Formal Verification of a C Value Analysis Based on Abstract Interpretation *

Sandrine Blazy[1], Vincent Laporte[1], Andre Maroneze[1], and David Pichardie[2]

[1] IRISA - Université Rennes 1
[2] Harvard University / INRIA

Abstract. Static analyzers based on abstract interpretation are complex pieces of software implementing delicate algorithms. Even if static analysis techniques are well understood, their implementation on real languages is still error-prone.
This paper presents a formal verification using the Coq proof assistant: a formalization of a value analysis (based on abstract interpretation), and a soundness proof of the value analysis. The formalization relies on generic interfaces. The mechanized proof is facilitated by a translation validation of a Bourdoncle fixpoint iterator.
The work has been integrated into the CompCert verified C-compiler. Our verified analysis directly operates over an intermediate language of the compiler having the same expressiveness as C. The automatic extraction of our value analysis into OCaml yields a program with competitive results, obtained from experiments on a number of benchmarks and comparisons with the Frama-C tool.

1 Introduction

Over the last decade, significant progress has been made in developing tools to support mathematical and program-analytic reasoning. Proof assistants like ACL2, Coq, HOL, Isabelle and PVS are now successfully applied both in mathematics (e.g., a mechanized proof of the 4-colour theorem [15] and of the Feit-Thompson theorem [16]) and in formal verification of critical software systems (e.g., the CompCert C-compiler [20] and the verified operating system kernel seL4 [18]).

Over the same time, automatic verification tools based on model-checking, static analysis and program proof have become widely used by the critical software industry. The main reason for their success is that they strengthen the confidence we can have in critical software by providing evidence of software correctness. The next step is to strengthen the confidence in the results of these verification tools, and proof assistants seem to be mature and adequate for this task. This paper presents a foundational step towards the formal verification of a static analysis based on abstract interpretation [10]: the formal verification using the Coq proof assistant of a value-range analysis operating over a real-world language.

* This work was supported by Agence Nationale de la Recherche, grant number ANR-11-INSE-003 Verasco.

Static analyzers based on abstract interpretation are complex pieces of software that implement delicate symbolic algorithms and numerical computations. Their design requires a deep understanding of the targeted programming language. Misinterpretations of the programming language informal semantics may lead to subtle soundness bugs that may be hard to detect by using only testing techniques. Implementing a value analysis raises specific issues related to low-level numeric computations. First, the analysis must handle the machine arithmetic that is (more or less) defined in the programming language. Second, some computations done by the analyzer rely on this machine arithmetic.

Thus, a prerequisite for implementing a static analyzer operating over a C-like language is to rely on a formal semantics of the programming language defining precisely the expected behaviors of any program execution (and including low-level features such as machine arithmetic). Such formal semantics are defined in the CompCert compiler (and it is unusual for a compiler). More precisely, each language of the compiler is defined by a formal semantics (in Coq) associating observable behaviors to any program. Observable behaviors include normal termination, divergence, abnormal termination and undefined behaviors (such as out-of-bounds array access). We have chosen one language of the compiler (the main intermediate language that has the same expressiveness as C, see Section 2) and we have formalized a static analyzer operating over this language. The advantage of this approach is that our analyzer as well as the formal semantics operate exactly over the same language.

The main peculiarity of the CompCert C-compiler is that it is equipped with a proof of semantic preservation [20]. This proof is made possible thanks to the formal semantics of the languages of the compiler. The proof states that any compiled program behaves exactly as specified by the semantics of its original program. It consists of the composition of correctness proofs for each compiler pass and thus involves reasoning on the different intermediate languages of the compiler.

All results presented in this paper have been mechanically verified using the Coq proof assistant. The complete Coq development is available online at http://www.irisa.fr/celtique/ext/value-analysis.

The paper makes the following contributions.

- It provides the first verified value analysis for a realistic language such as C and hence demonstrates the usability of theorem proving in static analysis of real programs.
- It presents a modular design with strong interfaces aimed at facilitating any further extension.
- It provides a reference description of basic techniques of abstract interpretation and thus gives advice on how to use the abstract interpretation methodology for this kind of exercice while maintaining a sufficiently low cost in terms of formal proof effort.
- It compares the performances of our tool (that has been generated automatically from our formalization and integrated into the CompCert compiler) with those of two interval-based value analyzers for C.

The paper exposes many examples taken from the formal development. It is structured to follow the development of a C value analysis based on abstract interpretation; from generic abstract domains (section 3), to fixpoint resolution (section 4) and numerical and memory abstractions (sections 5 and 6). Section 7 describes the experimental evaluation of our implementation. Related work is discussed in Section 8, followed by concluding remarks.

2 Background

This section starts with a short introduction to the Coq proof assistant. It is followed by a brief presentation of the CompCert architecture and memory model. The language our analyzer operates over is described at the end of this section.

