigorous grounds such as abstract interpretation [14], the possibility of an implementation bug remains. The alternative we investigate in this paper is *deductive formal verification of a static analyzer*: we apply program proof, mechanized with the Coq proof assistant, to the implementation of a static analyzer in order to prove its soundness with respect to the dynamic semantics of the analyzed language.

Our analyzer, called Verasco, is based on abstract interpretation; handles most of the ISO C 1999 language, with the exception of recursion and dynamic memory allocation; combines several abstract domains, both non-relational (integer intervals and congruences, floating-point intervals, points-to sets) and relational (convex polyhedra, symbolic equalities); and is entirely proved to be sound using the Coq proof assistant. Moreover, Verasco is connected to the CompCert C formally-verified compiler [26], ensuring that the safety guarantees established by Verasco carry over to the compiled code.

Mechanizing soundness proofs of verification tools is not a new idea. It has been applied at large scale to Java type-checking and bytecode verification [25], proof-carrying code infrastructures [1, 12], and verification condition generators for C-like languages [20, 23], among other projects. The formal verification of static analyzers based on dataflow analysis or abstract interpretation is less developed. As detailed in section 10, earlier work in this area either

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676966

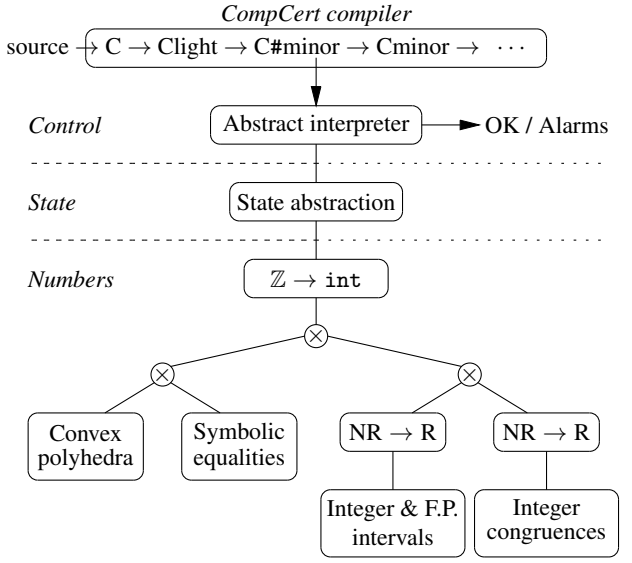

Figure 1. Modular architecture of the Verasco static analyzer

focuses on simple static analyses (dataflow analyses, no widening, non-relational domains only) or on mini-languages such as IMP. Compared with this earlier work on verified static analyzers, Verasco is a quantitative jump: the source language analyzed (most of C) is much more complex, and the static analysis technique used (combination of several abstract domains, including relational domains) is much more sophisticated.

This paper reports on the design and Coq verification of the Verasco static analyzer. In addition to the the quantitative jump mentioned above, we emphasize as a contribution the modular architecture of the analyzer and its verification, including fully-specified interfaces for the various components that makes it easy to connect new abstract domains to Verasco, as well as to reuse Verasco's domains in other projects. The full Coq development is available at `http://compcert.inria.fr/verasco/`.

The paper is organized as follows. Section 2 presents the general architecture of the analyzer. The next five sections give more details on the source language (§3), the abstract interpreter (§4), the state and memory abstraction (§5), the numerical abstract domains (§6) and how multiple domains communicate (§7). We finish by some notes on the Coq development (§8), preliminary experimental results (§9), discussion of related work (§10), and conclusions and perspectives (§11).

2. Architecture of the analyzer

The general architecture of the Verasco analyzer is depicted in figure 1. It is inspired by that of ASTRÉE [6] and is structured in three layers. At the top sits the abstract interpreter that infers abstract states at every program point and checks for potential run-time errors, raising alarms along the way. The abstract interpreter operates over the C#minor intermediate language described in section 3. This language is the second intermediate language in the CompCert compilation pipeline. The Verasco analyzer reuses the CompCert front-end to produce C#minor from the source C code. The semantics preservation theorem of CompCert guarantees that any safety property established on the C#minor intermediate language carries over to the assembly code generated by CompCert. Combining this theorem with the soundness theorem for Verasco, we obtain that any C#minor program that passes analysis without raising an alarm compiles to assembly code that is free of run-time errors. The Verasco abstract interpreter proceeds by fixpoint iteration that follows the structure of the C#minor program. Section 4 gives more details on this abstract interpreter and its soundness proof.

The middle layer of Verasco is an abstract domain for execution states, tracking the values of program variables, the contents of memory locations, and the chain of function calls. This state abstract domain is described in section 5. It is a parameter of the abstract interpreter, and has a well-defined interface (in terms of abstract operations provided and their specifications) outlined below. This parameterization makes it possible to experiment with several state domains of various precision, even though we currently have only one implementation of the state domain. Concerning values that arise during program execution, the domain tracks pointer values itself via points-to analysis, but delegates the tracking of numerical values to a numerical domain (bottom layer).

At the bottom layer of Verasco, the numerical abstract domain is itself an extensible combination of several domains. Some are non-relational, such as intervals and congruences, and track properties of the (integer or floating-point) value of a single program variable or memory cell. Others are relational, such as convex polyhedra and symbolic equalities, and track relations between the values of several variables or cells. Two domain transformers perform adaptation over domains: the "NR $\rightarrow$ R" transformer gives a relational interface to a non-relational domain, and the "$\mathbb{Z} \rightarrow$ `int`" transformer handles the overflow and wrap-around behaviors that occur when mathematical integers (type $\mathbb{Z}$) and their arithmetic operations are replaced by machine integers (n-bit vectors) and their modulo-2^n arithmetic. Section 6 describes these abstract domains and their verification; section 7 explains how they are combined and how they can exchange information during analysis.

Supporting such a modular composition of formally-verified abstract domains requires that they adhere to well-defined *interfaces*. Figure 2 shows one of the three major interfaces used in Verasco (slightly simplified), the one for "machine" relational domains that acts as gateway between the numerical domains and the state domain. All Verasco interfaces are presented as Coq's type classes. A machine relational domain consists of a type `t` equipped with a semi-lattice structure: a decidable ordering `leb`, a `top` element, a `join` operation that returns an upper bound of its arguments (but not necessarily the least upper bound), and a `widen` operation used to accelerate the convergence of fixpoint iteration with widening. There is no bottom element in Verasco's domains: instead, when we need to represent unreachability, we use the type `t+`$\bot$ that adds a generic `Bot` element to the domain `t`.

The three most important operations are `forget`, `assign` and `assume`. The `forget` x A operation removes all information associated with the variable x in state A, simulating a nondeterministic assignment to x. The type `var` of variables is another parameter of the class: it can be instantiated by program variables or, as the state abstract domain does, by abstract memory cells.

The `assign` x e A operation updates A to reflect the assignment of expression e to variable x. Numerical expressions e are built upon variables and constants using the arithmetic, logical and comparison operators of C#minor. (They are similar to C#minor expressions except that they do not feature memory loads and that intervals can occur instead of numerical constants, capturing some amount of nondeterminism.) When analyzing source-level assignments, it is crucial that the numerical domains receive a numerical expression as close as possible to the right-hand side of the source-level assignment, typically the same expression modulo the replacement of memory loads by variables representing the memory cells accessed; then, each domain can treat it to the best of its abilities. For example, on treating `x = y + z`, an interval domain will simply set `x` to the sum of the intervals associated with `y`

```
Class ab_machine_env (t var: Type): Type :=
  { leb: t → t → bool
  ; top: t
  ; join: t → t → t
  ; widen: t → t → t
  ; forget: var → t → t+⊥
  ; assign: var → nexpr var → t → t+⊥
  ; assume: nexpr var → bool → t → t+⊥
  ; nonblock: nexpr var → t → bool
  ; concretize_int: nexpr var → t → int_set+⊤+⊥

  ; γ : t → ℘ (var→num_val)
  ; gamma_monotone: forall x y,
    leb x y = true → γ x ⊆ γ y;
  ; gamma_top: forall x, x ∈ γ top;
  ; join_sound: forall x y,
    γ x ∪ γ y ⊆ γ (join x y)
  ; forget_correct: forall x ρ n ab,
    ρ ∈ γ ab →
    (upd ρ x n) ∈ γ (forget x ab)
  ; assign_correct: forall x e ρ n ab,
    ρ ∈ γ ab →
    n ∈ eval_nexpr ρ e →
    (upd ρ x n) ∈ γ (assign x e ab)
  ; assume_correct: forall e ρ ab b,
    ρ ∈ γ ab →
    of_bool b ∈ eval_nexpr ρ e →
    ρ ∈ γ (assume e b ab)
  ; nonblock_correct: forall e ρ ab,
    ρ ∈ γ ab →
    nonblock e ab = true →
    block_nexpr ρ e →
    False
  ; concretize_int_correct: ∀ e ρ ab i,
    ρ ∈ γ ab →
    NVint i ∈ eval_nexpr ρ e →
    i ∈ γ (concretize_int e ab)
}.
```

Figure 2. The interface for machine relational domains (slightly simplified).

and z, while a polyhedral domain will record the two inequalities $\mathtt{x} \leq \mathtt{y} + \mathtt{z} \wedge \mathtt{x} \geq \mathtt{y} + \mathtt{z}$.

Finally, `assume` e b A refines the abstract state A to reflect the fact that expression e evaluates to the truth value b (either `true` or `false`). It is used when analyzing conditional statements (`if`, `switch`) to keep track of the value of the discriminating expression.

Two query operations are provided to help the abstract interpreter detect potential run-time errors and raise alarms appropriately: `nonblock` e A returns `true` if e is guaranteed to evaluate safely and `false` if it can cause a run-time error (such as a division by zero) when evaluated in a concrete state matching A; and `concretize_int` e A returns the set of possible integer values for the expression e that can be deduced from the information in abstract state A.

