Pragmatics of Formally Verified Yet Efficient Static Analysis, in particular for Formally Verified Compilers

David Monniaux

Pragmatics of Formally Verified Yet Efficient Static Analysis, in particular for Formally Verified Compilers

David Monniaux

July 10, 2024

Abstract

Formally verified compilers and formally verified static analyzers are a solution to the problem that certain industries face when they have to demonstrate to authorities that the object code they run truly corresponds to its source code and that it satisfies certain properties.

From a scientific and technological point of view, they are a challenge: not only a number of nontrivial invariants and algorithms must be proved to be correct, but also the implementation must be reasonably effective so that the tools operate within reasonable time. Many optimizations in compilers rely on static analysis, and thus a formally verified compiler entails formally verified static analyses.

In this article, we explain some difficulties, possible solutions, design choices and trade-offs pertaining to verified static analysis, in particular when the solution of the analysis is expressed as some form of tree, map or set.

1 Introduction

Static Analysis consists in deriving information about software without actually running it, by analyzing its source or object code. In some cases, static analysis may consist in checking that the program satisfies some stylistic constraints (e.g., not reusing the same name for a global and a local variables), or checking for patterns that often indicate mistakes (e.g., a memory block is freed along the normal exit of a function but not along a side exit). In this paper, we shall be solely concerned about static analysis that aims at proving that all possible executions of a program satisfy certain properties, most often through some form of *abstract interpretation* [14].

Such static analysis may be used for three main purposes: (i) ascertaining which areas should be "manually" examined by engineers (e.g., if automated analysis can prove that runtime errors are absent from most of the program, the engineers can focus on the remainder); (ii) proving that software behaves correctly, for instance as an argument for authorities in case the software must be qualified for safety-critical applications; (iii) proving that certain conditions for optimizations during compilation are met (e.g., an operand is always nonnegative, so signed extension and unsigned extension coincide

on this operand). The degree of tolerable uncertainty about the analysis results varies depending on the use. Obviously, it can be a serious issue if static analysis derives incorrect results that result in miscompilation, that is, of the production of object code not matching the semantics of the source code.

Formally Verified Compilation Mainstream compilers have bugs [48, 46]. In most industries, bugs caused by compilers are not a major concern compared to the amount of bugs already present in the source code of the application. In certain safety-critical industries, it is required that the designers of an embedded computing system show that the object code it uses matches the high-level specification, and in particular that the object code matches the source code. Common approaches to that problem often involved running a well-known compiler with most, if not all, optimizations turned off, so that object code follows source code closely, and some human examinations [20]. Such a solution is costly both in terms of cost and code efficiency.

Two alternatives approaches have been proposed. One is whole-compilation *translation validation*: the program is compiled with a normal compiler, then a procedure tries to match the source and object codes, perhaps using debugging information, and to prove their correspondence. This however tends to impose some constraints on the compiler and compilation options used, the form of the source code, etc., for the matching heuristics to succeed. To our knowledge, the only large-scale example of this approach is seL4. [43] The other approach is *formally verified compilation*. Various tools, based on various formalisms (higher-order logic, Floyd-Hoare style proofs,...) can be used to prove that a program behaves according to a specification. In particular, it is possible to prove that a compiler behaves according to its specification, namely that it compiles programs correctly. CompCert [33, 32] is such a formally-verified C compiler, used in some safety-critical industries. [20] CakeML[1] is a formally-verified ML compiler.

Compilers need static analysis at certain steps. For instance, they may perform a *points-to analysis* to see if pointers may be aliased—knowing that some pointers may not alias, that is, may not point to the same memory locations, allows certain optimizations, such as swapping a load and a store operations. A formally verified compiler will thus need some form of formally verified static analysis.

Formal proofs come at a significant cost for developers: in development, a correctness proof for a procedure may be significantly larger than the procedure itself and require much more expert work; in maintenance, it must generally be updated whenever the procedure is updated. It is therefore often desirable to minimize the number of properties to prove. One way to achieve this is by splitting the analysis algorithm into an oracle needing no proof, and a formally verified checking procedure (*formally verified defensive programming* [8]).

Another issue is efficiency and access to low-level constructs. For instance, *hash-consing* is a well-known approach for speeding up certain symbolic computations, but it requires a global hash table (and possibly auxiliary tables for *memoization* of operations). This global hash table is part of the global state, and thus cannot be easily

[1] `https://cakeml.org/`

modeled inside a pure functional language such as Gallina (the language of the Coq proof assistant).

2 Software Discussed in the Article

CompCert In this paper, we shall discuss some pragmatic choices that have been made in designing the static analyses of the "official" releases of CompCert[2] as well as the "Chamois" branch.[3] We however expect our insights to be valid for any kind of formally verified static analysis.

CompCert is a formally verified compiler for a large subset of the C programming language. It is organized into a few unverified frontend steps taking C as input, a formally verified core, then a few unverified steps that produce assembly code. [38] The formally verified core is organized in a succession of passes operating on intermediate languages. Each intermediate language is equipped with a formal operational semantics. Instructions operate over program states, and may optionally emit externally observable events (external function calls, accesses to special CPU registers, accesses to volatile variables…).

Many optimization passes operate over the RTL intermediate representation, which models execution state as a control location inside a current function, an abstract call stack, a memory consisting in memory blocks, and local "pseudo registers". A later phase allocates these pseudo registers into stack frames and CPU registers.

The overall correctness theorem of the compiler is that, if compilation succeeds, then the sequence of externally visible events defined in the C source semantics is matched by the sequence of externally visible events defined in the assembly code semantics. (If undefined behavior occurs, then no guarantee is provided.) This correctness is proved by the composition of simulation proofs for each pass. Except for some source semantics, all semantics in CompCert are deterministic and the associated simulation proofs are forward simulations.[4]

The overall correctness proof discusses whole traces of execution. However most proofs arguments are local and deal with the replacement of some number of source steps by some number of target steps. In the simplest case, simulation is lock-step: one step of the program prior to the transformation is matched by one step of the program after the transformation. A lock-step forward simulation argument is of the form: "if a source state s_1 can take a step to a source state s_2, emitting events e (possibly none), and s'_1 is a target state that simulates s_1, then there must exist a target state s'_2 that simulates s_2 and so that s'_1 can take a step to s'_2 emitting the same events e":

$$\forall s_1 s_2 s'_1,\ s_1 \rightarrow_e s_2 \wedge s_1 \sim s'_1 \implies \exists s'_2,\ s'_1 \rightarrow_e s'_2 \wedge s_2 \sim s'_2$$

[2] `https://github.com/AbsInt/CompCert`

[3] `https://gricad-gitlab.univ-grenoble-alpes.fr/certicompil/Chamois-CompCert`

[4] The semantics of C is nondeterministic: for instance, the compiler is in general free to choose the evaluation ordering of the arguments to operators and function calls [42, §6.5.3]. The simulation proof should thus be backward: any execution of the compiled code should match one of the executions allowed by the source semantics. In addition, the compiler is allowed to assign arbitrary target executions to source executions with undefined behavior; this leads to a rather complex simulation property [33, §2.1]. If, however, the source and target languages are deterministic, this backward property is equivalent to a forward property: if the source program S has a defined behavior B, then the target program must also exhibit behavior B.

Many transformation or optimization passes rely on the results of a static analysis. For instance, constant propagation on RTL relies on a *value analysis* that establishes that certain pseudo-registers and certain memory locations contain certain values. The simulation argument for the pass then relies on the invariants produced by this analysis being inductive (correct at function entry and correct at the next step if correct at the current step). The state simulation relation $\sim$ refers to these invariants.

Because it is a compiler, CompCert should be reasonably fast, and this is a challenge: (i) the Coq code handles integers as linked lists of bits, as opposed to machine words, leading to inefficient arithmetic (ii) clearly separated transformation passes may be less efficient than passes than perform several operations at once (iii) analyses are performed when needed with no provision for preserving their results across passes.