2.1 Short Introduction to Coq

Coq is an interactive theorem prover. It consists in a strongly typed specification language and a language for conducting machine-checked proofs interactively. The Coq specification language is a functional programming language as well as a language for inductively defining mathematical properties, for which it has a dedicated type (`Prop`). Induction principles are automatically generated by Coq from inductive definitions, thus inductive reasoning is very convenient. Data structures may consist of properties together with dependent types. Coq's type system includes type classes. Coq specifications are usually defined in a modular way (e.g., using record types and functors, that are functions operating over structured data such as records). The user is in charge to interactively build proofs in the system but those proofs are automatically machine-checked by the Coq kernel. OCaml programs can be automatically generated by Coq from Coq specifications. This process is called extraction.

2.2 The CompCert Memory Model

There are 11 languages in the CompCert compiler, including 9 intermediate languages. These languages feature both low-level aspects such as pointers, pointer arithmetic and nested objects, and high-level aspects such as separation and freshness guarantees. A memory model [21] is shared by the semantics of all these languages. Memory states (of type `mem`) are collections of blocks, each block being an array of abstract bytes. A block represents a C variable or an invocation of `malloc`. Pointers are represented by pairs `(b,i)` of a block identifier and a byte offset `i` within this block. Pointer arithmetic modifies the offset part of a pointer value, keeping its block identifier part unchanged.

Values stored in memory are the disjoint union of 32-bit integers (written as `vint(i)`), 64-bit floating-point numbers, locations (written as `vptr(b,i)`), and the special value `undef` representing the contents of uninitialized memory. Pointer values `vptr(b,i)` are composed of a block identifier `b` and an integer byte offset `i` within this block. Memory chunks appear in memory operations `load` and `store`, to describe concisely the size, type and signedness of the value being stored.

Values:	$v ::= \texttt{vint}(i) \mid \texttt{vfloat}(f) \mid \texttt{vptr}(b, i)$	
	$\mid \texttt{undef}$	
Mem. chunks:	$\kappa ::= \texttt{Mint8signed} \mid \texttt{Mint8unsigned}$	8-bit integers
	$\mid \texttt{Mint16signed} \mid \texttt{Mint16unsigned}$	16-bit integers
	$\mid \texttt{Mint32}$	32-bit integers or pointers
	$\mid \texttt{Mfloat32}$	32-bit floats
	$\mid \texttt{Mfloat64}$	64-bit floats

In CompCert, a 32-bit integer (type `int`) is defined as a Coq arbitrary-precision integer (type `Z`) plus a property called `intrange` that it is in the range 0 to 2^{32} (excluded). The function `signed` (resp. `unsigned`) gives an interpretation of machine integers as a signed (resp. unsigned) integer. The properties `signed_range` and `unsigned_range` are examples of useful properties for machine integers.

```
Definition max_unsigned : Z := 2^32 - 1.
Definition max_signed : Z := 2^31 - 1.
Definition min_signed : Z := - 2^31.
Record int := {   intval: Z;
                  intrange: 0 ≤ intval < 2^32 }.
Definition unsigned (n: int) : Z := intval n.
Definition signed (n: int) : Z := if unsigned(n) < 2^31 then unsigned(n)
                                  else unsigned(n) - 2^32.
Theorem signed_range: ∀ i, min_signed ≤ signed(i) ≤ max_signed.
  Proof.  (* Proof commands omitted here *)  Qed.
Theorem unsigned_range: ∀ i, 0 ≤ unsigned(i) ≤ max_unsigned.
  Proof.  (* Proof commands omitted here *)  Qed.
```

2.3 The CFG Intermediate Language

The main intermediate language of the CompCert compiler is called Cminor, a low-level imperative language structured like C into expressions, statements and functions. Historically, Cminor was the target language of the compiler front-end. There are four main differences with C [20]. First, arithmetic operators are not overloaded. Second, address computations are explicit, as well as memory access (using load and store operations). Third, control structures are if-statements, infinite loops, nested blocks plus associated exits and early returns. Last, local variables can only hold scalar values and they do not reside in memory, making it impossible to take a pointer to a local variable like the C operator `&` does. Instead, each Cminor function declares the size of a stack-allocated block, allocated in memory at function entry and automatically freed at function return. The expression `addrstack(n)` returns a pointer within that block at constant offset `n`.

Cminor was designed to be the privileged language for integrating within CompCert other tools operating over C and other compiler front-ends. For instance, two front-end compilers from functional languages to Cminor have been connected to CompCert using Cminor, and a separation logic has been defined for

Cminor [2]. The Concurrent Cminor language extends Cminor with concurrent features and lies at the heart of the Verified Software Toolchain project [1].