The interface in figure 2 also specifies the soundness conditions for the abstract operations above. The specification uses a *concretization function* γ that maps abstract states A to sets of concrete environments `var` → `num_val` mapping variables to values. Here, values are the tagged union of 32-bit integers, 64-bit integers, and double-precision floating-point numbers (IEEE754 `binary64` format), that is, the numerical types offered by C#minor. As customary in Coq, the type $\wp(\mathtt{t})$ of sets of `t`'s is encoded as `t` → `Prop`, so that γ is really a two-place predicate relating abstract states and concrete environments. The specification does not use a full Galois connection $A \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \wp(C)$ because the abstraction function α is problematic in a constructive logic such as Coq's: first, it is not a computable function as soon as the type C of concrete "things" is infinite; second, for some domains, α is not well defined.[1]

The lack of an abstraction function α changes the style of specification of abstract operators, focusing the specification on soundness conditions and freeing us from the obligation to prove relative optimality. For example, the specification of `forget` in Galois-connection style would be

$$\mathtt{forget}\ x\ A = \alpha\{\rho[x \leftarrow v] \mid \rho \in \gamma(A), v \in \mathit{Values}\}$$

Instead, in γ-only style we state soundness as

$$\forall \rho, v,\ \rho \in \gamma(A) \Rightarrow \rho[x \leftarrow v] \in \gamma(\mathtt{forget}\ x\ A)$$

(condition `forget_correct` in figure 2). We could also state relative optimality using γ only:

$$\forall A',\ (\forall \rho, v,\ \rho \in \gamma(A) \Rightarrow \rho[x \leftarrow v] \in \gamma(A')) \Rightarrow \mathtt{forget}\ x\ A \sqsubseteq A'$$

However, we elected to mechanize proofs of soundness only, leaving relative optimality optional.

The other two major interfaces of Verasco are similar in spirit to what is described above, with the following differences. The interface for "ideal" numerical relational domains concretizes not to machine numbers, but to the sum of mathematical integers (type $\mathbb{Z}$) and FP numbers. It also supports communication channels between domains, as described in section 7. The interface for abstract execution states that mediates between the abstract interpreter and the state domain uses full C#minor expressions, including memory loads; adds abstract operations to handle memory stores and to push and pop function calls in a call context; and concretizes to C#minor memory states and local variable environments (section 5).

3. The C#minor language

C#minor is the second intermediate language in the CompCert compilation pipeline, immediately preceding Cminor, the entry language of the CompCert back-end described in [27]. Classically, C#minor is structured in functions, statements, and expressions.

Expressions:

$e ::= t$	reading a temporary variable
$\mid \mathtt{\&}x$	address of a variable
$\mid cst$	constants
$\mid op_1(e) \mid op_2(e, e')$	arithmetic operations
$\mid \mathtt{load}(\tau, e)$	memory load with size τ

Statements:

$s ::= \mathtt{skip}$	
$\mid t := e$	assignment
$\mid \mathtt{store}(\tau, e_1, e_2)$	memory store with size τ
$\mid t := e(e_1, \ldots, e_n)$	function call
$\mid (s_1; s_2)$	sequence
$\mid \mathtt{if}\ e\ s_1\ \mathtt{else}\ s_2$	conditional
$\mid \mathtt{loop}\ s$	infinite loop
$\mid \mathtt{block}\ s$	
$\mid \mathtt{exit}\ n$	terminate $n+1$ enclosing `blocks`
$\mid L : s$	define label L
$\mid \mathtt{goto}\ L$	jump to label L
$\mid \mathtt{return} \mid \mathtt{return}\ e$	function return

[1] For example, in the domain of linear rational inequalities, the set of pairs $\{(x, y) \mid x^2 + y^2 \leq 1\}$ has no best approximation as a polyhedron [28].

Functions:

f ::= *name* $(\ldots p_i \ldots)$ {
  `vars` $\ldots x_i[size_i] \ldots$ local variables
  `temps` $\ldots t_i \ldots$ temporary variables
  s function body
}

A program is composed of function definitions and global variable declarations. Variables are of two kinds: addressable (their address can be taken with the `&` operator) and temporary (not resident in memory). Expressions have no side effects: assignments, memory stores and function calls are statements. The arithmetic, logical, comparison, and conversion operators are roughly those of C, but without overloading: for example, distinct operators are provided for integer multiplication and FP division. Likewise, there are no implicit casts: all conversions between numerical types are explicit.

Statements offer both structured control and `goto` with labels. C loops as well as `break` and `continue` statements are encoded as infinite loops with a multi-level `exit` n that jumps to the end of the $(n+1)$-th enclosing `block`.

The first passes of CompCert perform the following transformations to produce C#minor from C sources. First, side effects are pulled outside of expressions, and temporaries are introduced to hold their values. For example, `z = f(x) + 2 * g(y)` becomes

```
t1 = f(x);  t2 = g(y);  z = t1 + 2 * t2;
```

This transformation effectively picks one evaluation order among the several orders allowed in C. Second, local variables of scalar types whose addresses are never taken are "pulled out of memory" and turned into temporaries. Third, all type-dependent behaviors are made explicit: operator overloading is resolved, implicit conversions are materialized, and array and `struct` accesses become `load` and `store` operations with explicit address computations. Fourth and last, C loops are encoded using `block` and `exit`, as outlined in [7].

The dynamic semantics of C#minor is given in small-step style as a transition relation $C \xrightarrow{\nu} C'$ between configurations C. The optional ν label is an observable event possibly produced by the transition, such as accessing a `volatile` variable. A typical configuration C comprises a statement s under consideration, a continuation k that describes what to do when s terminates (e.g., "move to the right part of a sequence", "iterate a loop once more", or "return from current function"), and a dynamic state ρ mapping temporaries to their values, local and global variables to their addresses, and memory cells to their contents. The continuation k encodes both a context (where does s occur in the currently-executing function) and a call stack (the chain of pending function calls). Some transition rules actually perform computations (e.g., an assignment); others are refocusing rules that change s and k to focus on the next computation.

The transition rules for C#minor are omitted from this paper, but resemble those for Cminor given in [27, section 4]. The semitone difference between Cminor and C#minor is that every Cminor function has exactly one addressable variable called the stack data block, while a C#minor function has zero, one or several variables bound to logically separated memory blocks.

4. The abstract interpreter

Exploring all execution paths of a program during static analysis can be achieved in two ways: the control flow graph (CFG) approach and the structural approach. In the CFG approach, a control flow graph is built with program points as nodes, and edges carrying elementary commands (assignments, tests, ...). The transfer function T for the analysis is defined for elementary commands. The analyzer, then, sets up a system of inequations

$$A_{p'} \sqsupseteq T\ c\ A_p \quad \text{where } p \xrightarrow{c} p' \text{ is a CFG edge}$$

with unknowns A_p, the abstract states associated to every program point p. This system is then solved by global fixpoint iteration over the whole CFG.

In contrast, the structural approach applies to languages with structured control and compound statements such as `if` and loops. There, the transfer function T is defined over basic as well as compound statements: given the abstract state A "before" the execution of statement s, it returns $T\ s\ A$, the abstract state "after" the execution of s. For sequences, we have $T\ (s_1; s_2)\ A = T\ s_2\ (T\ s_1\ A)$. For loops, T takes a local fixpoint of the transfer function for the loop body.

Since C#minor is a mostly structured language (only `goto` statements are unstructured), the abstract interpreter for C#minor follows the structural approach. This obviates the need to define program points for C#minor (a nontrivial task). Moreover, structural abstract interpreters use less memory than CFG-based ones, maintaining only a few different abstract states at any time, instead of one per program point. However, the transfer function for our abstract interpreter is more involved than usual, because control can enter and leave a C#minor statement in several ways. The statement s can be entered normally at the beginning, or via a `goto` that branches to one of the labels defined in s. Likewise, s can terminate either normally by running to the end, or prematurely by executing a `return`, `exit`, or `goto` statement. Consequently, the transfer function is of the form

$$T\ s\ (A_i, A_l) = (A_o, A_r, A_e, A_g)$$

where A_i (input) is the abstract state at the beginning of s, A_o (output) is the abstract state after s terminates normally, A_r (return) is the state if it returns, and A_e (exits) maps exit numbers to the corresponding abstract states. The `goto` statements are handled by two maps from labels to abstract states: A_l (labels) and A_g (gotos), the first representing the states that can flow to a label defined in s, the second representing the states at `goto` statements executed by s. Figure 3 excerpts from the definition of T and shows all these components in action.

The `loop` case computes a post-fixpoint with widening and narrowing, starting at $\bot$ and iterating at most N_{widen} times. The `pfp` iterator is, classically, defined as

$$\texttt{pfp}\ F\ A\ N = \begin{cases} \top & \text{if } N = 0 \\ \texttt{narrow}\ F\ A\ N_{\text{narrow}} & \text{if } A \sqsupseteq F\ A \\ \texttt{pfp}\ F\ (A \nabla F\ A)\ (N-1) & \text{otherwise} \end{cases}$$

$$\texttt{narrow}\ F\ A\ N = \begin{cases} A & \text{if } N = 0 \\ \texttt{narrow}\ F\ (F\ A)\ (N-1) & \text{if } A \sqsupseteq F\ A \\ A & \text{otherwise} \end{cases}$$

Each iteration of `pfp` uses the widening operator ∇ provided by the abstract domain to speed up convergence. Once a post-fixpoint is found, F is iterated up to N_{narrow} times in the hope of finding a smaller post-fixpoint.