The *Chamois* branch features additional optimizations and experiments. Since it adds many additional passes, it is even more sensitive to inefficiencies. The *Verasco* static analyzer (see Section 7.1) was implemented on top of CompCert's front-end.

Astrée The Astrée static analyzer [7, 6] verifies that C programs do not reach undefined behaviors, including assertion violations, by automatically deriving inductive invariants. It originally targeted safety-critical code for avionic control applications. It is not formally verified, but some of its design choices inspired formally verified tools (Verasco…) and some of the efficiency challenges it faced are found in other tools.

Astrée performs whole program analysis, following structured control flow (with some extra constructs for dealing with `goto`). It abstracts numerical variables using intervals, octagons and specific abstract domains for control applications (numerical filters).

Because it performs whole program analysis on control programs that typically contain a number of remanent variables[5] linear in the size of the program, the performance of the data structures used to map program variables into memory cell indices and memory cell indices into abstract values was very important (§3).

Even though Astrée is not expected to perform as fast as a compiler, industries typically expect it to run (say, during the night) throughout their development process, as some form of continuous integration process, to catch possible problems early.

3 Tree, Maps and Sets in Static Analyses

When implementing static analyses, it is often necessary to use data structures representing maps or sets. We shall briefly see here some of the efficiency challenges they pose, before seeing, in later sections, some of the solutions we brought forth.

Non-Relational Analyses Static analysis typically maps every control location, say in a procedure, to some information. We consider here the case where this information pertains to the reachable program states at that location. Such an analysis is deemed

[5] By remanent variables we mean all those that are created at program startup and have indefinite lifetime: global variables, file-local and function-local `static` variables. A function-local `static` variable has local scope, but its value is retained from one call to the function to the next, as opposed to an `auto` (default case) local variable, which is created when coming into scope and destroyed when coming out of scope.

non-relational if the information is independent across variables; in contrast, a relational analysis will attempt tracking some forms of relationships between variables. A classical example of non-relational analysis is *interval analysis*, which tracks one interval per variable.

Consider for instance the following program: $y := x; z = x - y$. We perform interval analysis: to each variable at every location is associated an abstract value that is an interval. Assume the precondition $x \in [0, 1]$, then the analysis will derive $y \in [0, 1]$, and then $z \in [-1, 1]$, which is correct but a strict over-approximation of the exact postcondition $z = 0$. This postcondition, however, may be reached only by knowing the relationship $x = y$, not just by propagating per-variable information.

Despite that kind of weaknesses, non-relational analyses are extensively used, including inside compilers, because they are quite cheap. "Official" releases of CompCert, for instance, have a *value analysis*[6] that tracks if a variable is known to be actually a constant, or, if a pointer, whether it points to certain zones. This analysis can, for instance, track that pointers derived from arguments to a function point outside of the stack frame of that function, and thus cannot alias with pointers that are known to point inside the stack frame. In addition to this value analysis, Chamois has an interval analysis for integer variables [7], which is used for showing that certain variables are nonnegative (for replacing operations by simpler ones if they operate only on nonnegative numbers) and that certain computations do not overflow 32-bit values and thus can be promoted to 64-bit without changes in semantics.

A common way to implement non-relational analyses is to compute, for every control location, a data structure implementing a map from variable identifiers to abstract values. Obviously, such data structure should have fast access both for reading and writing values: when an instruction $r := f(a, b, c, d)$ is analyzed, the analyzer must fetch the abstract values for a, b, c and d from the structure, apply the abstract operation corresponding to f, then write the result to the structure. If the data structure is to be stored for every location, then this write operation should retain the old structure in addition to the new one. In addition, we should avoid needless data duplication, thus old and new structures should share as much as possible.

Obviously, this structure may be implemented as a functional map, through balanced binary trees (as in the Astrée static analyzer [7, §6.2]), Patricia trees or similar.[8] In fact, CompCert has successively had two libraries implementing prefix trees mapping positive integers to values through decomposition from low-order to high-order bits. The second one [2] has the nice property that two extensionally equal maps ($m(k) = m'(k)$ for all k) must be actually identical data structures: the maps are canonical, whereas, in the first version, it was possible to have two different data structures representing the same map. In Coq, it is often easier to work with canonical representations, since this avoids reasoning with respect to an equivalence relation such as extensional equality or semantic equivalence: doing so entails at least tedious proofs that operations behave the same modulo that equivalence, and sometimes one ends up with impossible obstacles, especially if using dependent types.

[6] See `ValueDomain.v` and `ValueAnalysis.v`

[7] See `ZIntervalDomain.v`, `ZIntervalAnalysis.v` and `BTL_ZIntervalAnalysis.v`

[8] A Patricia tree is a radix tree with radix equal 2.

Unfortunately, such a data structure is inefficient if it must be considered globally, which happens in two cases in static analysis:

- When two flows of control join at a certain point, such as the end of an if-then-else construct, with maps m and m' then one must construct a map $m \sqcup_{maps} m'$ such that $(m \sqcup m')(k) = m(k) \sqcup m'(k)$ where $\sqcup$ is the least upper bound operator for the abstract values. This entails going through all keys k.
- When one checks the invariant for inductiveness, one checks that $m(k) \sqsubseteq m'(k)$ for all k.

It was soon recognized when designing the Astrée system [7, §6.2] that if the analysis tracks all variables in the program, including global variables, these global join operations may come to have intolerable cost. These operations have cost linear in the number $|V|$ of variables, but in a program, especially the kind of safety-critical control programs that Astrée or verified compilation targets, the number of global variables is linear in the size $|P|$ of the program. The number of if-then-else operations is also linear in the size $|P|$ of the program. This means that, even for a loop-free program, the total cost of just the $\sqcup_{maps}$ operations will grow quadratic in the size of the program, which quickly becomes intolerable.

For CompCert's value analysis, two workarounds are used. Firstly, the analysis is local to each function, as opposed to Astrée. Global variables are assumed to contain arbitrary values at function entry, except for read-only variables, which are assumed to contain their initialization value (the map for read-only globals is computed once and for all and thus there are no costs associated to joins). The analysis tracks changes to global memory inside the function, but the data structure used just has to track a limited number of updates as opposed to tracking the entire memory state. Secondly, for local variables that are not allocated a memory location (pseudo-registers), only live variables are tracked. [9]

Another possible approach would be to use a sparse analysis based on single static assignment (SSA) form. [21]

Data-flow facts Many data-flow analyses attach to each control location a set s of dataflow "facts". When several control flows join at a location, the set of facts known to be alway true at that location is the intersection of the set of facts known to be always true at the incoming edges. Therefore, $s \sqcup s'$, the semantic "least upper bound operation", is actually $s \cap s'$.[10]

The question then becomes how to implement these sets. The same kinds of tree-like structures used for maps can be used, with values being either $0/1$ or simply always 1 (because the absence of a key/value association in a map may be interpreted as 0). Again, the efficiency problems lie in the global operations: set union, set intersection, and inclusion testing.

[9] See `ValueAnalysis.v`: information about a variable is cleared when analyzing the instruction at its last use, "last" being taken in a total ordering of program locations. This approach is easy to prove sound, since it is always sound to forget information about a variable.

[10] This explains why the conventions for lattices in abstract interpretation and dataflow analyses are often opposite, with the "top" element in abstract interpretation, meaning "I know nothing", being implemented by $\emptyset$, the bottom elements of sets.

Symbolic execution One way to validate the results of an optimization phase is to check that the original and the transformed programs are equivalent through symbolic execution. In the absence of branching control-flow constructs, symbolic execution means executing the program over symbolic inputs, computing intermediate values as terms over these symbolic inputs and the possible arithmetic operations. If these terms are equal in the outcomes of two programs, then these programs are equivalent. For instance, $x := 3; y := 3; z := x + 1$ and $x := 3; y := x; z := y + 1$ are equivalent because both produce $x : 3$, $y : 3$, $z : 3 + 1$. Thus checking equivalence of two programs boils down to checking equivalence of terms represented as trees, that is, again, checking that two trees are equal.