As control-flow is still complex in Cminor (due to the presence of nested blocks and exits), we have first designed a new intermediate language called CFG that is adapted for static analysis: 1) its expressions are Cminor expressions (i.e., side-effect free C expressions), 2) its programs are represented by their control flow graphs, with explicit program points and 3) the control flow is restricted to simple unconditional and conditional jumps. The CFG syntax is defined in Figure 1. Floating-point operators are omitted in the figure, as our analysis does not compute any information about floats. Statements include assignment to local variables, memory stores, if-statements and function calls. Expressions include reading local variables, constants and arithmetic operations, reading store locations, and conditional expressions. As in the memory model, loads and stores are parameterized by a memory chunk κ.

The CFG language is integrated into the CompCert compiler, as shown in Figure 2. There is a translation from Cminor to CFG and a theorem stating that any terminating or diverging execution of a CFG program is also a terminating or diverging execution of the original Cminor program. Thus, instead of analyzing Cminor programs, we can analyze CFG programs and use this theorem to propagate the results of the CFG analysis on Cminor programs. For instance, in order to show that Cminor is memory safe, we only need to show that CFG is memory safe.

For the purpose of the experiments that we conduct in Section 7, we use an inlining pass recently added to the CompCert compiler. It was implemented and proved correct by X.Leroy for another language of the compiler, RTL, that is similar to CFG except that it only handles flat expressions. Since our analysis operates on CFG, we have adapted this inlining pass to CFG. Adapting the soundness proof of this transformation to CFG has been left for future work.

The concrete semantics of CFG is defined in small-step style as a transition relation between execution states. An execution state is a tuple called σ. Among the components of σ are the current program point (i.e., a node in the control-flow graph), the memory state (type `mem`) and the environment (type `env`) mapping program variables to values. We use $\sigma.E$ to denote the environment of a state σ, and $\texttt{dom}(\sigma.E)$ to denote its domain (i.e., the set of its variables). We use $\texttt{reach}(P)$ to denote the set of states belonging to the execution trace of P.

Our value analysis (called `value_analysis`) computes for each program point the estimated values of the program variables. When the value of a variable is an integer `i` or a pointer value of offset `i`, the estimate provides two numerical ranges `signed_range` and `unsigned_range`. The first one over-approximates the signed interpretation of `i` and the other range over-approximates its unsigned interpretation. We note `ints_in_range (signed_range, unsigned_range) i` this fact. Thus, given a program P, $\texttt{value\_analysis}(P)$ yields a map such that for each node l in its control flow graph and each variable v, $\texttt{value\_analysis}(P)[l, v]$ is a pair of sound ranges for v. The following theorem states the soundness of the value analysis: for every program state that may be reached during the execution

Constants:	c ::=	$n \mid f$	integer and floating-point constants
		$\mid$ `addrsymbol`(id, n)	address of a symbol plus an offset
		$\mid$ `addrstack`(n)	stack pointer plus a given offset
Expressions:	a ::=	id	variable identifier
		$\mid c$	constant
		$\mid op_1\ a$	unary arithmetic operation
		$\mid a_1\ op_2\ a_2$	binary arithmetic operation
		$\mid a_1 ?\ a_2 : a_3$	conditional expression
		$\mid$ `load`(κ, a)	memory load
Unary op.:	op_1 ::=	`cast8unsigned`	8-bit zero extension
		$\mid$ `cast8signed`	8-bit sign extension
		$\mid$ `cast16unsigned`	16-bit zero extension
		$\mid$ `cast16signed`	16-bit sign extension
		$\mid$ `boolval`	0 if null, 1 if non-null
		$\mid$ `negint`	integer opposite
		$\mid$ `notbool`	boolean negation
		$\mid$ `notint`	bitwise complement
Binary op.:	op_2 ::=	`+` $\mid$ `-` $\mid$ `*` $\mid$ `/` $\mid$ `%`	arithmetic integer operators
		$\mid$ `<<` $\mid$ `>>` $\mid$ `&` $\mid$ `|` $\mid$ `^`	bitwise operators
		$\mid$ `/`$_u$ $\mid$ `%`$_u$ $\mid$ `>>`$_u$	unsigned operators
		$\mid$ `cmp`(b)	integer signed comparisons
		$\mid$ `cmpu`(b)	integer unsigned comparisons
Comparisons:	b ::=	`<` $\mid$ `<=` $\mid$ `>` $\mid$ `>=` $\mid$ `==` $\mid$ `!=`	relational operators
Statements:	i ::=	`skip`(l)	no operation (go to l)
		$\mid$ `assign`(id, a, l)	assignment
		$\mid$ `store`(κ, a, a, l)	memory store
		$\mid$ `if`(e, l_{true}, l_{false})	if statement
		$\mid$ `call`$(sig, id^?, a, a*, l)$	function call
		$\mid$ `return`$(a)^?$	function return