In both widening and narrowing iterations, we use "fuel" N to convince Coq that the recursions above are terminating, and to limit analysis time. We did not attempt to prove termination of iteration with widening: it would require difficult proofs over the widening operators of all our abstract domains [35], for no gain in soundness. Alternatively, we could delegate the computation of a candidate post-fixpoint A to an untrusted iterator written in Caml, then check $A \sqsupseteq F\ A$ in a verified Coq function. This would cost one more invocation of the F function, and it is unclear how the Caml implementation could be made more efficient than the `pfp` function above, written and verified in Coq.

$$
\begin{aligned}
T\ (x := e)\ (A_i, A_l) &= (\texttt{assign}\ x\ e\ A_i, \bot, \bot, \bot) \\
T\ (s; s')\ (A_i, A_l) &= (A'_o, A_r \sqcup A'_r, A_e \sqcup A_e, A_g \sqcup A'_g) \text{ where } \begin{cases} (A_o, A_r, A_e, A_g) = T\ s\ (A_i, A_l) \\ (A'_o, A'_r, A'_e, A'_g) = T\ s'\ (A_o, A_l) \end{cases} \\
T\ (\texttt{if}(e)\ s\ \texttt{else}\ s')\ (A_i, A_l) &= T\ s\ (\texttt{assume}\ e\ \texttt{true}\ A_i, A_l) \sqcup T\ s'\ (\texttt{assume}\ e\ \texttt{false}\ A_i, A_l) \\
T\ (\texttt{loop}\ s)\ (A_i, A_l) &= (\bot, A_r, A_e, A_g) \text{ where } (A_o, A_r, A_e, A_g) = \texttt{pfp}\ (\lambda(X_o, X_r, X_e, X_g).\ T\ s\ (A_i \sqcup X_o, A_l))\ \bot\ N_{\text{widen}} \\
T\ (\texttt{exit}\ n)\ (A_i, A_l) &= (\bot, \bot, (\lambda n'.\ \texttt{if}\ n' = n\ \texttt{then}\ A_i\ \texttt{else}\ \bot), \bot) \\
T\ (\texttt{block}\ s)\ (A_i, A_l) &= (A_o \sqcup A_e(0), A_r, (\lambda n.\ A_e(n+1)), A_g) \text{ where } (A_o, A_r, A_e, A_g) = T\ s\ (A_i, A_l) \\
T\ (\texttt{goto}\ L)\ (A_i, A_l) &= (\bot, \bot, \bot, (\lambda L'.\ \texttt{if}\ L' = L\ \texttt{then}\ A_i\ \texttt{else}\ \bot)) \\
T\ (L : s)\ (A_i, A_l) &= T\ s\ (A_i \sqcup A_l(L), A_l)
\end{aligned}
$$

Figure 3. Representative cases of the C#minor abstract interpreter

Optionally, the abstract interpreter can unroll loops on the fly: the N first iterations of the loop are analyzed independently in sequence; the remaining iterations are analyzed with a `pfp` fixpoint. This delays widening and gains precision. The unrolling factor N is currently given by an annotation in the source code.

In addition to the analysis of `loop` statements, a post-fixpoint is computed for every C#minor function to analyze `goto` statements. This function-global iteration ensures that the abstract states at `goto` statements are consistent with those assumed at the corresponding labeled statements. In other words, if s is the body of a function and $T\ s\ (A_i, A_l) = (A_o, A_r, A_e, A_g)$ is its analysis, the analysis iterates until $A_g(L) \sqsubseteq A_l(L)$ for every label L. When this condition holds, the abstraction of the function maps entry state A_i to exit state $A_o \sqcup A_r$ corresponding to the two ways a C#minor function can return (explicitly or by reaching the end of the function body).

Concerning functions, the abstract interpreter reanalyzes the body of a function at every call site, effectively unrolling the function definition on demand. We use fuel again to limit the depth of function unrolling. Moreover, since the state abstract domain does not handle recursion, it raises an alarm if a recursive call can occur.

The abstract interpreter is written in monadic style so that alarms can be reported during analysis. We use a logging monad: when an alarm is raised, it is collected in the log, but analysis continues. This is better than stopping at the first alarm like an error monad would do: often, widening reaches a state that causes an alarm, but the subsequent narrowing steps cause this alarm to go away.

The soundness proof for the abstract interpreter is massive, owing to the complexity of the C#minor language. To keep the proof manageable, we break it in two parts: 1- the definition and soundness proof of a suitable Hoare logic for C#minor, and 2- a proof that the abstract interpreter infers Hoare "triples" that are valid in this logic.

We first explain the approach in a simplified case, that of the IMP-like subset of C#minor, without `goto`, `exit` and `return`. In this subset, statements can only be entered at the beginning and exited at the end, and the transfer function is of the form $T\ s\ A = A'$. Intuitively, we expect this transfer function to be sound if, for any statement s and initial abstract state A such that the analysis $T\ s\ A$ raises no alarm, the execution of s started in any concrete state $\rho \in \gamma(A)$ does not go wrong, and if it terminates in state ρ', then $\rho' \in \gamma(T\ s\ A)$. The way we prove this property is first to show that if the analysis $T\ s\ A$ raises no alarms, then the weak Hoare triple $\{\gamma(A)\}\ s\ \{\gamma(T\ s\ A)\}$ can be derived in an appropriate program logic [5]. Then, we show that this program logic is sound with respect to the operational semantics of the language: if $\{P\}\ s\ \{Q\}$ can be derived in the logic, then the execution of s, started in a state satisfying P, does not go wrong, and if it terminates, it does so on a state satisfying Q.

The approach outlined above extends to the whole C#minor language, but not without elbow grease. C#minor statements can terminate in multiple ways: normally, or prematurely on an `exit`, `return` or `goto` statement. They can also be entered in two ways: at the beginning of the statement, or via a `goto` to a label defined within. Consequently, our program logic for C#minor manipulates Hoare "heptuples" of the form

$$\{P,\ P_l\}\ s\ \{Q,\ Q_r,\ Q_e,\ Q_g\}$$

where P is the precondition if s is entered normally, $P_l(L)$ the precondition if s is entered by a `goto` L, Q the postcondition if s terminates normally, $Q_r(v)$ the postcondition if s terminates by a `return` of value v, $Q_e(i)$ the postcondition if s terminates by `exit`(i), and $Q_g(L)$ the postcondition if s terminates by `goto` L. We omit the rules of this program logic from this paper, as they are similar to those of the program logics for Cminor and Clight by Appel and Blazy [2, 3] (without the separation logic aspects).

THEOREM 1 (Soundness of the abstract interpreter). *Assume that the analysis $T\ s\ (A_i, A_l)$ returns (A_o, A_r, A_e, A_g) without raising an alarm. Then, the heptuple*

$$\{\gamma(A_i), \gamma(A_l)\}\ s\ \{\gamma(A_o), \gamma(A_r), \gamma(A_e), \gamma(A_g)\}$$

is derivable in the C#minor program logic.

It remains to show the soundness of the program logic with respect to the continuation-based small-step semantics of C#minor. Taking inspiration from the work of Appel *et al.* on step-indexed semantics [3, 4], we say that a configuration (s, k, ρ) is safe for n steps if no sequence of at most n transitions starting from (s, k, ρ) triggers a run-time error: it either performs n transitions or reaches a final configuration after $n' < n$ transitions. We say that a continuation k is safe for n steps with respect to the postconditions $(Q,\ Q_r,\ Q_e,\ Q_g)$ if:

$$
\begin{aligned}
Q\ \rho &\Rightarrow (\texttt{skip}, k, \rho) \text{ safe for } n \text{ steps} \\
Q_r\ v\ \rho &\Rightarrow (\texttt{return}(v), k, \rho) \text{ safe for } n \text{ steps} \\
Q_e\ i\ \rho &\Rightarrow (\texttt{exit}(i), k, \rho) \text{ safe for } n \text{ steps} \\
Q_g\ L\ \rho &\Rightarrow (\texttt{goto}\ L, k, \rho) \text{ safe for } n \text{ steps}
\end{aligned}
$$

We can then state and prove soundness of the C#minor program logic as follows.

THEOREM 2 (Soundness of the program logic). *Assume that the heptuple* $\{P,\ P_l\}\ s\ \{Q,\ Q_r,\ Q_e,\ Q_g\}$ *is derivable in the program logic. Further assume that* k *is safe for* n *steps with respect to* $(Q,\ Q_r,\ Q_e,\ Q_g)$*. If* ρ *satisfies* P*, then* (s, k, ρ) *is safe for* n *steps. Moreover, if* ρ *satisfies* P_l L *and* L *is the label of sub-statement* s' *of* s*, in continuation* k' *extending* k*, then* (s', k', ρ) *is safe for* n *steps.*

5. The state abstract domain

The state abstraction in Verasco tracks values contained both in memory and in temporary variables. Each value is attached to an abstract memory cell, or "cell" for short, representing one unit of storage: either a temporary variable or a scalar part of a local or global variable. A scalar part is either the whole variable if it is of scalar type, or an element of scalar type in this variable (*e.g.*, a field in a structure, an element in an array). Therefore, a cell is given by the kind and the name of the variable it belongs to, the name of the function in which this variable is declared (unless it is global), and an offset and a value size (unless it is a temporary variable).

Cells:	$c ::= \texttt{temp}(f, t)$	the temporary t in function f
	$\mid \texttt{local}(f, x, \delta, \tau)$	slice of size τ at offset δ in the local variable x of function f
	$\mid \texttt{global}(x, \delta, \tau)$	slice of size τ at offset δ in the global variable x

The state abstract domain interacts with three specific abstract domains: an abstract value $(\texttt{tp}, \texttt{pt}, \texttt{nm})$ is a triple in $\texttt{types} \times \texttt{points-to} \times \texttt{num}$. First, type information is tracked for every memory cell in order to disambiguate, when possible, the behavior of C#minor operators. For example addition of integer or pointer values has various behaviors depending on the actual types of its arguments: adding two integers yields an integer; adding an integer to a pointer yields a pointer in the same block; and adding two pointers is undefined (therefore raises an alarm). In addition this type information is used to prove that values are not undefined, for instance when loaded from the memory.