Such a system ignores the semantic meaning of operators ($3 + 1$, $1 + 3$ and 4 wil be considered different terms), but it is possible to enrich it by rewriting rules implementing such transformations; again checking equivalence boils down to checking equalities of terms in normal forms with respect to rewriting. Equivalences of programs containing branching controls, or even loops (through auxiliary invariants), may be checked likewise. The equivalence requirement may be relaxed to only apply to live variables.

As we shall see in Section 6, this symbolic execution approach has been extensively used inside Chamois for implementing optimizations such as basic block or superblock scheduling, loop-invariant code motion and strength reduction of index multiplication in loops. Efficiency here lies in being able to construct terms, apply rewriting at the root, and check for term equality very efficiently.

4 Solutions for Efficiency

If we were implementing a regular compiler (or some other category of symbolic tool, such as a computer algebra system, proof assistant…), a number of implementation "tricks" would be available to us to ensure efficiency. It is however not so easy to use these "tricks" in code formalized within a proof assistant, in particular if the proof assistant, such as Coq, views programs as purely functional.

4.1 Physical Pointer Equality

The efficiency problem that we pointed out in Section 3 can be stated as: when we apply a global operation (inclusion testing, least upper bound…) on two maps m and m', the cost of that operation is proportional to the number of variables $|V|$ in the maps even if the maps are very similar. For instance, at the end of an if-then-else construct

```
if (y < 0) {
  x = 3;
} else {
  x = 5;
}
```

the interval map will be the same from both branches except for the variables `x` and `y`, but the least upper bound operation for abstract values will be applied to all variables, even those that have not been touched in either branch.

```
# type t = A of int;;
# let x = A 0 and y = A 0;;
# x = y;;
- : bool = true
# x == x;;
- : bool = true
# x == y;;
- : bool = false
```

Figure 1: Pointer equality can distinguish between two equal values

```
Axiom tree_phys_eq: tree → tree → ?? bool
Axiom tree_phys_eq_correct: ∀ t1 t2,
  tree_phys_eq t1 t2 ⇝ true → t1 = t2
```

Figure 2: Using Boulmé's monad system [8, 19], a pointer equality ("physical equality") operator is declared over a tree datatype, inside a "may return" monad with Boolean return type (`?? bool`). The axiom states that if this operator has returned true, then the two values are semantically equal.

The solution used in Astrée for least upper bound operations was, when traversing the tree data structures implementing maps from variables to abstract values, to opportunistically detect cases when the subtree that would be produced would be identical to one of the input subtrees, and in this case to return that input subtrees through the same pointer. [7, §6.2] For instance, if computing the least upper bound $\sqcup_{maps}$ of two subtrees given by identical pointers, the procedure would immediately return the same pointer. The procedure for testing inclusion of two subtrees would first check if the two subtrees were given by identical pointers and return true immediately in that case. That opportunistic use of identical pointers was key to the efficiency of the analysis.

Can this approach be adapted to a formally verified context? It is tempting to add a predicate ==, meaning "physically equal pointers", and an axiom $\forall x \forall y\ x == y \implies x = y$. Unfortunately, this leads to paradoxes, because $x = y$ means that x can be substituted by y in any context and still yield identical results ("Leibniz equality"). Consider the program in Figure 1. The expressions `x == x` and `x == y` should yield identical results because x = y, but they do not. In short, the problem is that == allows distinguishing between semantically identical values (here `x` and `y`), whereas in logic no relation can be finer than equality.

One possible solution is to model == as a nondeterministic operation within a nondeterministic "may return" monad [8, 19]. This monad encapsulates possibly nondeterministic computations: $c \rightsquigarrow v$ means that the computation c may evaluate to v; the difference with an ordinary expression is that it is impossible to derive $v = v'$ from $c \rightsquigarrow v$ and $c \rightsquigarrow v'$. With some syntactic sugar, it allows writing Coq programs that use nondeterministic expressions; instead $c : v$ denoting a deterministic computa-

tion c of type v, we have c :??v denoting a nondeterministic, possibly nonterminating, computation evaluating to a value of type v. Here, we consider that physical equality is non-deterministic because it may return different Booleans (Fig. 1) when called twice with parameters that are semantically equal. We also add an axiom stating that if $x == y \rightsquigarrow \mathsf{true}$, then $x = y$ (Coq declarations in Fig. 2): if two pointers are equal then the objects they point to are equal. Then, the same opportunistic approach as in Astrée could be implemented.

In the code presented in Figure 2, the second `Axiom` is one in the logical sense (it states a logical property that will be assumed from then on), while the first just declares a function `tree_phys_eq` taking two trees as argument and returning a Boolean in the "may return" monad.

If evaluating terms inside Coq itself, `phys_eq` will not be available (evaluation stops on axioms, which are considered *uninterpreted functions*). However, CompCert is not directly executed within Coq, but rather the Coq code is extracted to OCaml, which is then compiled and linked together with manually written OCaml code to form the final executable. It is in particular possible at that point to state that certain Coq axioms, declaring types and functions, are realized by certain OCaml constructs.[11] When extracting Coq to OCaml, we set up the extraction mechanism so that `?? bool` gets extracted to `bool`[12], that is, the monad is elided, and `phys_eq` gets extracted to pointer equality `(==)`.

Shortcut Test Another possible formalization, introduced by Jourdan [29], is to declare physical equality as a special "shortcut" test that computes a result through a fast path when two terms are known to be equal (through physical equality), under the condition that this fast path returns the same result as the slow path. The Coq formalization is:

```
Axiom phys_eq : ∀ {A B : Type} (x y : A)
  (fast_path slow_path : unit → B)
  (Hsame : x=y → fast_path tt = slow_path tt), B
```

This axiom cannot introduce logical inconsistency: it can be implemented naively by just calling the slow path

```
Definition phys_eq_impl {A B: Type} (x y : A)
  (fast_path slow_path: unit → B)
  (Hsame: x=y → fast_path tt=slow_path tt):= slow_path tt
```

The following definition extends this scheme to cases when the fast path is correct only if $x = y$:

```
Axiom phys_eq : ∀ {A B : Type} (x y : A)
  (fast_path : x = y → B)
  (slow_path : unit → B)
  (Hsame: ∀ (eq: x=y), fast_path eq = slow_path tt), B
```

[11] See e.g. `extraction.v`/`extraction.vexpand`, and `ImpPrelude.v` in Chamois

[12] We also set up the extraction mechanism so that, instead of declaring a new OCaml type translating Coq's `bool` type, we reuse OCaml's standard Boolean type. This is standard.

The advantage of this approach is that it does not require reasoning within a monad. However, it imposes some form of local confluence of the computations between the slow and fast paths.

4.2 Hash Consing

The opportunistic approach consists in recognizing locally that some value that we are about to construct is equal to a value that we already have (perhaps a parameter to the function), and return that value instead of constructing another occurence of it, so that it the result can be recognized to be identical to that value by pointer equality.

A more general approach is *hash consing* [17]: all values created so far for a particular datatype are stored in a *hash* table, and, when a value is about to be *cons*tructed,[13] the table is checked for an existing copy of that value, which is returned instead if it exists. This ensures that no two copies of the same value can coexist (at different memory locations) in the system, and thus pointer equality is equivalent to value equality.

Such a hash table would continue growing and storing useless values. One possible workaround is to make the hash table local to a phase of the computations and discard it at the end of the phase. Another is to replace the hash table by a *weak hash table*, [14] so that values considered unreachable by the garbage collector of the execution platform are removed from the hash table (a value being reachable from the weak hash table does not make it considered as reachable by the garbage collector, as opposed to a normal hash table).

Because of the importance of hash-consing for implementing certain forms of symbolic computations, there has been some interest in how to use it from formally verified software written in a purely functional language, in particular with the Coq proof assistant [11]. We shall now discuss various difficulties, workarounds, and trade-offs involved in this.