Fig. 1. Abstract syntax of CFG

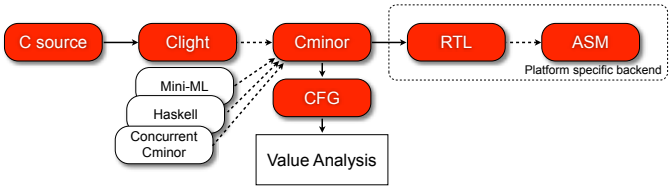

Fig. 2. Integration of the value analysis in the CompCert toolchain

of a program, any program point and variable, every variable valuation computed by the analysis is a correct estimation of the exact value given by the concrete semantics.

Theorem 1 (Soundness of the value analysis). *Let P be a program, $\sigma \in \texttt{reach}(P)$ and $res = \texttt{value\_analysis}(P)$ be the result of the value analysis. Then, for each program point l, for each local variable $v \in \texttt{dom}(\sigma.E)$ that contains an integer i (i.e., $\sigma.E(v) = \texttt{vint}(i) \vee \exists b, \sigma.E(v) = \texttt{vptr}(b, i)$), the property ($\texttt{ints\_in\_range}$ $res[l, v]$ i) holds.*

2.4 Overview of a Modular Value Analysis

Our value analysis is designed in a modular way: a generic fixpoint iterator operates over generic abstract domains (see Section 3). The iterator is based on the state-of-the-art Bourdoncle [6] algorithm that provides both efficiency and precision (see Section 4).

The modular design of the abstract domains is inspired from the design of the Astrée analyzer. It consists in three layers that are showed in Figure 3. The simplest domains are numerical abstract domains made of intervals of machine integers. These domains are not aware of the C memory model.

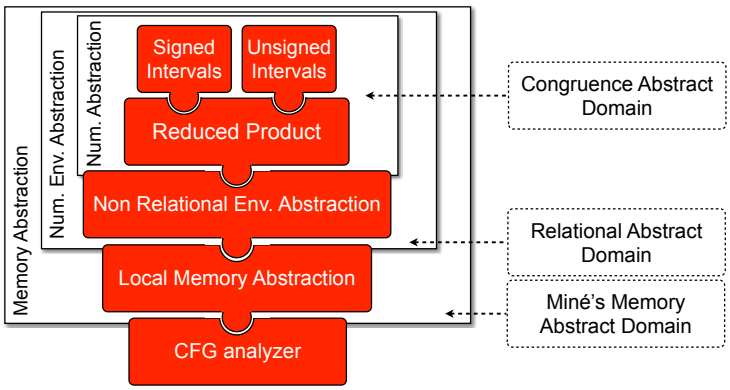

Fig. 3. Design of abstract domains: a three-layer view

In a C program, a same piece of data can be used both in signed and unsigned operations, and the results of these operations differ from one interpretation to the other. Thus, we have two numerical abstract domains, one for each interpretation. Our analysis computes the reduced product of the two domains in order to make a continuous fruitful information exchange between these two domains (see Section 5).

Then, we build abstract domains representing numerical environments. We provide a non-relational abstraction that is parameterized by a numerical abstract domain. The last layer is the abstract domain representing memory. It is

parameterized by the previous layer and links the abstract interpreter with the numerical abstract domains (see Section 6).

This modular design is targeted to connect at each layer other abstract domains. They are represented in dotted lines in Figure 3. For example, several abstract memory models can be used instead of the current one while maintaining the same interfaces with the rest of the formal development. The ultimate goal is to enhance our current abstract interpreter in order to connect it to a memory domain *à la* Miné [23]. The current interfaces are also compatible with any relational numerical abstract domain. At the top, more basic numerical abstractions as congruence could be added and plugged into our reduced product.

3 Abstract Domain Library

This section describes the library we have designed to represent our abstract domains. First, it defines generic abstract domains. Then, it details the interval abstract domain. Last, it explains how to combine abstract domains.

3.1 Abstract Domain Interface

Abstract interpretation provides various frameworks [10] for the design of abstract semantics. The most well-known framework is based on Galois connections but some relaxed frameworks exist. They are generally used when some useful abstraction does not fulfill standard properties (e.g., polyhedral abstract domains [12] do not form a complete lattice). In our context, a relaxed framework is required because of the associated lightweight proof effort.