Then, a points-to graph associates to each memory cell a set of memory blocks the cell may point to. A cell whose content is not a pointer may be associated with any set.

Finally, we track properties of the numbers contained in cells. The abstract state domain is parameterized by a relational numerical abstract domain following the interface from figure 2. Cells act as variables for this numerical domain, which therefore abstracts environments ρ mapping cells to numbers. The number associated with a cell is its content if it is of numerical type, or, if it contains a pointer, the byte offset part of this pointer. Driving a relational numerical abstract domain requires special care because we must feed it with rich numerical commands, as we now illustrate. Consider the following C#minor assignment occurring in a function `f`.

```
s := s + load(int32, &T + 8×i + 4)
```

Here, `s` and `i` are temporary variables (*i.e.*, the memory cell related to `s` is `temp(f,s)`) and `T` is a global variable of array type.

Type analysis computes that `&T` is a pointer whereas `s` and `i` are integers. Points-to analysis infers that `s` and `i` do not point to any block, but `&T` definitely points to the block of `T`. In order to feed the relational numerical domain with the assignment, we must transform the `load(int32, &T + 8×i + 4)` expression into its corresponding cells. We query the points-to domain to determine which memory blocks may be accessed during this load, yielding the block of `T`. We also query the numerical domain to obtain the possible values of `8×temp(f,i) + 4`, that is, the byte offsets that may be accessed during the load. Assume this query returns the set $\{4, 12\}$. We then approximate the load expression by the two cells `global(T,4,int32)` and `global(T,12,int32)`, send the numerical domain the two assignments

```
temp(f,s) := temp(f,s) + global(T,4,int32)
temp(f,s) := temp(f,s) + global(T,12,int32)
```

and finally take the join of the two new numerical abstract states thus obtained.

For a memory store $\texttt{store}(\tau, e_l, e_r)$, we perform a similar technique and generate a set $\{c_l^1, \ldots, c_l^p\}$ of p cells that may be modified during this store, and a set $\{e_r^1, \ldots, r_r^q\}$ of numerical expressions that capture the concrete value computed by expression e_r. When $p = 1$, we can perform a strong update that overwrites the previous value of the cell c_l^1. Otherwise, we must perform a weak (conservative) update and assume that any cell in $\{c_l^1, \ldots, c_l^p\}$ may still hold its old value after the update.

Concretization The types and points-to domains concretize to functions from cells to values whereas the numerical domain concretizes to purely numerical values. To combine them into one concretization, the last one is lifted using an agreement relation $\sim_{\texttt{tp}}$ (parameterized by the types information) ensuring that types and numerical values agree and that values bound by ρ match the numerical values bound by ρ'. In particular, if the value $\rho(c)$ is a pointer, the number $\rho'(c)$ is the byte offset part of this pointer.

More precisely, a function ρ from cells to values is in the concretization of an abstract value $(\texttt{tp}, \texttt{pt}, \texttt{nm})$ iff 1) it is in the concretization $\gamma(\texttt{tp})$ of the types domain `tp`, 2) it is in the concretization $\gamma(\texttt{pt})$ of the points-to domain (which means that for every cell c that contains a pointer value with a block pointer b, then $b \in \texttt{pt}(c)$) and 3) in the concretization $\gamma(\texttt{nm})$ of the numerical domain, there exists a concrete mapping ρ' from cells to numerical values that agrees with ρ:

$$\rho \in \gamma(\texttt{tp}, \texttt{pt}, \texttt{nm}) \Leftrightarrow \rho \in \gamma(\texttt{tp}) \cap \gamma(\texttt{pt}) \wedge \exists \rho' \in \gamma(\texttt{nm}),\ \rho \sim_{\texttt{tp}} \rho'$$

Finally, a function ρ from cells to values can be related to a C#minor execution state (call stack and memory state): the call stack determines a mapping from `temp` cells to values and from `local` and `global` cells to memory locations, and the memory state gives values to these locations.

Inter-procedural analysis To analyze a function call such as `x:=f(args)`, where `f` is an expression, the function is first resolved, then the abstract state at function entry is prepared, and finally the function body is (recursively) analyzed from this state. There, the analysis of `return` statements will compute the abstract state after the call and assignment to `x`.

Function resolution uses points-to information to compute a set of functions that expression `f` may point to.

The `push_frame` operation of the state abstract domain performs the assignments corresponding to argument passing: arguments are evaluated in the context of the caller and then assigned to local variables of the callee.

Since local variables are identified by their names and the functions they belong to, the call chain is remembered as an abstract stack. In addition, to be able to distinguish local from global variables in expressions, the set of variables local to each function is remembered in this abstract stack.

Symmetrically, the `pop_frame` operation is used when analyzing `return` e statements. The expression e is analyzed in the callee context and assigned to a temporary in the caller context. Then, `pop_frame` simulates the freeing of local variables on function exit: this consists in invalidating the information associated to them, as well as invalidating pointers that may point to them.

As an illustration, consider the following program.

```
int f(int *p) { return *p; }
int main(void) {
```

```
int x = 1;
int t = f(&x); return t; }
```

When analyzing the call to the function `f`, the argument expression is processed and the local variable `p` of `f` is assigned in the three abstract domains. In particular, `p` is assigned to the constant zero (the value of its offset) in the numerical domain and to the block `local(main,x)` in the points-to domain. Therefore, the return expression of `f` is known to access exactly one cell, about which information can be queried in the various domains. This information is then assigned back to the temporary `t` of `main`.

Progress verification At the same time we transform abstract states, we perform verifications to prove that every C#minor expression evaluates safely (without blocking) in its evaluation context. In particular, we check that every load and store is performed within bounds and with the correct alignment. We also check that every deallocation of local variables at function returns is performed on valid pointers, as well as various other side conditions (*e.g.* function pointers must have a null offset).

6. The numerical abstract domains

6.1 Intervals and congruences

The first numerical domains verified in Verasco are non-relational domains of intervals ($x \in [a, b]$) and congruences ($x \bmod n = p$). They abstract numbers consisting of the union of mathematical integers (type $\mathbb{Z}$) and double-precision floating-point (FP) numbers (IEEE754's `binary64` format). Treating both kinds of numbers at once facilitates the analysis of conversions between integers and FP numbers. Analyzing mathematical, exact integers instead of machine integers greatly simplifies the implementation and proof of abstract integer operations. In contrast, there is no benefit in analyzing FP numbers using a more mathematical type such as rationals: FP operations (with rounding) behave well with respect to FP ordering, and abstract operations that use rationals are costly.

Integer intervals Integer intervals are either $[a, b]$ with $a, b \in \mathbb{Z}$ or $\top$, standing for $(-\infty, \infty)$. We do not represent the semi-open intervals $[a, \infty)$ nor $(-\infty, b]$. The implementation of arithmetic and comparison operators over integer intervals is standard, with comparisons returning sub-intervals of $[0, 1]$. We go to great lengths, however, to derive tight intervals for bit-wise logical operations ("and", "or", "not", ...). The widening operator does not jump immediately to $(-\infty, \infty)$ but first tries to replace the upper bound by the next higher number in a list of well-chosen thresholds (zero and some powers of 2), and likewise replacing the lower bound by the next lower threshold [16]. The implementation and soundness proof build on Coq's `ZArith` library, which defines $\mathbb{Z}$ from first principles, essentially as lists of bits; no untrusted big integer library is involved.

Floating-point intervals Likewise, FP intervals are either $[a, b]$ where a and b are non-NaN but possibly infinite FP numbers, or $\top$. Not-a-Number (NaN) belongs in $\top$ but not in $[a, b]$. Interval analysis for FP arithmetic is complicated by the various special FP numbers (NaN, infinities, signed zeroes), but remains surprisingly close to reasoning over real numbers. IEEE754 specifies FP arithmetic operations as "compute the exact result as a real, then round it to a representable FP number". For example, addition of two finite FP numbers x, y is defined as $x \oplus y = \circ(x + y)$, where $\circ$ is one of the rounding modes defined in IEEE754. Crucially, all these rounding modes are *monotonic* functions. Therefore, if $x \in [a, b]$ and $y \in [c, d]$,

$$x \oplus y = \circ(x + y) \in [\circ(a + c), \circ(b + d)] = [a \oplus c, b \oplus d]$$

This property provides tight bounds for $\oplus$ and other FP operations, provided the rounding mode is known statically. This is the case for C#minor, which specifies round to nearest, ties to even, and gives no way for the program to dynamically change the rounding mode. Likewise, and unlike ISO C, C#minor specifies exactly the precision used by FP operations and the places where conversions between precisions occur. Therefore, the FP interval domain does not need to account for excess precision and possible double rounding.

The implementation and proof of the FP interval domain build on the Flocq library, which provides first-principles specifications and implementations of FP arithmetic in terms of mathematical integers and reals [9].

Integer congruences The congruence domain abstracts integers as pairs (n, m) of a modulus m and a constant n, representing all integers equal to n modulo m:

$$\gamma(n, m) = \{x \in \mathbb{Z} \mid \exists k, x = n + km\}$$

The case $m = 1$ corresponds to $\top$. The case $m = 0$ is meaningful and corresponds to constant propagation: $\gamma(n, 0) = \{n\}$. Tracking constants this way enables more precise analysis of multiplications and divisions. This domain of congruences is crucial to analyze the safety of memory accesses, guaranteeing that they are properly aligned.