As an untrusted oracle One possible implementation of hash consing is to use the hash table as an untrusted oracle. When we are about to construct a term $t = C(x, y, z)$ where C is a term constructor and x, y, z are subterms, we query the hash table for a copy t' of that term. We do not trust this t': we check that it is of the form $C(x', y', z')$, and then retain it if $x == x'$, $y == y'$ and $z == z'$ all $\rightsquigarrow$ true, which should always be the case if the hash table works properly.

A weakness of that system is that, even though it will never create two identical copies of the same term (provided all term constructions go through the process described above), and thus pointer equality is equivalent to equality for practical purposes, pointer equality is not *provably* equivalent to equality. In other words, we cannot conclude from the fact that $x == y \rightsquigarrow$ false that $x \neq y$, even though this works in reality. Furthermore, this system runs the whole computation inside a "may return" monad, which complicates proofs and forces the whole of the program to be executed inside that monad.

[13] While the name *hash consing* is associated with Lisp terminology from the 1970s, the idea of hash tables and hash-consing appeared as early as 1958 in the Soviet Union, in the context of compilation. [16]

[14] In particular, functor `Weak.Make` in OCaml. Weak hash tables are however available in other languages, e.g. `WeakHashMap` in Java.

```
Inductive tree :=
| Node : tree → tree → tree
| Leaf : nat → tree

Fixpoint tree_eqb (t t' : tree) :=
  match t, t' with
  | (Leaf n), (Leaf n') ⇒ Nat.eqb n n'
  | (Node l r), (Node l' r') ⇒
      (tree_eqb l l') && (tree_eqb r r')
  | _, _ ⇒ false
  end

Fixpoint tree_eqb_fast (t t' : tree) : ??bool :=
  DO cmp ↭ tree_phys_eq t t';;
  if cmp then RET true else
  match t, t' with
  | (Leaf n), (Leaf n') ⇒ RET (Nat.eqb n n')
  | (Node l r), (Node l' r') ⇒
      DO cmp_l ↭ tree_eqb_fast l l';;
      if cmp_l then tree_eqb_fast r r'
      else RET false
  | _, _ ⇒ RET false
  end

Lemma tree_eqb_fast_correct: ∀ t1 t2,
    WHEN tree_eqb_fast t1 t2 ⇝ b THEN b = tree_eqb t1 t2
```

Figure 3: A simple tree datatype with a naive equality test and a fast "shortcut" equality test, using the pointer equality defined in Fig. 2. It is possible to prove that the two coincide (below) and that they implement equality testing.

Because of the inconvenience of rewriting the whole of CompCert inside a monad, certain optimizations in Chamois, which are validated using a symbolic execution engine based on hash-consing [45], use an "unsafe exit" from the "may return" monad they were using. This "unsafe exit" turns a nondeterministic reduction $e \rightsquigarrow v$ into an ordinary value v. The reason why this is unsafe is that if we apply it to e such that e may return different values $v \neq v'$ ($e \rightsquigarrow v$ and $e \rightsquigarrow v'$), then considering this return value as deterministic leads to $v = v'$ and then an absurd case. The designers of Chamois however considered that this was not an issue, since this would somehow involve a case where the same optimization phase would be deliberately run twice on the same input, then the (nondeterministic) outputs compared and an absurd case entered if they differ.

Efficiency Tradeoff Let us compare now the naive and the shortcut equality tests in a context where trees are always produced by hash-consing, and thus there exist no two identical subtrees at distinct addresses. When given two identical trees, the naive test will traverse them fully, with complexity linear in the size of the tree. The shortcut test will terminate immediately with a positive answer. When given two different trees t and t', the naive test will still need to fully traverse identical subtrees, until it finds a path from the root that leads to different items in t and t'. This path forms a *witness* that the two trees are different.[15] The shortcut test will avoid these traversals and instead converge directly on such a witness. Its complexity is thus bounded by the minimum of the depths of t and t'.

Arguably, if trees are hash-consed, it should not be necessary to find a witness path for the difference of two trees, because the pointer equality test at the root gives the answer. However, in order to conclude that if the pointer equality test yields false, then the trees are different, we must use the invariant that all trees are created by hash-consing, a very strong assumption that cannot be directly expressed within the system. If we do not have this assumption and just the assumption that pointer equality (through a "may return" monad) implies tree equality, we end up with a less efficient equality test in case the trees differ.

Inequality as error In Chamois, phases validated by symbolic execution check equalities of terms by pointer equality, and the program exits immediately if it returns false. There is therefore no need to generate a witness that two terms are not equal.

As a trusted oracle In the "untrusted oracle" hash consing system, nothing prevents the application code from creating trees not going through the hash consing system. If we want the property that all trees go through hash-consing, then an approach is to ask Coq to extract the datatype to a specific OCaml type, with user-specified OCaml constructor and "match" operations. Then, whenever Coq code creates a tree in that datatype, it will call the "smart constructor", which will perform hash consing. [11, §6.1] Such an approach also makes it easy to hide fields, such as unique identifiers or hash values, that are not relevant at the Coq level.

This amounts to trusting the workings of the hash table and the hashing mechanism (in particular, that we will always be able to find extant elements in the table). If the hash table is weak, this means we trust its non-trivial interaction with the garbage collector of the execution platform. This extends the *trusted computing base* of the static analyzer or compiler. This is the choice that was made for the "hashed sets" library for sets of positive numbers, used to represent sets of dataflow facts in the CSE3 global common subexpression elimination phase of Chamois [40].

The argument here is that CompCert's trusted computing base [38] already includes Coq itself, which uses OCaml hash tables internally, so it does not seem that trusting the same hash table in extracted code adds much to it.

[15] By *witness* we mean a piece of data that is sufficient to establish the property. Here, to establish that t and t' are different, it is sufficient to exhibit a path that leads to different nodes in the two trees.

Using a Hash-Consed Backend Language An alternative to using a custom constructor would be to use a special backend language with automatic hash-consing of datatypes, an approached pioneered by HLISP [22]. The GimML language,[16] from the ML family [25, 23, 24], automatically performs hash-consing on datatypes on which it is safe to do so, which is for instance used to implement efficient finite sets and maps.

5 Invariant inference

Static analysis of programs containing loops must often compute inductive invariants. These invariants are often obtained by an iterative fixed-point (or post-fixed-point) computation.

5.1 Fixed-Point Computation

Iterative Computation Invariants are obtained as fixed points of certain operators. Consider transition systems where transitions are of the form $(p, \sigma) \to (p', \sigma')$, where $p, p' \in P$ are control locations and $\sigma, \sigma' \in \Sigma$ are data states. An *invariant* over that transition system is thus a mapping from P to the powerset of Σ, associating to each control location a set of states that must contain the states reachable at this location.

A state (p, σ) is reachable if and only if it is either an initial state (typically, there is a p_0 initial location with an associated set Σ_0 of initial data states), or if there exists a reachable state (p_{pre}, σ_{pre}) such that $(p_{pre}, \sigma_{pre}) \to (p, \sigma)$. An *inductive invariant* I is thus a mapping from P to the powerset of Σ such that (i) it contains initial states ($\Sigma_0 \subseteq I(p_0)$) (ii) it is inductive: for all $(p, \sigma) \to (p', \sigma')$ with $\sigma \in I(p)$, then $\sigma' \in I(p')$. An inductive invariant is an invariant, but an invariant may be noninductive.