Since our main goal is to provide a formal proof of soundness for the result of an analysis, some additional properties such as best approximation or completeness do not require a machine checked proof. In some previous work of the last author, a framework has been defined for the purpose of machine checked proofs [14]. In this paper, we push further this initiative and provide a more minimalist framework. The signature of abstract domains is of the following form. [3]

```
Notation 𝒫(A) := (A → Prop).        (* identify sets and predicates *)
Notation x ∈ P := (P x).
Notation P1 ⊆ P2 := (incl P1 P2).   (* property inclusion *)

Record adom (A:Type) (B:Type) : Type := {
  le: A → A → bool;                  (* partial order test        *)
  top: A;                            (* greatest element          *)
  join: A → A → A;                   (* least upper bound         *)
  widen: A → A → A;                  (* widening operator         *)
  gamma: A → 𝒫(B);                   (* concretization function   *)
  gamma_monotone:                    (* monotonicity of gamma     *)
   ∀ a1 a2, le a1 a2 = true → (gamma a1) ⊆ (gamma a2);
  top_sound:                         (* top over-approximates any *)
```

[3] In this paper, for the sake of simplicity, we only use records to structure our formalization. However, in our development, we also use more advanced Coq features such as type classes.

```
    ∀ x, x ∈ (gamma top);              (*   concrete property       *)
  join_sound: ∀ x y:A,                 (* join over-approximates    *)
    (gamma x) ∪ (gamma y) ⊆ gamma (join x y);  (* concrete union  *)
}.
```

Here, `A` is the type of abstract values, `B` is the type of concrete values, and the type of the abstract domain is `(adom A B)`. This type is a record with various operators (described on the right part) and properties about them. This record contains only three properties: the monotonicity of the `gamma` operator, the soundness of the `top` element and the soundness of the least upper bound operator `join`. We do not provide formal proof relating the abstract order with `top` or `join`. Indeed any *weak-join* will be suitable here. The lack of properties about the widening operator is particularly surprising at first sight. In fact, as we will explain in Section 4, the widening operator is used only during fixpoint iteration and this step is validated *a posteriori*. Thus, only the result of this iteration step is verified and we don't need a widening operator for that purpose.

The `gamma` operator of *every* abstract domain will be noted γ. The type class mechanism enables Coq to infer which domain it refers to.

3.2 Example of Abstract Domain: Intervals

Our value analysis operates over compositions of abstract domains. The most basic abstract domain is the domain of intervals. Figure 4 defines the abstract domain of intervals made of machine integers, that are interpreted as signed integers. This instance is called `signed_itv_adom`. The definitions are standard and only some of them are detailed in the figure. An interval represents the range of the signed interpretation of a machine integer. Thus, `top` is defined as the largest interval with bounds `min_signed` and `max_signed`. The concretization is defined as follows. A machine integer `n` belongs to the concretization of an interval `itv` iff `signed(n)` belongs to `itv`. The proof of the lemma `top_sound` follows from the `signed_range` theorem given in Section 2.2.

```
Record itv := {min: Z; max: Z}.

Definition signed_itv_adom : adom itv int := {
   le := (λ itv1 itv2, ...);          (* definition omitted here *)
   top := { min:= min_signed; max:= max_signed};
   join := (λ itv1 itv2, ...);        (* definition omitted here *)
   widen := (λ itv1 itv2, ...);       (* definition omitted here *)
   gamma := (λ itv n, itv.min ≤ signed(n) ≤ itv.max);
   top_sound := (...);                (* proof term omitted here *)
   gamma_monotone := (...);           (* proof term omitted here *)
   join_sound := (...);               (* proof term omitted here *)
}.
```

Fig. 4. An instance called `signed_itv_adom`: the domain of intervals (made of signed machine integers) with a concretization to $\mathcal{P}$`(int)`.

We also define a variant of this domain with a concretization using an unsigned interpretation of machine integers: `(λ itv n, itv.min≤unsigned(n)≤itv.max)`. As explained in Section 5, combining both domains recovers some precision that may be lost when using only one of them.

The `itv` record type provides only lower and upper bounds of type `Z`. Using the expressiveness of the Coq type system, we could choose to add an extra field requiring a proof that `min ≤ max` holds. While elegant at first sight, this would be rather heavyweight in practice, since we must provide such a proof each time we build a new interval. For the kind of proofs we perform, if such a property was required, we would generally have an hypothesis of the form `i ∈ (γ itv)` in our context and it would trivially imply that `itv.min ≤ itv.max` holds.

3.3 Abstract Domain Functors

Our library provides several functors that build complex abstract domains from simpler ones.

Direct Product A first example is the product `(adom (A*A') B)` of two abstract domains `(adom A B)` and `(adom A' B)`, where the concretization of a pair `(a,a'):A*A'` is the intersection `(γ a) ∩ (γ a')`.