6.2 From non-relational to relational domains

The non-relational domains of intervals and congruences share a common interface, specifying a type `t` of abstract values; a concretization γ to the union of $\mathbb{Z}$ integers and double-precision floats; functions to abstract constants; and functions to perform "forward" and "backward" analysis of C#minor operators. The following excerpt from the interface gives the flavor of the "forward" and "backward" functions.

```
forward_unop: i_unary_operation → t+⊤ → t+⊤+⊥;
backward_unop:
  i_unary_operation → t+⊤ → t+⊤ → t+⊤+⊥;
forward_unop_sound: ∀ op x x_ab, x ∈ γ x_ab →
  eval_iunop op x ⊆ γ (forward_unop op x_ab);
backward_unop_sound: ∀ op x x_ab, x ∈ γ x_ab →
  ∀ res res_ab, res ∈ γ res_ab →
  eval_iunop op x res →
  x ∈ γ (backward_unop op res_ab x_ab);
```

The "forward" functions compute an abstraction of the result given abstractions for the arguments of the operator. The "backward" functions take abstractions for the result and the arguments, and produce possibly better approximations for the arguments. For example, the backward analysis of $[0, 1] + [1, 2]$ with result $[0, 1]$ produces $[0, 0]$ for the first argument and $[1, 1]$ for the second one.

A generic domain transformer turns any non-relational domain satisfying this interface into a relational domain. Abstract environments are implemented as sparse finite maps from variables to non-$\top$ abstract values. Mappings from a variable to $\top$ are left implicit to make abstract environments smaller. The `assign` $x\ e\ A$ operation of the relational domain first computes an abstract value a for expression e, looking up abstract values $A(y)$ for variables y occurring in e and using the forward operators provided by the non-relational domain. The result is the updated abstract environment $A[x \leftarrow a]$.

The `assume` $e\ b\ A$ operation of the relational domain abstractly evaluates e "in reverse", starting with an expected abstract result that is the constant 1 if b is true and the constant 0 otherwise. The expected abstract result a_{res} is propagated to the leaves of e using the backward operators of the non-relational domain. When encountering a variable y in e, the abstract value of y is refined, giving $A' = A[y \leftarrow A(y) \sqcap a_{res}]$.

This simple construction makes it easy to add other non-relational domains and combine them with relational domains.

6.3 Convex polyhedra

To exercise the interface for relational numerical domains, Verasco includes a relational domain of convex polyhedra. It builds on the VPL library of Fouilhé *et al.* [18], which implements all required operations over convex polyhedra represented as conjunctions of linear inequalities with rational coefficients. VPL is implemented in Caml using GMP rational arithmetic, and therefore cannot be trusted. However, every operation produces a Farkas certificate that can easily be checked by a validator written and proved sound in Coq [19]. For example, the `join` operation, applied to polyhedras P_1, P_2, returns not only a polyhedron P but also Farkas certificates proving that the system of inequations $(P_1 \vee P_2) \wedge \neg P$ is unsatisfiable. Therefore, any concrete state $\rho \in \gamma(P_1) \cup \gamma(P_2)$ is also in $\gamma(P)$, establishing the soundness of `join`.

This is, currently, the only instance of verified validation a posteriori in Verasco. While simpler abstract domains can be verified directly with reasonable proof effort, verified validation is a perfect match for this domain, avoiding proofs of difficult algorithms and enabling efficient implementations of costly polyhedral computations.

6.4 Symbolic equalities

The relational domain of symbolic equalities records equalities $x = e_c$ between a variable x and a conditional expression e_c, as well as facts $e_c =$ `true` or $e_c =$ `false` about the Boolean value of an expression. Conditional expressions e_c extend numerical expressions e with zero, one or several if-then-else selectors in prenex position:

$$e_c ::= e \mid e_c \texttt{ ? } e_c : e_c$$

On its own, this domain provides no numerical information that can be used to prove absence of run-time errors. Combined with other numerical domains via the communication mechanism of section 7, symbolic equalities enable these other domains to analyze `assume` operations more precisely. A typical example comes from the way CompCert compiles C's short-circuit Boolean operators `&&` and `||`. The pass that pulls side effects outside of expressions transforms these operators into assignments to temporary Boolean variables. For example, "`if (f(x) > 0 && y < z) { s; }`" becomes, in C#minor,

```
t1 = f(x);
if (t1 > 0) { t2 = y < z; } else { t2 = 0; }
if (t2) { s; }
```

The symbolic equality domain infers $\texttt{t}_2 = \texttt{t}_1 > 0 \texttt{ ? } \texttt{y} < \texttt{z} : 0$ at the second `if`. Assuming $\texttt{t}_2 =$ `true` in `s` adds no information in the interval and polyhedra domains. Noticing this fact, these domains can query the domain of equalities, obtain the symbolic equality over $\texttt{t}_2$ above, and learn that $\texttt{y} < \texttt{z}$ and $\texttt{t}_1 > 0$.

A set of equalities $x = e_c$ and facts $e_c = b$ concretizes to all concrete environments that validate these equations:

$$\gamma(Eqs, Facts) = \left\{ \rho \;\middle|\; \begin{array}{l} (x = e_c) \in Eqs \quad \Rightarrow \rho(x) \in \texttt{eval}\ \rho\ e_c \\ (e_c = b) \in Facts \Rightarrow b \in \texttt{eval}\ \rho\ e_c \end{array} \right\}$$

A new equation is added by `assign`, and a new fact is added by `assume`. All equations and facts involving variable x are removed when doing `assign` or `forget` over x. We do not track previous values of assigned variables the way value numbering analyses do. Likewise, we do not treat equalities between variables $x = y$ specially.

The clever operation in this domain is the join between two abstract states: this is where conditional expressions are inferred. Computing (optimal) least upper bounds between sets of symbolic equalities is known to be difficult [21], so we settle for an overapproximation. Equations and facts that occur in both abstract states (using syntactic equality for comparison) are kept. If one state contains an equality $x = e'_c$ and a fact $e_c =$ `true`, and the other state contains $x = e''_c$ and $e_c =$ `false`, the equality $x = e_c \texttt{ ? } e'_c : e''_c$ is added to the joined state. All other equalities and facts are discarded. Widening is similar to join, except that new conditional expressions are not inferred.

6.5 Handling machine integers

Numerical domains such as intervals and polyhedra are well understood as abstractions of unbounded mathematical integers. However, most integer types in programming languages are bounded, with arithmetic overflows treated either as run-time errors or by "wrapping around" and taking the result modulo the range of the type. In C#minor, integer arithmetic is defined modulo 2^N with $N = 32$ or $N = 64$ depending on the operation. Moreover, C#minor does not distinguish between signed and unsigned integer types: both are just N-bit vectors. Some integer operations such as division or comparisons come in two flavors, one that interprets its arguments as unsigned integers and the other as signed integers; but other integer operations such as addition and multiplication are presented as a single operator that handles signed and unsigned arguments identically.

All these subtleties of machine-level integer arithmetic complicate static analysis. For specific abstract domains such as intervals and congruences, ad hoc approaches are known, such as strided intervals [36], wrapped intervals [31], or reduced product of two intervals of $\mathbb{Z}$, tracking signed and unsigned interpretations respectively [8]. These approaches are difficult to extend to other domains, especially relational domains. In Verasco, we use a more generic construction that transforms any relational domain over mathematical integers $\mathbb{Z}$ into a relational domain over N-bits machine integers with modulo-2^N arithmetic.

We first outline the construction on a simple, non-relational example. Consider the familiar domain of intervals over $\mathbb{Z}$, with concretization $\gamma([l,h]) = \{x : \mathbb{Z} \mid l \leq x \leq h\}$. To adapt this domain to the analysis of 4-bit machine integers (type `int4`), we keep the same abstract values $[l,h]$ with $l, h \in \mathbb{Z}$, but concretize them to machine integers as follows:

$$\gamma_m([l,h]) = \{b : \texttt{int4} \mid \exists n : \mathbb{Z},\ n \in \gamma([l,h]) \wedge b = n \bmod 2^4\}$$

In other words, the mathematical integers in $\gamma([l,h])$, that is, $l, l+1, \ldots, h-1, h$, are projected modulo 16 into bit vectors.

All arithmetic operations that are compatible with equality modulo 2^4 can be analyzed using the standard abstract operations over $\mathbb{Z}$-intervals. For example, 4-bit addition `add` is such that $\texttt{add}\ b_1\ b_2 = (b_1 + b_2) \bmod 2^4$. If it is know that $x \in [0,2]$, we analyze $\texttt{add}\ x\ 15$ like we would analyze $x + 15$ (addition in $\mathbb{Z}$), obtaining $[0,2] + [15,15] = [15,17]$. This interval $[15,17]$ concretizes to three 4-bit vectors, $\{15,0,1\}$ with unsigned interpretation and $\{-1,0,1\}$ with signed interpretation. An overflow occurred in the unsigned view, but the $\mathbb{Z}$-interval arithmetic tracks it correctly. The same technique of analyzing machine operations as if they were exact works for addition, subtraction, and bitwise operations (and, or, not), without loss of precision, and also for multiplication and left shift, possibly with loss of precision.