Assume we have an abstract domain $\Sigma^\sharp$ for representing subsets of Σ. Assume also that we have a function S such that, for $p \in P$ and $\sigma^\sharp \in \Sigma^\sharp$, $(p, \sigma^\sharp)$ is a finite set of pairs $(p', \sigma^{\sharp\prime})$ such that if $(p, \sigma) \to (p', \sigma')$, $\sigma \in \gamma(I^\sharp(p))$, then there exist $(p', \sigma^{\sharp\prime}) \in S(p, I^\sharp(p))$ such that $\sigma' \in \gamma(\sigma^{\sharp\prime})$. In other words, this function S associates to each control location a finite set of (*successor*, *abstract state*) pairs; most instructions have only one successor, branching instructions have several. In the simplest cases, the successor abstract state will be the same regardless of the successor, but it may be useful to have differing abstract states, for instance to reflect the information brought in by the condition on the branching instruction (for instance, after a condition $i = 42$ we know that $i = 42$). An inductive invariant $I^\sharp$ for the transition system in the abstract domain is a mapping from P to $\Sigma^\sharp$ such that (i) it contains an abstract version of the initial states ($\sigma_0^\sharp \sqsubseteq I^\sharp(p_0)$) (ii) it is inductive: for all $(p', \sigma^{\sharp\prime}) \in S(p, I^\sharp(p))$, then $\sigma^{\sharp\prime} \sqsubseteq I^\sharp(p')$. These properties are decidable by a simple procedure, assuming $\sqsubseteq$ and S are computable.

The usual approach to solving such a problem is a workset-based algorithm, in which the workset contains a list of states whose successors may not contain yet the elements that are propagated. The initial workset W contains just p_0, and $I^\sharp(p_0)$ is initialized to $\sigma_0^\sharp$. As long as W is nonempty, a p is picked from it, and for every

[16] `https://projects.lsv.fr/agreg/?page_id=258` Formerly HimML.

$(p', \sigma^{\sharp\prime}) \in S(p, I^\sharp(p))$, the algorithm checks if $\sigma^{\sharp\prime} \sqsubseteq I^\sharp(p')$; if not, $I^\sharp(p')$ is replaced by its "least upper bound" with $\sigma^{\sharp\prime}$, and p' is added to W.

The "least upper bound" operator does not actually need to be the least upper bound, it just has to yield a result greater than its operands. In lattices with infinite ascending chains, one usually replaces it with a *widening* operator, which ensures converges in a finite number of iterations. The widening operator may be applied only at a selected subset of control locations sufficient to break all cycles in the control-flow graph.

If this algorithm terminates, and thus reaches a *fixed point*, then this fixed point is an inductive invariant. Under some monotonicity condition for S[17] and with the assumption that the least upper bound operator is really the least upper bound, it will compute the least inductive invariant in the abstract domain.

The order in which elements are picked from W is unimportant for correctness, but is important for efficiency. For instance, if a procedure consists in two successive loops (or, in terms of graphs: its control-flow graph consists of two strongly connected components), it is more efficient to first compute the fixed point of the first loop, then that of the second loop, rather than the two at the same time. This is achieved by sorting control locations in reverse postorder and picking the least element from W with respect to this ordering. W can be implemented as a heap.[18]

This algorithm, also known as Kildall's algorithm, is implemented in the official releases of CompCert, with the restriction that all edges outgoing from the same control location receive the same abstract state. Chamois also has another fixpoint algorithm lifting that restriction. In neither case, the convergence of the algorithm is proved: the algorithm, unless W becomes empty, iterates up to a very large "fuel" natural integer, and if it reaches it, gives up and returns an error. Proving convergence would entail arguing about the absence of infinite ascending chains in $X^\sharp$ and the finiteness of P.

It is not a problem in practice that the analysis can report an error; in this case one can either give up on the optimization that requested the analysis, or safely use $\top$ ("anything is possible") at all locations.

Fixed-Point Checking The above approach directly computes an inductive invariant in a formally verified manner. Another way is to compute the invariant using an oracle, and then check that it is inductive using a verified procedure.

The elements of the static analysis lattice are likely to be maps (in the case of a non-relational domain, mapping variables to abstract values) or sets (in the case of dataflow analysis). Checking that a fixed point is reached amounts to inclusion testing or even, depending on how the fixed point problem is formulated, to equality testing. For efficiency, one may want to apply the methods discussed in Section 4: inclusion tests apply "shortcuts" when identical subtrees (thus identical submaps or subsets) are detected, and hash-consing ensures that identical subtrees actually get identical pointers.

[17] Note that the value analysis in CompCert is not monotone and that the "least upper bound" operator is not the least upper bound. See issue 490.

[18] In CompCert, `Renumber.v` renumbers the control locations of a procedure so that the entrypoint is maximal, and one picks the maximal element in the workset.

5.2 Data-Flow Facts

In data-flow analysis, the lattice $\Sigma^\sharp$ of abstract elements is the powerset of a finite set F of elementary dataflow facts. One difficulty is that the set F may not be known in advance. In fact, it may be advantageous to dynamically enrich F during the fixed point computation. When an elementary fact is to be used, say $x = y + z$, it is looked up in a hash table that associates an integer to it; if it does not exist in the table, a fresh index is associated to it. We may also compute auxiliary tables, such as, for every variable v, the set of dataflow facts that are to be invalidated by a write to v (e.g. the fact $x = y + z$ is to be invalidated by writes to x, y or z).

At the end of the fixed point computation, we thus have, among other information (i) a mapping from P to subsets of the final F, represented as sets of integer indices (ii) a table mapping these indices to their semantics as elementary dataflow facts, which was being updated during the fixed point computation but which can be now taken as a read-only data structure. Note that we may have other tables, but since we have not proved that they are updated consistently, we cannot easily use them in subsequent verified computations. We thus rebuild auxiliary tables in a verified manner, if necessary.

We then check that the computed fixed point is truly inductive, in a verified manner. This is how the data-flow facts used for global common subexpression and condition elimination (CSE3 pass) are established in Chamois [40].

In Chamois, the integer indices associated to data-flow facts are positive, and the sets of positive integers are represented as binary trees indexed by the binary decomposition of the integers. These binary trees constitute a canonical representation: a set may be represented only by one tree, and thus semantics equality is equivalent to structural equality. These binary trees are built using hash-consing, and thus structural equality is equivalent to pointer equality. Many operations on the sets (equality testing, union, intersection, inclusion testing, ...) are sped up by checking for shortcuts when some subtrees are equal, whch boils down to pointer equality.

Chamois uses hash-consing as a trusted oracle when it comes to binary trees representing data-flow facts: the constructor and pattern-matching operations are replaced through the extraction mechanism by suitable OCaml code, and the equality test is mapped to pointer equality.[19] This choice may be considered excessive; it would have been sufficient to just use pointer equality to short-cut inclusion tests, just using as an axiom that pointer equality implies structural equality. However, this would have entailed programming inside a "may return" monad, as though the algorithms were nondeterministic, even in cases where it can be proved that the result is deterministic (inclusion testing has a uniquely defined Boolean return value). Stronger attention was awarded to the ease of expressing results simply than to maximal reduction of the trusted computing base.

6 Symbolic Execution

Another kind of static analysis used for proving the correctness of optimization phases in Chamois is *symbolic execution* [45, 27, 26]. The basic idea is that two sequences

[19] `HashedSet.v`

of instructions are equivalent if and only if they leave the same final results in the variables, regardless of the order in which they did the operations. An extension of this idea is to consider only those variables that are live at the end of the computation.[20] This equivalence can be established by computing, for every live variable at the end of the computation, a term expressing it as a function of the variables at the beginning of the computation. These terms are obtained by applying the operations in both programs symbolically. This is for instance used to show that the code after scheduling performs as the one before scheduling.

For instance, if x and v are not live at the end of the computation, these two programs are equivalent:

$u := x + y$	$u := x + y$
$z := x + y$	$t := x - y$
$t := x - y$	$x := 0$
$v := x - y$	$z := u$

and this can be established by computing symbolic forms: at the end of the computation, in both cases, $y = y_0$, $z = x_0 + y_0$, $t = x_0 - y_0$, $u = x_0 + y_0$ where v_0 denotes the initial value of variable v. Note that, as seen on the second program when performing $z := u$, doing this symbolic execution naively may duplicate expressions. Ideally, we would like $x_0 + y_0$ to be stored only once, and only a pointer to it be copied; thus terms should form a DAG (directed acyclic graph), not individual trees.