Lifting a Bottom Element A bottom element is not mandatory in our definition of abstract domains because some sub-domains do not necessarily contain one. For instance, the domain of intervals does not contain such a least element. Still in our development, the bottom element plays a specific and important role since we use it for reduction. We hence introduce a polymorphic type `A+⊥` that lifts a type `A` with an extra bottom element called `Bot`. We then define a simple functor `lift_bot` that lifts any domain `(adom A B)` on a type `A` to a domain on `A+⊥`. In this new domain, the concretization function extends the concretization of the input domain and `γ Bot = ∅`.

```
Definition botlift (A:Type): Type := Bot | NotBot (x:A).
Notation A+⊥ := (botlift A).
Definition lift_bot (A B: Type): adom A B → adom (A+⊥) B :=
    (* definition omitted here *)
```

Finite Reduced Map Lifted domains are used for instance as input for an important functor of finite maps. CompCert uses intensively the `TREE` interface. Given an implementation `T` of the interface `TREE` and a type `A`, an element of type `(TREE.t A)` represents a partial map from keys (of type `T.elt`) to values of type `A`. The interface is implemented for several kinds of keys in the CompCert libraries. In our development, we use it to map variables to abstract values, but also program points to abstract environments. The functor implements the following type.

```
AbTree.make(T:TREE)(A B:Type): adom A B → adom (T.t A)+⊥ (T.elt → B)
```

An element in `(T.t A)+⊥` is turned into a function of type `T.elt → A+⊥` via the function `get` that satisfies the following equations.

```
get(Bot)(k)      = Bot
get(NotBot m)(k) = top          (* if m[k] is undefined *)
get(NotBot m)(k) = NotBot m[k]  (* otherwise *)
```

As a consequence, the `top` element is represented in a lazy way: a key is associated to it as soon as it is not bound in the partial map. Furthermore, the map is reduced w.r.t. the bottom element of the input domain: as soon as we try to bind a key to the bottom element, the whole map is shrunk to `Bot`. This situation is interesting for dead code elimination and more generally for the whole precision of an analysis.

4 Fixpoint Resolution

From a proof point of view, the main lesson learned from the CompCert experiment is the following. When formally verifying a complex piece of software relying on sophisticated data structures and delicate algorithms, it is not realistic to write the whole software using exclusively the specification language of the proof assistant. A more pragmatic approach to formal verification consists in reusing an existing implementation in order to separately verify its results. This approach is not optimal, but it is worthwhile when the algorithm is a sophisticated piece of code and when the formal verification of each of the results is much easier than the formal verification of the algorithm itself.

The CompCert compiler combines both approaches in order to facilitate the proofs. Most of the compiler passes are written and proved in Coq. A few compiler passes (e.g., the register allocation [26]) are not written directly in Coq, but formally verified in Coq by a translation validation approach. Our value analysis also combines both approaches. We have formally verified a checker that validates *a posteriori* the untrusted results of a fixpoint engine written in OCaml, that finds fixpoints using widening and narrowing operators.

As many data flow analyses, our value analysis can be turned into the fixpoint resolution of an equation system on a lattice. CompCert already provides a classical Kildall iteration framework to iteratively find the least fixpoint of an equation system. But using such a framework is impossible here for two reasons. First, the lattice of bounded intervals contains very long ascending chains that make standard Kleene iterations too slow. Second, the non-monotonic nature of widening and narrowing makes fixpoint iteration sensible to the iteration order of each equation.

We have then designed a new fixpoint resolution framework that relies on the general iteration techniques defined by Bourdoncle [6]. First, Bourdoncle provides a strategy computation algorithm based on Tarjan's algorithm to compute strongly connected subcomponents of a directed graph and find loop headers for widening positioning. This algorithm also orders each strongly connected subcomponent in order to obtain an iteration strategy that iterates inner loops until stabilization before iterating outer loops. Bourdoncle then provides an efficient fixpoint iteration algorithm that iterates along the previous strategy and requires a minimum number of abstract order tests to detect convergence.

This algorithm relies on advanced reasoning in graph theory and formally verifying it would be highly challenging. This frontal approach would also certainly be too rigid because widening iteration requires several heuristic adjustments to reach a satisfactory precision in practice (loop unrolling, delayed widenings, decreasing iterations). We have therefore opted for a more flexible verification strategy: as Bourdoncle strategies, fixpoints are computed by an external tool (represented by the function called `get_extern_fixpoint`) and we only formally verify a fixpoint checker (called `check_fxp`).