Other arithmetic operations such as division, right shifts, and comparisons are not compatible with equality modulo 2^4. (These are exactly the operations that must come in two flavors, unsigned and signed, at the machine level.) For example, $-1 \leq 0$ but $15 \not\leq 0$, even though $-1 = 15 \pmod{2^4}$. To analyze these operations, we first try to reduce the intervals for their arguments to the interval $[L, H)$ expected by the operation: $[0, 15]$ for an unsigned operation and $[-8, 7]$ for a signed operation. To this end, we just add an appropriate multiple of 16 to the original interval; this operation does not change its γ_m concretization. Continuing the example

above, $\texttt{add}\ x\ 15 \in [15, 17]$, viewed as a signed integer, can be reduced to the interval $\texttt{add}\ x\ 15 \in [-1, 1]$ by subtracting 16. Therefore, the signed comparison $\texttt{le\_s}\ (\texttt{add}\ x\ 15)\ 4$ can be analyzed as

$$\texttt{le\_s}\ (\texttt{add}\ x\ 15)\ 4 \in ([-1, 1] \leq [4, 4]) = \texttt{true}$$

If we need to view $\texttt{add}\ x\ 15 \in [15, 17]$ as an unsigned 4-bit integer, the best contiguous interval we can give is $[0, 15]$. Therefore, the unsigned comparison $\texttt{le\_u}\ (\texttt{add}\ x\ 15)\ 4$ is analyzed as

$$\texttt{le\_u}\ (\texttt{add}\ x\ 15)\ 4 \in ([0, 15] \leq [4, 4]) = \top$$

In the signed comparison case, the unsigned overflow during the computation of $\texttt{add}\ x\ 15$ is benign and does not harm the precision of the analysis. In the unsigned comparison case, the overflow is serious and makes the result of the comparison unpredictable.

On the example of intervals above, our construction is very close to wrapped intervals [31]. However, our construction generalizes to any relational domain that satisfies the Verasco interface for "ideal" numerical domains. Such domains abstract ideal environments $\rho : \texttt{var} \to \mathbb{Z} + \texttt{float64}$ where integer-valued variables range over mathematical integers, not machine integers. Consider such a domain, with abstract states A and concretization function γ. We now build a domain that abstracts machine environments $\rho_m : \texttt{var} \to \texttt{int32}+\texttt{int64}+\texttt{float64}$ where integer-valued variables are 32- or 64-bit machine integers with wrap-around arithmetic. We keep the same type A of abstract states, but interpret them as sets of machine environments via

$$\gamma_m(A) = \{\rho_m \mid \exists \rho \in \gamma(A),\ \forall v,\ \rho_m(v) \cong \rho(v)\}$$

The agreement relation $\cong$ between a machine number and an ideal number is defined as

$$\begin{array}{rcll} b \in \texttt{int32} & \cong & n \in \mathbb{Z} & \text{iff } b = n \bmod 2^{32} \\ b \in \texttt{int64} & \cong & n \in \mathbb{Z} & \text{iff } b = n \bmod 2^{64} \\ f \in \texttt{float64} & \cong & f' \in \texttt{float64} & \text{iff } f = f' \end{array}$$

The abstract operations $\texttt{assign}_m$, $\texttt{assume}_m$ and $\texttt{forget}_m$ over the machine domain are defined in terms of those of the underlying ideal domain after translation of the numerical expressions involved:

$$\begin{aligned} \texttt{assign}_m\ x\ e\ A &= \texttt{assign}\ x\ [\![e]\!]_A\ A \\ \texttt{assume}_m\ e\ b\ A &= \texttt{assume}\ [\![e]\!]_A\ b\ A \\ \texttt{forget}_m\ x\ A &= \texttt{forget}\ x\ A \end{aligned}$$

The translation of expressions inserts just enough normalizations so that the ideal transformed expression $[\![e]\!]_A$ evaluates to an ideal number that matches (up to the $\cong$ relation) the value of e as a machine number. Variables and constants translate to themselves. Arithmetic operators that are compatible with the $\cong$ relation, such as integer addition and subtraction, and all FP operations, translate isomorphically. Other arithmetic operators have their arguments reduced in range, as explained below.

$$\begin{aligned} &[\![x]\!]_A = x \\ &[\![\texttt{add}\ e_1\ e_2]\!]_A = [\![e_1]\!]_A + [\![e_2]\!]_A \\ &[\![\texttt{le\_u}\ e_1\ e_2]\!]_A = \\ &\quad \texttt{reduce}_A [\![e_1]\!]_A\ [0, 2^{32}) \leq \texttt{reduce}_A\ [\![e_2]\!]_A\ [0, 2^{32}) \\ &[\![\texttt{le\_s}\ e_1\ e_2]\!]_A = \\ &\quad \texttt{reduce}_A\ [\![e_1]\!]_A\ [-2^{31}, 2^{31}) \leq \texttt{reduce}_A\ [\![e_2]\!]_A\ [-2^{31}, 2^{31}) \end{aligned}$$

The purpose of $\texttt{reduce}_A\ e\ [L, H)$, where $[L, H)$ is an interval of width 2^N, is to reduce the values of e modulo 2^N so that they fit the interval $[L, H)$. To this end, it uses a $\texttt{get\_itv}\ e\ A$ operation of the ideal numerical domain that returns a variation interval of e. From this interval, it determines the number q of multiples of 2^N that must be subtracted from e to bring it back to the interval $[L, H)$. This is not always possible, in which cases $[L, H)$ is returned as the reduced expression. (Remember that numerical expressions in Verasco are nondeterministic and use intervals as constants.)

$$\begin{aligned} &\texttt{reduce}_A\ e\ [L, H) = \\ &\quad \texttt{let}\ [l, h) = \texttt{get\_itv}\ e\ A\ \texttt{in} \\ &\quad \texttt{if}\ h - l \geq 2^N\ \texttt{then}\ [L, H)\ \texttt{else} \\ &\qquad \texttt{let}\ q = \lfloor (l - L)/2^N \rfloor\ \texttt{in} \\ &\qquad \texttt{if}\ h + q.2^N \leq H\ \texttt{then}\ e - q.2^N\ \texttt{else}\ [L, H) \end{aligned}$$

The translation of expressions is sound in the following sense.

LEMMA 3. *Assume $\rho_m \in \gamma_m(A)$, $\rho \in \gamma(A)$ and $\rho_m(x) \cong \rho(x)$ for all variables x. Then, for all machine expressions e,*

$$v_m \in \texttt{eval}_m\ \rho_m\ e \Rightarrow \exists v,\ v \in \texttt{eval}\ \rho\ [\![e]\!]_A\ \wedge v_m \cong v$$

It follows that the $\texttt{assign}_m$ and $\texttt{assume}_m$ operations of the transformed domain are sound.

7. Communication between domains

Several abstract domains are used in order to keep track of different kinds of properties. For example, we need interval information to check that array accesses are within bounds. We use a congruence domain to check alignment of memory accesses. A domain of symbolic equalities helps us dealing with boolean expressions.

All those domains need to communicate. For example, only the interval domain is able to infer numerical information for all operators, including bitwise operators, division and FP operations; however, all other domains may need to use this information. Another example is the symbolic equalities domain: if the condition of a test is just a variable, another domain can substitute the variable with a boolean expression provided by this symbolic domain in order to refine their abstract states.

The classic approach to combining two abstract domains and make them exchange information is the *reduced product* [15]. Implementations of reduced products tend to be specific to the two domains being combined, and are difficult to scale to the combination of $n > 2$ domains. Reduced products are, therefore, not a good match for the modular architecture of Verasco. Instead, we use a system of inter-domain communications based on channels, inspired by that of ASTRÉE [17]. We define an input channel as follows:

```
Record in_chan : Type :=
  { get_itv: iexpr var → IdealIntervals.abs+⊥;
    get_eq_expr: var → option (mux_iexpr var) }.
```

An input channel is a record of functions. Each function corresponds to a specific kind of query: `get_itv` returns an interval of variation for an expression, while `get_eq_expr` returns an expression that evaluates to the same value as the variable it is called on. This type of channels is meant to be extended when needed. For example, one could add a query for a linear approximation of an expression, which could be answered by a dedicated domain and used by linear relational domains.

Like abstract states, channels have a concretization function. Intuitively, an environment $\rho : \texttt{var} \to \mathbb{Z} + \texttt{float64}$ is in the concretization of a channel if all answers to queries are valid in ρ:

```
Record in_chan_gamma chan ρ : Prop :=
  { get_itv_correct:
      ∀ e, eval_iexpr ρ e ⊆ γ (chan.(get_itv) e);
    get_eq_expr_correct:
      ∀ x e, chan.(get_eq_expr) x = Some e →
             eval_mux_iexpr ρ e (ρ x) }.
```

Channels are used by domains when they need information from another domain. When an operation of a domain receives as argument an abstract value, it also receives a channel corresponding to

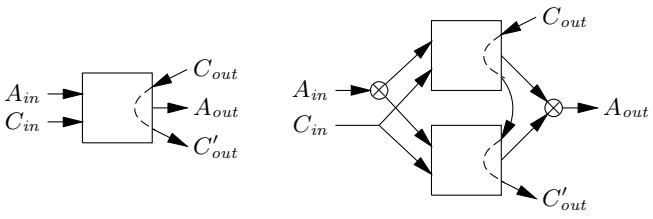

Figure 4. Communication channels between abstract operators. Left: single operator. Right: composition of two operators from different domains.

the same concrete environments. (Channel C_{in} in figure 4.) Symmetrically, each transfer function that returns a new abstract value also returns a channel C'_{out} that other domains can query to obtain information on the state after the execution of the transfer function. Finally, yet another channel C_{out} is provided as extra argument to each transfer function. Querying this channel will provide information on the state after the execution of transfer functions from other domains. Only abstract domains having already computed these function can answer these queries. In other words, the domain operation produces C'_{out} by enriching information already present in C_{out} with information of its own. For example, the `assign` transfer function, which corresponds, in the concrete, to an assignment of an expression to a variable, has the following type:

```
assign: var → iexpr var → t * in_chan →
        in_chan → (t * in_chan)+⊥;
```

The first and second arguments are the assigned variable and expression. The third argument is a pair representing the initial states: an abstract value and the channel C_{in}. The fourth argument is the channel C_{out} representing the current information about the final state after the assignment. If no contradiction is found, `assign` returns the final abstract state and the enriched channel C'_{out}. The specification of `assign` is as follows:

```
assign_correct: ∀ x e ab chan ρ n,
    n ∈ eval_iexpr ρ e →
    ρ ∈ γ ab →
    (upd ρ x n) ∈ γ chan →
    (upd ρ x n) ∈ γ (assign x e ab chan);
```

Here, we extend γ to pairs of abstract values and channels, taking γ `(x, y) =` γ `x` $\cap$ γ `y` and using Coq type classes. This specification of `assign` is analogous to the one in figure 2. The difference is that we add an hypothesis stating that the two channels given as parameters are correct with respect to initial and final states respectively. Moreover, we demand that the returned channel be correct with respect to the final state.