After symbolic execution has been done, we need to check that for every live variable, the final symbolic terms are identical. In practice, this will always be the case, unless there has been some bug in the optimization phase. We thus need to optimize the case where the terms are equal. If we check term equivalence naively, by traversal, the complexity of the check will be linear in the size of the terms as trees. That size can be exponential in the length of the programs to be analyzed, as in the following example: $a_1 := op(a_0, a_0); \ldots; a_n := op(a_{n-1}, a_{n-1})$ where a_n is a complete binary tree of depth n. In contrast, the size of a_n as a directed acyclic graph (DAG) is linear in n.

The solution for this is, again, hash-consing. The terms are hash-consed, and the optimization is accepted only if the final terms are equal in the sense of pointer equality. Note that the correctness of that approach only relies on pointer equality implying term equality. In particular, it does not require pointer disequality implying term disequality: should this condition not be met (perhaps due to a bug in hashing), the only risk would be that a correct optimization result would be refused, with the compiler skipping that optimization or terminating with an internal error. Such problems can be weeded out by careful testing [41].

The symbolic execution system in Chamois is implemented inside a "may return" monad. The hash tables and monad are discarded at the end of the optimization phase. Again, this is theoretically "unsafe", but the only way this could create an issue is if the same optimization was run twice and the results compared so as to lead to an absurd case in case they differed.

Symbolic execution and expressions being tested for purely syntactic equivalence

[20] In the case of programs with operations that may trap, such memory accesses (in case of invalid addresses) or division (if division by zero is trapping), there is also the requirement that a program may be transformed into another only if the set of expressions that may trap in the second program is included in that for the first.

have limitations. For instance, the trees representing the integer expressions $a + (b + c)$ and $(a + b) + c$ are different, but they are semantically equivalent. For certain optimizations, such as *strength reduction*, it is necessary to identify some syntactically different expressions; this can be achieved by applying suitable rewriting rules along with the symbolic computation, so as to compare canonical forms at the end. [26].

Furthermore, both loop-invariant code motion and strength reduction need invariants (e.g. "variable t in the transformed program stands for expression $p + 8 \times i$ in the original program"). These invariants are computed by untrusted oracles, and are checked for correctness by the symbolic execution engine using the rewriting rules. [26]

7 Related Work: Other Forms of Static Analysis

We have so far discussed formally verified static analysis from the point of view of analyses used for optimizing compilation. Let us briefly discuss some other forms of analyses that are commonly used for program verification.

7.1 Relational Abstract Domains: Convex Polyhedra

The Verasco project[21] [29, 30] aimed at fitting CompCert with a formally verified static analyzer capable of automatically proving certain properties, such as the absence of certain runtime errors (buffer overflows, arithmetic overflows...).[22] It could use both non-relational (intervals) and relational (convex polyhedra) numeric analyses.

Verasco uses data structures with sharing much like Astrée, implemented through physical pointer equality [29, Ch. 9]. The convex polyhedra libary VPL[23] [10, 34, 35, 36, 9, 18, 19], despite being based on a constraint-only representation that supposedly scales better than the conventional double representation (generators and constraints) with respect to dimension, also had scaling issues, as expected from such a highly relational analysis. The relational analyses would likely have scaled better if applied to select subsets of the variables ("packs"), as done in Astrée. [37]

The design choices of the VPL reflect some of the concerns and insights described in this paper: inclusion or equality tests should be fast, and it is often easier to implement a verified operation as the composition of an unverified oracle and a verified checker.

The conventional approach to polyhedral computations is through "double representation", where a polyhedron is represented both as a system of *constraints* (faces) and a system of *generators* (vertices, and, in the case of unbounded polyhedra, rays and lines). The two representations are dual, and it is possible to move from one to the other using various algorithms. One representation may also be used to eliminate redundancies from the other. The problem with this approach, from the point of view of a verified analyzer, is that it is unsound to omit generators, and thus the conversion from

[21] `http://compcert.inria.fr/verasco/`

[22] In other words, that project aimed at implementing a formally verified, simpler analogue of tools such as Astrée [7, 6] or Frama-C value analysis.

[23] `https://github.com/VERIMAG-Polyhedra/VPL`

constraints to generators must be shown not to skip any. VPL eschewed this approach and instead represented polyhedra by constraints only.

Operations on constraints defining polyhedra may be justified by, in essence, showing that certain constraints are logical consequences of others. This entailment may be justified by showing that the consequence is a combination of the antecedent constraints with nonnegative coefficients, often known as *Farkas coefficients*.[24] For instance, if we want to show that the projection on x, parallel to y, of the polyhedron defined by $x + y \leq 1$ and $x - y \leq 2$ is included in the $x \leq 3/2$, then it is sufficient to recognize that by multiplying $x + y \leq 1$ by $1/2$ and $x - y \leq 2$ by $1/2$ and adding them, we obtain $x \leq 3/2$. A procedure computing the projection of a polyhedron can thus justify that its result includes the correct projection by providing, for each constraint it outputs, a Farkas certificate establishing that it is a consequence of the constraints in the original polyhedron. This generalizes to other operation than projections: Farkas certificates prove that the result polyhedron includes the polyhedron to be computed. This inclusion is sufficient for proving soundness [25]

In the original design of the VPL, certificates were computed explicitly and fed to the verifier, which entailed much bookkeeping. The system was later redesigned so that certificates are not explicitly computed, but rather appear as elements of a datatype that can only be manipulated through certain basic operations that are shown to preserve the property that they are valid certificates. [8] This simplifies the design (the number of lines of code was halved) and slightly improves performance.

Despite the difficulties involved in the double representation approach, some recent work proposes formally verified yet effective ways of computing the edge-vertex graphs [1]. Again, that approach uses certificates that certain properties are correct (these certificates are later checked for correctness by formally-verified code) as opposed to proving total correctness.

7.2 SAT / SMT Solvers

SAT and SMT solvers have been extensively used to check properties of hardware and software systems, and even to prove or disprove mathematical conjectures [28]. When a SAT solver answers positively and provides a purported model, it is easy to check if it is truly a model. When a SAT solver answers negatively, there is no such obvious witness. SAT solvers may however be instrumented to produce some kind of trace of their execution, which can then be verified independently, a feature now considered *de rigueur*.[26] The scientific challenge is to design a certificate format that is both compact

[24]Farkas' lemma states that a linear inequality is a consequence of other linear equalities if and only if it can be expressed as a nonnegative combination of these (the result extends to affine inequalities by allowing relaxation of the constant coefficient). The "if" part is trivial but the "only if" part is a bit more involved, it may be for instance established by instrumenting Fourier-Motzkin elimination. Farkas' lemma is closely related to the strong duality theorem in linear programming.

[25]These certificates are however insufficient for proving that the computed polyhedron is exactly the projected polyhedron sought, but that is not needed for soundness of the analysis.

[26]In fact, because of the general unreliability of negative results from SAT solvers unless such precaution is taken, certificates of unsatisfiability have been required for UNSAT tracks in the SAT competition since 2013.

(full trace logging is too expensive) and yet relatively easy to check (so that the verifier can be kept simple). Some tools may be used to reprocess and simplify the certificate.

It is possible to write a formally verified checker for SAT with checking time on the same order as the solving and proof processing times [8]. The SMTCoq tactic takes it further: it calls external SMT solvers and processes the certificates they produce to reach a goal in Coq [15, 31, 3]. Again, the same principle applies: an untrusted solver is coupled with a verified checker.

SAT/SMT solvers cannot normally be used to analyze programs containing loops, except by unfolding them to a finite depth. However, there exist some approaches (IC3/PDR in particular) that use SAT/SMT as subprocedures and are capable of proving safety properties. The reliability of these tools was once not too great [39, §6]. Producing efficient yet formally verified versions for them seems a challenge.