Our fixpoint analyzer is defined below, given an abstract domain `ab`, a program `P` and its entry point `entry`, the transfer functions `transfer` and initial abstract values `init`.

```
Definition solve_pfp (ab: adom t B) (P: PTree.t instruction)
  (entry: node) (transfer: node→instruction→list(node*(t→t)))
  (init: t) : node → t :=
  let fxp := get_extern_fixpoint entry ab P transfer init in
  if check_fxp entry ab P transfer init fxp then fxp else top.
```

The verification of the fixpoint checker yields the following property: the concretization of the result of the `solve_pfp` function is a post-fixpoint of the concrete transfer function. That is, given the analysis result `fxp`, for each node `pc` of the program, applying the corresponding transfer function `tf` to the analysis result yields an abstract value included in the analysis result.

```
Lemma solve_pfp_postfixpoint: ∀ ab entry P transfer init fxp,
  fxp = solve_pfp ab P entry transfer init →
  ∀ pc i, P[pc] = i →
  ∀ (pc',tf) ∈_list (transfer pc i), γ(tf(fxp pc)) ⊆ γ(fxp pc').
Proof. (* proof commands are omitted here *) Qed.
```

5 Numerical Abstraction

Following the design of the Astrée analyzer [11], our value analysis is parameterized by a numerical abstract domain that is unaware of the C memory model. We first present the interface of abstract numerical environments, then how we abstract numerical values in order to build non relational abstract environments. Finally, we show concrete instances of numerical domains and how they can be combined.

5.1 Abstraction of Numerical Environments

The first interface captures the notion of numerical environment abstraction. Given a type `t` for abstract values and a notion of variable `var` (simple positive integers in our development), we require an abstract domain that concretizes to $\mathcal{P}$(`var` → `int`) and provide three sound operators `range`, `assign` and `assume`.

```
sign_flag ::= Signed | Unsigned
Definition ints_in_range (r:sign_flag → itv+⊥) : int :=
                          (γ (r Signed)) ∩ (γ (r Unsigned)).
Record int_dom (t:Type) := {
```

```
  int_adom: adom t (var → int); (* abstract domain structure *)
  (* signed/unsigned range of an expression *)
  range: nexpr → t → sign_flag → itv+⊥;
  range_sound: ∀ e ρ ab,
    ρ ∈ γ ab → eval_nexpr ρ e ⊆ ints_in_range (range e ab);
  (* assignment of a variable by a numerical expression *)
  assign: var → nexpr → t → t;
  assign_sound: ∀ x e ρ n ab,
    ρ ∈ γ ab → n ∈ eval_nexpr ρ e → (upd ρ x n) ∈ γ (assign x e ab);
  (* assume a numerical expression evaluates to true *)
  assume: nexpr → t → t;
  assume_sound: ∀ e ρ ab,
    ρ ∈ γ ab → Ntrue ∈ eval_nexpr ρ e → ρ ∈ γ (assume e ab)
}.
```

This interface matches with any implementation of a relational abstract domain [12] on machine integers. To increase precision, it relies on a notion of expression tree (type `nexpr`) defined as follows and relying on CFG operators.

$e_{tr} ::= \texttt{NEvar}\ id \mid \texttt{NEconst}\ c \mid \texttt{NEunop}\ op_1\ e_{tr} \mid \texttt{NEbop}\ op_2\ e_{tr}\ e_{tr} \mid \texttt{NEcond}\ e_{tr}\ e_{tr}\ e_{tr}$

These expressions are associated with a big-step operational semantics `eval_nexpr` of type `(var→int) → nexpr → 𝒫(int)` that we define as a partial function represented by a relation. The semantics is not detailed in this paper.

5.2 Building Non-relational Abstraction of Numerical Environments

Implementing a fully verified relational abstract domain is a challenge in itself and it is not in the scope of this paper. We implement instead the previous interface with a standard non relational abstract environment of the form `var → V♯` where `V♯` abstracts numerical values. The notion of abstraction of numerical values is captured by the following interface.

```
Record num_dom (t:Type) := {
  num_adom : adom t int;                (* abstract domain structure *)
  meet: t → t → t+⊥;          (* over-approximation of the concrete *)
  meet_sound: ∀ x y, (γ x) ∩ (γ y) ⊆ γ (meet x y); (* intersection *)
  range: t → sign_flag → itv+⊥;           (* signed/unsigned range *)
  range_sound: ∀ x:t, γ x ⊆ ints_in_range (range x);
  const: constant → t; const_sound:  (*omitted*);
  forward_unop: unary_operation → t → t+⊥;
  forward_unop_sound: ∀ op x,
    Eval_unop op (γ x) ⊆ γ (forward_unop op x);
  forward_binop: (* omitted *); forward_binop_sound: (* omitted *);
  backward_unop: (* omitted *); backward_unop_sound: (* omitted *);
  backward_binop: binary_operation → t → t → t → t+⊥ * t+⊥;
  backward_binop_sound: ∀ op x y z i j k,
    eval_binop op i j k → i ∈ (γ x) → j ∈ (γ y) → k ∈ (γ z) →
    let (x',y') := backward_binop op x y z in
      i ∈ (γ x') ∧ j ∈ (γ y')
}.
```

It is defined as a carrier `t`, an abstract domain structure `num_adom` and a bunch of *abstract transformers*. Some operators are forward ones: they provide properties about the output of an operation. For instance, the operator `const` builds an

abstraction of a single value. Some operators are backward ones: given some properties about the input and expected output of an operation, they provide a refined property about its input. Each operator comes with a soundness proof.