An implementation of such a specification has to create a channel. For each query, the implementation can choose to forward it to the channel received as its fourth parameter, effectively forwarding it to another domain, or to answer it using its own information, or to do both and combine the information. For example, an interval domain will answer `get_itv` queries but not `get_eq_expr` queries, forwarding the latter to other domains.

This interface for channel-aware transfer functions such as `assign` makes it easy to combine two domains and have them communicate. Verasco provides a generic combinator (pictured as $\otimes$ in figure 1) that takes two abstract domains over ideal numerical environments and returns a product domain where information coming from both domains is stored, and where the two domains can communicate via channels. The definition of `assign` for the product is the following:

```
assign v e (ab:(A*B)*in_chan) chan :=
  let '((a, b), abchan) := ab in
  (* Computation on the first component *)
  do_bot retachan <- assign v e (a, abchan) chan;
  let '(reta, chan) := retachan in
  (* Computation on the second component,
     using the new input channel *)
  do_bot retbchan <- assign v e (b, abchan) chan;
  let '(retb, chan) := retbchan in
  NotBot ((reta, retb), chan)
```

The "plumbing" implemented here is depicted in figure 4, right part. As shown there, the C_{out} channel passed to the second abstract domain is the C'_{out} channel generated by the first abstract domain. This enables the second abstract domain to query the first one.

Using this product construction, we can build trees (nested products) of cooperating domains. As depicted in figure 1, the input to the "$\mathbb{Z} \to$ `int`" domain transformer described in section 6.5 is such a combination of numerical domains. Abstract states of this combination are pairs of, on the one hand, nested pairs of abstract states from the individual numerical domains, and, on the other hand, a channel. The channel is not only used when calling abstract transfer functions, but also directly in order to get numerical information such as variation intervals. When the "$\mathbb{Z} \to$ `int`" domain transformer calls a transfer function, it simply passes as initial C_{out} channel a "top" channel whose concretization contains all concrete environments: `assign x e v in_chan_top`.

One final technical difficulty is comparison between abstract states, such as the subsumption test $\sqsubseteq$ used during post-fixpoint computation. In the upper layers, abstract states comprise (nested pairs of) abstract values plus a channel. Therefore, it seems necessary to compare two channels, or at least one channel and one abstract state. However, channels being records of functions, comparison is not decidable. Our solution is to maintain the invariant that *channels never contain more information than what is contained in the abstract values they are paired with*. That is, at the top of the combination of domains, when we manipulate a pair `(ab, chan)` of an abstract value and a channel, we will make sure that γ`(ab)` $\subseteq$ γ`(chan)` holds. In order to check whether one such pair `(ab1, chan1)` is smaller than another `(ab2, chan2)`, we only need to check that γ`(ab1, chan1)` $\subseteq$ γ`(ab2)`, which is easily decidable. Thus, the type of the comparison function for abstract values is:

```
leb: t * in_chan → t → bool
```

Note that it is useful to provide `leb` with a channel for its first argument: an abstract domain can, then, query other domains in order to compare abstract values.

However, the constraint γ`(ab)` $\subseteq$ γ`(chan)` is too strong for real abstract domains: it makes it impossible for a domain to forward a query to another domain. Instead, we use a weaker constraint for every transfer function. In the case of `assign`, we prove:

```
∀ x e in chan0 ab chan,
  assign x e in chan0 = NotBot (ab, chan) →
  γ chan0 ∩ γ ab ⊆ γ(chan)
```

That is, the returned channel contains no more information than what is contained in the returned abstract value and the given channel. When `chan0` is `in_chan_top`, it follows that γ`(ab)` $\subseteq$ γ`(chan)`, ensuring the soundness of the `leb` comparison function.

The property that limits the amount of information contained in channels is useful beyond the proof of soundness for comparisons: it is also a sanity check, ensuring that the returned channel only depends on the abstract value and on the channel given as the last argument, but not for instance on channels or abstract values

	Specs	Proofs	Overall
Interfaces	1081	139	3%
Abstract interpreter	2335	2204	13%
State abstraction	2473	3563	17%
Numerical domains	7805	8563	48%
Domain combinators	1305	1826	9%
Intervals	1489	2224	10%
Congruences	403	288	2%
Polyhedra (validator)	4038	3598	22% (from [19])
Symbolic equalities	570	627	3%
Miscellaneous libraries	3153	3329	19%
Total	16847	17040	33887 lines

Table 1. Size of the Coq development

previously computed. This is important for efficiency, because if the closures contained in the channel made references to previous channels or abstract values, this would make the analyzer keep old abstract values in memory, leading to bad memory behavior.

8. Implementation and mechanization

The Coq development for Verasco is about 34 000 lines, excluding blanks and comments. An additional 6 000 lines of Caml implement the operations over poyhedra that are validated a posteriori. The Coq sources split equally between proof scripts, on the one hand, and algorithms, specifications and statements of theorems on the other. Table 1 shows the relative sizes of the various components of Verasco. The parts reused from CompCert (e.g., syntax and semantics of C#minor) are not counted.

The interfaces that provide the backbone of Verasco are pleasantly lean. The bulk of the development is the abstract domains for states and (especially) for numbers, which involve large case analyses and difficult proofs over integer and F.P. arithmetic.

The Coq proofs are essentially constructive. The axiom of excluded middle and the axiom of functional extensionality are used in a few places, for convenience rather than by necessity. However, floating-point arithmetic is specified using Coq's theory of real numbers, which relies on classical logic axioms.

Except for the operations over polyhedra, the algorithms used by Verasco are implemented directly in Coq's specification language as function definitions in purely functional style. An executable analyzer is obtained by automatic extraction of Caml code from these function definitions and those of CompCert.

9. Experimental results

We conducted preliminary experiments with the executable C#minor static analyzer obtained as described in section 8. We ran the analyzer on a number of small test C programs (up to a few hundred lines). The purpose was to verify the absence of run-time errors in these programs. To exercise further the analyzer, we added support for a built-in function `verasco_assert(e)` to explicitly ask the analyzer to prove invariants that are expressible as C expressions `e`.

To model inputs, we added support for two other built-in functions: `any_int64` and `any_double`, which nondeterministically return a value of the requested type. They are often coupled to the built-in function `verasco_assume(b)`[2] to further constrain their results. These functions are also used to model library functions, whose code is not available or trusted.

[2] We only consider program executions where the boolean expression `b` is true when `verasco_assume(b)` is reached.

The following describes representative programs that we analyzed. The other examples have similar characteristics and lead to comparable observations.

Function integration The example `integr.c` is a small program adapted from a CompCert benchmark. Most of its code is given below.

```
typedef double (*fun)(double);
fun functions[N] = { id, square, fabs, sqrt };

double
integr(fun f, double low, double high, int n) {
    double h, x, s; int i;
    h = (high - low) / n; s = 0;
    for (i = n, x = low; i > 0; i--, x += h)
        s += f(x);
    return s * h;
}

int main(void) {
    for (int i = 0; i < any_int(); ++i) {
        double m = any_double();
        verasco_assume( 1. <= m );
        int n = any_int();
        verasco_assume (0 < n);
        integr(functions[i % N], 0., m, n);
  }
  return 0;
}
```

This program repeatedly computes an approximation of the integral of a function between zero and some number greater than one. The function in question is picked from a constant array. It stresses various aspects of the analyzer such as function pointers, arrays, floating point and machine arithmetic.

Numerical simulations Two programs of a few hundred lines taken from the CompCert benchmark, `nbody.c` and `almabench.c`, feature heavy numerical (floating point) computations and array manipulation.

Cryptographic routines The `smult.c` example performs scalar multiplication. It is taken from the cryptography library NaCl. Scalars and group elements are stored in arrays of bytes or unsigned integers. Many of these arrays are initialized within a loop. Understanding that such an array is indeed properly initialized at the end of the loop would require a dedicated analysis beyond the scope of this paper [22]. Instead, we annotated the program to request full unrolling of these loops during analysis, thus preventing fixpoint computations.

Preliminary results On the examples described above, Verasco was able to prove the absence of run-time errors. This is encouraging, since these examples exercise many delicate aspects of the C language: arrays, pointer arithmetic, function pointers, and floating-point arithmetic. As summarized in the table below, analysis times are high. Section 11 discusses possible directions to speed up the analyzer.

Program	Size	Time[3]
`integr.c`	42 lines	0.1s
`smult.c`	330 lines	86.0s
`nbody.c`	179 lines	30.8s
`almabench.c`	352 lines	328.6s

[3] Xeon E3-1240 processor, 3.4GHz, 8Mo cache, 16Go RAM.

10. Related work

Early work on the mechanized verification of static analyses was conducted in the framework of dataflow analyses. This includes Klein and Nipkow's verified Java bytecode verifier [25], Cachera *et al.*'s Coq formalization of dataflow analysis [11], and the verified dataflow analyses that support optimizations in CompCert [27]. Only non-relational abstract domains are considered, and there is no widening to accelerate the convergence of fixpoint iterations. Hofmann *et al.* verify a generic fixpoint solver usable in this context [24].