8 Future Work

The various approaches that we described to implement hash-consing had various drawbacks. A solution would be to use a state monad directly implemented in Coq, as opposed to using extraction tricks, and carry the state of the hash table inside that monad. This entails implementing the hash table in Coq. In older versions of Coq, the array in which the hash table is stored had to be itself implemented using some kind of functional map structure (ordered tree, binary tree…), which was inefficient. Thus, hash-consing implemented in this fashion was more of an academic exercise than for actually running it [11, §5], though some have successfully pushed to approach to a full ROBDD implementation complete with a stop-and-copy garbage collector. [12]

A more modern approach would use Coq's relatively recent support of native integers and native arrays [4][47, §2.1.13, "Primitive objects"]. The native integers would be used to implement the hash functions efficiently, and the native arrays would be used for the table itself. These arrays are *persistent*: the store operation is defined to return a new version of the array; old versions of the array can still be accessed, but the implementation is optimized for the case where only the most recently updated version of the array is accessed. Internally, only the last version of the persistent array is retained, inside a regular (mutable) array, and the previous versions are stored as explicit deltas from this last version. [13, §2.3] If the previous versions are no longer used, these deltas will eventually get garbage-collected. There exist extraction configurations so that, when extracting Coq to OCaml, these persistent arrays get extracted to an OCaml implementation along these lines. After extraction, we would thus obtain a hash table similar to one we could have implemented manually in OCaml.

It remains to be seen how exactly to use native integers and arrays to implement an efficient hash table for hash-consing, reflecting the necessary invariants (all extant objects have been allocated in the table). It seems a much more difficult task to make this table "weak", since this amounts to incorporating part of garbage collection (if only some system of reference counting) inside the formalization, whereas normally garbage collection is left to the runtime system. Perhaps, again, it is best to renounce collecting the terms that have been created and left unused during the use of the hash table, and instead wait until the state monad is exited and the hash table is discarded.

This fits the use of hash consing inside an optimization or code transformation phase, where everything needed for the internals of the phase can be discarded at the end of the application of that phase (perhaps even at the end of the application of that phase to a particular function).

Exiting the may-return monad abruptly at the end of an optimization phase is most likely not dangerous, but is inelegant. A better approach would be to allow exiting the may-return monad if it always returns the same value as some deterministic computation.

9 Conclusion

We have successfully implemented and formally verified a number of optimizations (prepass and postpass scheduling, loop unrollings and rotations, global common subexpression elimination, loop-invariant code motion, store motion, strength reduction...) that were not present in the official releases of CompCert [44, 40, 45, 26, 27]. Most of these transformations involve some form of static analysis to establish their correctness, whether this static analysis computes some invariants or establishes the equivalence of two blocks by symbolic execution.

While these optimizations may appear to be well-known, and are generally available in mainstream, unverified compilers such as GCC or LLVM, there was on every occasion significant work to be done for identifying the necessary invariants, the necessary properties to be proved, for distinguishing what actually needed to be formally verified from what was not, and for formalizing the optimization in a way that made proofs tractable. We echo here remarks often made about the formalization of semantics, algorithms, or mathematical proofs:[27] badly chosen formalism often allows small-scale works and it is only when attempting larger proofs or algorithms that one will encounter difficulties. We will even go as far as to say that formalizing known optimizations helps understand them better. [5]

There is often a choice to be made between proving *complete correctness* of the analysis (the analysis always succeeds and compute a correct result) and *partial correctness* (if the analysis succeeds, then it computes a correct result). The latter is often much easier to prove than the former: there is no need to prove termination (one may either have part of the procedure in untrusted code with no termination requirement, or use a high maximum number of iterations and fail if it is reached), and one can split the analysis into an untrusted oracle and a formally verified checker.

Even at the level of the individual test, we can examine closely what is actually needed or not. For instance, we may check whether two structures are equal ("are these two maps equals" when searching for an invariant, "are these two terms equal" in symbolic execution), but only the positive answer needs to be correct. If a negative answer is produced instead of a positive, the only consequences would be that needless iterations of a search would be run, or an optimization would be refused whereas it could have gone through.

[27] For instance in talks given by Georges Gonthier about his work on the four-color theorem and the Feit-Thomson theorem.

One key insight is that we can apply different standards of proof to different properties that we expect of an algorithm or analysis scheme. We may, for instance, formally verify soundness, but prove optimality only on paper, and establish performance by experimental measurements. [41]

Acknowledgments

We wish to thank Sylvain Boulmé for his insights and comments.

References

[1] Xavier Allamigeon, Quentin Canu, and Pierre-Yves Strub. "A Formal Disproof of Hirsch Conjecture". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*. Ed. by Robbert Krebbers et al. ACM, 2023, pp. 17–29. DOI: 10.1145/3573105.3575678.

[2] Andrew W. Appel and Xavier Leroy. "Efficient Extensional Binary Tries". In: *J. Autom. Reason.* 67.1 (2023), p. 8. DOI: 10.1007/s10817-022-09655-x.

[3] Michaël Armand et al. "A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses". In: *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 135–150. DOI: 10.1007/978-3-642-25379-9_12.

[4] Michaël Armand et al. "Extending Coq with Imperative Features and Its Application to SAT Verification". In: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 83–98. DOI: 10.1007/978-3-642-14052-5_8.

[5] Jeremy Avigad. "Mathematics and the formal turn". In: *Bulletin of the American Mathematical Society* 61.2 (Apr. 2024), 225—240. DOI: 10.1090/bull/1832.

[6] Bruno Blanchet et al. "A static analyzer for large safety-critical software". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*. Ed. by Ron Cytron and Rajiv Gupta. ACM, 2003, pp. 196–207. DOI: 10.1145/781131.781153.

[7] Bruno Blanchet et al. "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software". In: *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*. Ed. by Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough. Vol. 2566. Lecture Notes in Computer Science. Springer, 2002, pp. 85–108. DOI: 10.1007/3-540-36377-7_5.

[8] Sylvain Boulmé. “Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles). (Programmation défensive formellement vérifiée (calculs efficaces et vérifiés en Coq, à partir d’oracles OCaml potentiellement non fiables))”. Habilitation. Université Grenoble Alpes, 2021. HAL: tel-03356701.

[9] Sylvain Boulmé and Alexandre Maréchal. “Refinement to Certify Abstract Interpretations, Illustrated on Linearization for Polyhedra”. In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, 2015, pp. 100–116. DOI: 10.1007/978-3-319-22102-1_7.

[10] Sylvain Boulmé et al. “The Verified Polyhedron Library: an Overview”. In: *20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2018, Timisoara, Romania, September 20-23, 2018*. IEEE, 2018, pp. 9–17. DOI: 10.1109/SYNASC.2018.00014.

[11] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. “Implementing and Reasoning About Hash-consed Data Structures in Coq”. In: *J. Autom. Reason.* 53.3 (2014), pp. 271–304. DOI: 10.1007/s10817-014-9306-0.

[12] Clément Chavanon, Frédéric Besson, and Tristan Ninet. “PfComp: A Verified Compiler for Packet Filtering Leveraging Binary Decision Diagrams”. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*. Ed. by Amin Timany et al. ACM, 2024, pp. 89–102. DOI: 10.1145/3636501.3636954.

[13] Sylvain Conchon and Jean-Christophe Filliâtre. “A persistent union-find data structure”. In: *Proceedings of the ACM Workshop on ML, 2007, Freiburg, Germany, October 5, 2007*. Ed. by Claudio V. Russo and Derek Dreyer. ACM, 2007, pp. 37–46. DOI: 10.1145/1292535.1292541.

[14] Patrick Cousot and Radhia Cousot. “Abstract Interpretation Frameworks”. In: *J. Log. Comput.* 2.4 (1992), pp. 511–547. DOI: 10.1093/logcom/2.4.511.

[15] Burak Ekici et al. “SMTCoq: A Plug-In for Integrating SMT Solvers into Coq”. In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. Ed. by Rupak Majumdar and Viktor Kuncak. Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 126–133. DOI: 10.1007/978-3-319-63390-9_7. HAL: hal-01669345.

[16] A. P. Ershov. “On Programming of Arithmetic Operations”. In: *Commun. ACM* 1.8 (Aug. 1958). The original article in the Proceedings of the Academy of Sciences of the USSR is available from https://www.mathnet.ru/php/archive.phtml?wshow=paper&jrnid=dan&paperid=28010&option_lang=rus, 3–6. ISSN: 0001-0782. DOI: 10.1145/368892.368907. URL: https://doi.org/10.1145/368892.368907.