The first attempt to mechanize abstract interpretation in its full generality is Monniaux's master's thesis [30]. Using the Coq proof assistant and following the orthodox approach based on Galois connections, he runs into difficulties with α abstraction functions being nonconstructive, and with the calculation of abstract operators being poorly supported by Coq. Later, Pichardie's Ph.D. thesis [34, 35] mechanizes the γ-only presentation of abstract interpretation that we use in Verasco. Widening in fixpoint iterations as well as relational domains are supported, but the applications to Java static analysis presented in [34] use only non-relational domains. Blazy *et al.* use Pichardie's approach to verify an interval analysis for the RTL intermediate language of CompCert [8]. Bertot [5] and Nipkow [33] give alternate presentations of this approach, respectively in Coq and in Isabelle/HOL, resulting in pedagogical abstract interpreters for the IMP mini-language.

Many of the formalizations mentioned above run into serious complications to prove the termination of widened fixpoint iteration, using either integer measures [33] or well-founded orderings [11, 35]. The proof obligations related to termination account for much of the difficulty of constructing modular hierarchies of abstract domains. In Verasco, we forego termination proofs and elect to verify partial correctness only.

The ongoing project closest to Verasco in terms of ambitions is SparrowBerry by Cho *et al.* [13]. Rather than proving the soundness of a static analyzer, they follow a proof-carrying code approach: the existing, untrusted SparseSparrow C static analyzer is instrumented to produce analysis certificates, which are checked for correctness by a validator proved correct in Coq. Validation *a posteriori* reduces the overall proof effort to some extent. Indeed, we use it locally in Verasco to implement the polyhedral domain. However, we were reluctant to validate the analysis of a whole program by fear that the resulting certificates would be very large and take too long to check. (One of the claimed reasons why Astrée scales [16] is that it keeps few abstract states in memory at any given time.) This fear may be unfounded in light of the very good checking times reported for SparrowBerry [13]. We note, however, that SparrowBerry implements only one, non-relational domain (integer intervals), and that it does not handle a number of C features that Verasco handles (floating-point arithmetic, unions, wrap-around in unsigned integer arithmetic, pointer comparisons, and function pointers).

11. Conclusions and perspectives

Here is the final theorem in the Coq verification:

```
Theorem vanalysis_correct :
 forall prog res tr,
 vanalysis prog = (res, nil) →
 program_behaves (semantics prog) (Goes_wrong tr) →
 False.
```

Paraphrasing: if the whole-program analyzer `vanalysis` returns an empty list of alarms as its second result, the execution of the program cannot get stuck on a run-time error, regardless of the trace of inputs `tr` given to the program.

Verasco is an ongoing experiment, but at this stage of the project it already demonstrates the feasibility of formally verifying a realistic static analyzer based on abstract interpretation. Having to write Coq specifications and proofs did not preclude Verasco from handling a large source language (the subset of C typically used in critical embedded systems) nor from supporting multiple, nontrivial numerical abstract domains. Rather, this requirement that everything is specified and proved sound steered us towards a highly modular architecture, with well-specified interfaces and generic combinators to mix and adapt abstract domains. Most of these domains and domain combinators can be reused in other tool verification projects; some of them also act as reference implementations for advanced techniques for which no implementation was publically available before, such as the channel-based combination of abstractions.

The current Verasco analyzer can be extended in many directions. First, the algorithmic efficiency of the analyzer needs improvement. Currently, Verasco can take several minutes to analyze a few hundred lines of C. There are numerous sources of inefficiencies, which we are currently analyzing. One is Coq's integer and FP arithmetic, built from first principles (lists of bits). It should be possible to parameterize Verasco over arithmetic libraries such that more efficient big integer libraries and processor-native FP numbers can be used as an alternative. Another potential source of inefficiency is the purely functional data structures (AVL trees, radix-2 trees) used for maps and sets. ASTRÉE scales well despite using similar, purely functional data structures, but only because their implementations take advantage from *physical sharing* within and between tree data structures [16]. In Verasco, we already obtained good speedups by preserving preexisting sharing between the arguments of join operations. Going further, we could re-share a posteriori using general hash-consing [10].

Extending Verasco on the source language side, dynamic memory allocation could probably be handled by extending the memory abstraction so that one abstract memory cell can stand for several concrete memory locations, such as all the blocks created by a `malloc` inside a loop. Recursion raises bigger challenges: both the memory abstraction and the abstract interpreter would require heavy modifications so that they can merge the abstract states from multiple, simultaneously-active invocations of a recursive function, perhaps using call strings in the style of k-CFA [32].

Concerning the C#minor abstract interpreter, precision of the analysis would be improved by smarter unverified heuristics for loop unrolling, not relying on programmer-inserted annotations. The more elegant and powerful method would certainly be provided by a general trace partitioning mechanism [37]. Also, the nested fixpoint iterations arising out of the analysis of nested loops can be costly. It should be possible to accelerate convergence by starting the inner iterations not at $\bot$ but at the post-fixpoint found at the previous outer iteration. The starting points could be provided by an external, untrusted oracle written in imperative Caml.

The hierarchy of numerical abstractions is set up to accommodate new abstract domains easily. High on our wish list is a domain of *octagons*: linear inequalities of the form $\pm x \pm y \leq c$ [28]. Octagons are algorithmically more efficient than convex polyhedra. Moreover, using floating-point numbers for the coefficients of their difference bound matrices, octagons can infer inequalities involving floating-point program variables and not just integer variables. Such a use of FP arithmetic in a static analyzer is fraught with danger of numerical inaccuracy and deserves a formal proof of soundness. Just like convex polyhedra, octagons also need expression linearization heuristics [29] to extract more information out of nonlinear expressions.

Acknowledgments

This work is supported by Agence Nationale de la Recherche, grant ANR-11-INSE-003. A. Fouilhé, S. Boulmé, D. Monniaux and M. Périn developed the VPL library and the verified validator mentioned in section 6.3. J. Feret and A. Miné provided advice based on their experience with Astrée.

References

[1] A. Ahmed, A. W. Appel, C. D. Richards, K. N. Swadi, G. Tan, and D. C. Wang. Semantic foundations for typed assembly languages. *ACM Trans. Program. Lang. Syst.*, 32(3), 2010.

[2] A. W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, 2014.

[3] A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In *TPHOLs*, volume 4732 of *LNCS*, pages 5–21. Springer, 2007.

[4] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

[5] Y. Bertot. Structural abstract interpretation: A formal study using Coq. In *Language Engineering and Rigorous Software Development, LerNet Summer School*, pages 153–194. Springer, 2008.

[6] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.

[7] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *Formal Methods*, volume 4085 of *LNCS*, pages 460–475. Springer, 2006.

[8] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *SAS*, volume 7935 of *LNCS*, pages 324–344. Springer, 2013.

[9] S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *ARITH*, pages 243–252. IEEE, 2011.

[10] T. Braibant, J.-H. Jourdan, and D. Monniaux. Implementing and reasoning about hash-consed data structures in Coq. *J. Autom. Reasoning*, 53(3):271–304, 2014.

[11] D. Cachera, T. P. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theor. Comput. Sci.*, 342(1):56–78, 2005.

[12] A. Chlipala. Modular development of certified program verifiers with a proof assistant,. *J. Funct. Program.*, 18(5-6):599–647, 2008.

[13] S. Cho, J. Kang, J. Choi, C.-K. Hur, and K. Yi. SparrowBerry: A verified validator for an industrial-strength static analyzer. `http://ropas.snu.ac.kr/sparrowberry/`, 2013.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[15] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, page 269–282. ACM, 1979.

[16] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does Astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[17] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In *ASIAN*, volume 4435 of *LNCS*, pages 272–300. Springer, 2006.

[18] A. Fouilhé, D. Monniaux, and M. Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In *SAS*, volume 7935 of *LNCS*, pages 345–365. Springer, 2013.

[19] A. Fouilhé and S. Boulmé. A certifying frontend for (sub)polyhedral abstract domains. In *VSTTE*, volume 8471 of *LNCS*, pages 200–215. Springer, 2014.

[20] D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In *ITP*, volume 7406 of *LNCS*, pages 99–115. Springer, 2012.

[21] S. Gulwani, A. Tiwari, and G. C. Necula. Join algorithms for the theory of uninterpreted functions. In *FSTTCS*, volume 3328 of *LNCS*, pages 311–323. Springer, 2004.

[22] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.

[23] P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. In *VSTTE*, volume 7152 of *LNCS*, pages 2–17. Springer, 2012.

[24] M. Hofmann, A. Karbyshev, and H. Seidl. Verifying a local generic solver in Coq. In *SAS*, volume 6337 of *LNCS*, pages 340–355, 2010.

[25] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[26] X. Leroy. Formal verification of a realistic compiler. *Comm. ACM*, 52(7):107–115, 2009.

[27] X. Leroy. A formally verified compiler back-end. *J. Automated Reasoning*, 43(4):363–446, 2009.

[28] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[29] A. Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI*, volume 3855 of *LNCS*, pages 348–363. Springer, 2006.

[30] D. Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Master's thesis, U. Paris 7, 1998.

[31] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In *APLAS*, volume 7705 of *LNCS*, pages 115–130. Springer, 2012.

[32] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[33] T. Nipkow. Abstract interpretation of annotated commands. In *ITP*, volume 7406 of *LNCS*, pages 116–132. Springer, 2012.

[34] D. Pichardie. *Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés*. PhD thesis, U. Rennes 1, 2005.

[35] D. Pichardie. Building certified static analysers by modular construction of well-founded lattices. *Electr. Notes Theor. Comput. Sci.*, 212:225–239, 2008.

[36] T. W. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *PEPM*, pages 100–111. ACM, 2006.

[37] X. Rival and L. Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007.