[17] Jean-Christophe Filliâtre and Sylvain Conchon. “Type-safe modular hash-consing”. In: *Proceedings of the ACM Workshop on ML, 2006, Portland, Oregon, USA, September 16, 2006*. Ed. by Andrew Kennedy and François Pottier. ACM, 2006, pp. 12–19. DOI: 10.1145/1159876.1159880.

[18] Alexis Fouilhé. “Revisiting the abstract domain of polyhedra : constraints-only representation and formal proof. (Le domaine abstrait des polyèdres revisité : représentation par contraintes et preuve formelle)”. PhD thesis. Grenoble Alpes University, France, 2015. HAL: tel-01286086.

[19] Alexis Fouilhé and Sylvain Boulmé. “A Certifying Frontend for (Sub)polyhedral Abstract Domains”. In: *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*. Ed. by Dimitra Giannakopoulou and Daniel Kroening. Vol. 8471. Lecture Notes in Computer Science. Springer, 2014, pp. 200–215. DOI: 10.1007/978-3-319-12154-3_13.

[20] Ricardo Bedin França et al. “Towards Formally Verified Optimizing Compilation in Flight Control Software”. In: *Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop PPES 2011, March 18, 2011, Grenoble, France*. Ed. by Philipp Lucas et al. Vol. 18. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011, pp. 59–68. DOI: 10.4230/OASIcs.PPES.2011.59.

[21] Laure Gonnord. “Contributions to program analysis: expressivity and scalability. (Contributions aux analyses de programmes, expressivité, passage à l’échelle)”. Habilitation. Université Claude Bernard Lyon 1, 2017. HAL: tel-01633065.

[22] Eiichi Goto. *Monocopy and Associative Algorithms in an Extended Lisp*. Tech. rep. TR 74-03. Information Science Laboratory, Faculty of Science, University of Tokyo, Apr. 1974. URL: https://www.cs.utexas.edu/users/hunt/research/hash-cons/hash-cons-papers/monocopy-goto.pdf.

[23] Jean Goubault. “HimML: Standard ML with Fast Sets and Maps”. In: *In 5th ACM SIGPLAN Workshop on ML and its Applications*. Also INRIA RR-2265. ACM Press, 1994.

[24] Jean Goubault. *Implementing Functional Languages with Fast Equality, Sets and Maps: an Exercise in Hash Consing*. Tech. rep. May 1994 version also available. Bull S.A. Corporate Research Center, 1992. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.1757&rep=rep1&type=pdf.

[25] Jean Goubault-Larrecq. *The GimML reference manual*. version 1.0. July 2021. URL: http://www.lsv.fr/~goubault/GimML/refman.pdf.

[26] Léo Gourdin et al. “Formally Verifying Optimizations with Block Simulations”. In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023). DOI: 10.1145/3622799. HAL: hal-04102940.

[27] Léo Gourdin. “formally verified postpass scheduling with peephole optimization for AArch64”. In: *AFADL*. 2021. URL: https://www.lirmm.fr/afadl2021/papers/afadl2021_paper_9.pdf.

[28] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. “Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer”. In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. Lecture Notes in Computer Science. Springer, 2016, pp. 228–245. DOI: 10.1007/978-3-319-40970-2_15.

[29] Jacques-Henri Jourdan. “Verasco: a Formally Verified C Static Analyzer. (Verasco: un analyseur statique pour C formellement vérifié)”. PhD thesis. Paris Diderot University, France, 2016. HAL: tel-01327023.

[30] Jacques-Henri Jourdan et al. “A Formally-Verified C Static Analyzer”. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. Ed. by Sriram K. Rajamani and David Walker. ACM, 2015, pp. 247–259. DOI: 10.1145/2676726.2676966.

[31] Chantal Keller. “A Matter of Trust: Skeptical Communication Between Coq and External Provers. (Question de confiance : communication sceptique entre Coq et des prouveurs externes)”. PhD thesis. École Polytechnique, Palaiseau, France, 2013. HAL: pastel-00838322.

[32] Xavier Leroy. “A Formally Verified Compiler Back-end”. In: *J. Autom. Reason.* 43.4 (2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4.

[33] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Commun. ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814.

[34] Alexandre Maréchal. “New Algorithmics for Polyhedral Calculus via Parametric Linear Programming. (Nouvelle Algorithmique pour le Calcul Polyédral via Programmation Linéaire Paramétrique)”. PhD thesis. Grenoble Alpes University, France, 2017. HAL: tel-01695086.

[35] Alexandre Maréchal, David Monniaux, and Michaël Périn. “Scalable Minimizing-Operators on Polyhedra via Parametric Linear Programming”. In: *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*. Ed. by Francesco Ranzato. Vol. 10422. Lecture Notes in Computer Science. Springer, 2017, pp. 212–231. DOI: 10.1007/978-3-319-66706-5_11.

[36] Alexandre Maréchal and Michaël Périn. “Efficient Elimination of Redundancies in Polyhedra by Raytracing”. In: *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*. Ed. by Ahmed Bouajjani and David Monniaux. Vol. 10145. Lecture Notes in Computer Science. Springer, 2017, pp. 367–385. DOI: 10.1007/978-3-319-52234-0_20.

[37] Antoine Miné. “The octagon abstract domain”. In: *High. Order Symb. Comput.* 19.1 (2006), pp. 31–100. DOI: 10.1007/s10990-006-8609-1.

[38] David Monniaux and Sylvain Boulmé. “The Trusted Computing Base of the CompCert Verified Compiler”. In: *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*. Ed. by Ilya Sergey. Vol. 13240. Lecture Notes in Computer Science. Springer, 2022, pp. 204–233. DOI: 10.1007/978-3-030-99336-8_8.

[39] David Monniaux and Laure Gonnord. “Cell Morphing: From Array Programs to Array-Free Horn Clauses”. In: *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. Ed. by Xavier Rival. Vol. 9837. Lecture Notes in Computer Science. Springer, 2016, pp. 361–382. DOI: 10.1007/978-3-662-53413-7_18.

[40] David Monniaux and Cyril Six. “Formally Verified Loop-Invariant Code Motion and Assorted Optimizations”. In: *ACM Trans. Embed. Comput. Syst.* 22.1 (2023), 3:1–3:27. DOI: 10.1145/3529507.

[41] David Monniaux et al. “Testing a Formally Verified Compiler”. In: *Tests and Proofs - 17th International Conference, TAP 2023, Leicester, UK, July 18-19, 2023, Proceedings*. Ed. by Virgile Prevosto and Cristina Seceleanu. Vol. 14066. Lecture Notes in Computer Science. Springer, 2023, pp. 40–48. DOI: 10.1007/978-3-031-38828-6_3. HAL: hal-04096390.

[42] *Programming languages—C*. International standard. ISO/IEC, 9899:1999.

[43] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. “Translation validation for a verified OS kernel”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 471–482. DOI: 10.1145/2491956.2462183.

[44] Cyril Six. “Compilation optimisante et formellement prouvée pour un processeur VLIW”. PhD thesis. Grenoble Alpes University, France, 2021. HAL: tel-03326923.

[45] Cyril Six, Sylvain Boulmé, and David Monniaux. “Certified and efficient instruction scheduling: application to interlocked VLIW processors”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 129:1–129:29. DOI: 10.1145/3428197.

[46] Chengnian Sun et al. “Toward Understanding Compiler Bugs in GCC and LLVM”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: Association for Computing Machinery, 2016, 294–305. ISBN: 9781450343909. DOI: 10.1145/2931037.2931074.

[47] *The Coq Reference Manual*. 8.17.1. The Coq Development Team. June 2023. URL: https://github.com/coq/coq/releases/tag/V8.17.1.

[48] Xuejun Yang et al. “Finding and understanding bugs in C compilers”. In: *PLDI*. ACM, 2011, pp. 283–294. DOI: 10.1145/1993498.1993532.