We also implement a functor that lifts any abstraction of numerical values into a numerical environment abstraction. It relies on the functor for finite reduced maps that we have presented at the end of Section 3. Here, `PTree` provides an implementation of the `TREE` interface for the `var` type.

```
NonRelDom.make(t): num_dom t → int_dom ((PTree.t t)+⊥)
```

The most advanced operator in this functor is the `assume` function. It relies on a backward abstract semantics of expressions.

```
Fixpoint backward_expr (e:nexpr) (ab:t) (itv:Val) : t :=
  match e with
    | ...
    | NEcond b l r ⇒
        join
          (backward_expr b (backward_expr r ab itv) (const Nfalse))
          (backward_expr b (backward_expr l ab itv)
               (backward_unop boolval (eval_expr b ab) (const Ntrue)))
  end.
```

We just show and comment the case of conditional expressions. Given such an expression `NEcond b l r`, an abstract environment `ab` and the expected value `itv` of this expression, we explore the two branches of the condition. In one case, the condition `b` evaluated to `Nfalse` and the *right* branch `r` evaluated to `itv`. In the other case, the condition `b` evaluated to anything whose boolean value is `Ntrue` and the *left* branch `l` evaluated to `itv`. Then we have to consider that any of the two branches might have been taken, hence the join.

Equipped with such backward operators, the analysis is then able to deal with complex conditions like the following: `if (0 <= x && x < y && y < z && z < t && t < u && u < v && v < 10)`. When analysing the true branch of this `if`, it is sound to assume that the condition holds. The backward operator will propagate this information and infer one bound for each variable. Since backward evaluation of conditions goes right to left, the following bounds are inferred: $v < 10$, $u < 9$, $t < 8$, $z < 7$, $y < 6$, and $0 \leq x < 5$. Unfortunately, no information is propagated from left to right. However applying again the `assume` function does propagate information between the various conditions. Iterating this process finally yields the most precise intervals for all variables involved in this condition.

Notice that inferring such precise information is possible thanks to the availability of complex expressions in the analyzed CFG program. Compare for example with Frama-C which, prior to any analysis, destructs boolean operations into nested `if`s; it is thus unable to give both bounds for each variable.

5.3 Abstraction of Numerical Values: Instances and Functor

We gave two instances of the numerical value abstraction interface: the intervals of signed integers and the intervals of unsigned integers. Several operations are defined on intervals together with their proofs of correctness. We have to take

into account machine arithmetic. We do not try to precisely track integers that wrap-around intentionally. Instead we systematically test if an overflow may occur and fall back to `top` when we can't prove the absence of overflow.

```
Definition repr  (i: itv): itv := if leb i top then i else top.
Definition add (i j: itv): itv :=
  repr { min := i.min + j.min; max := i.max + j.max}.
```

We also rely on a reduction operator when the result of an operation may lead to an empty interval. Since our representation of intervals contains several elements with the same (empty) concretization, it is important to always use a same representative for them.[4]

```
Definition reduce (min max:Z): itv+⊥ :=
   if min ≤ max then NotBot (ITV min max) else Bot.

Definition backward_lt (i j: itv): itv+⊥ * itv+⊥ :=
      (meet i (reduce min_signed (j.max-1)),
       meet j (reduce (i.min+1) max_signed)).
```

At run-time, there are no *signed* or *unsigned* integers; there are only *machine* integers that are bit arrays whose interpretation may vary depending on the operations they undergo. Therefore choosing one of the two interval domains may hamper the precision of the analysis. Consider the following example C program.

```
1  int main(void) { signed s; unsigned u;
2    if (*) u = 2^31 - 1; else u = 2^31;
3    if (*) s = 0; else s = -1;
4    return u + s;  }
```

At the end of line 2, an unsigned interval can exactly represent the two values that the variable `u` may hold. However, the least signed interval that contains them both is `top`. Similarly, at the end of line 3, a signed interval can precisely approximate the content of variable `s` whereas an unsigned interval would be extremely imprecise. Moreover, comparison operations can be precisely translated into operations over intervals (e.g., intersections) only when they share the same signedness. Therefore, so as to get as precise information as possible, we need to combine the two interval domains. This is done through reduction.

To combine abstractions of numerical values in a generic and precise way, we implement a functor that takes two abstractions and a sound reduction operator and returns a new abstraction based on their reduced product.

```
Definition reduced_product (t t':Type) (N:num_dom t) (N':num_dom t')
  (R:reduction N N') : num_dom (t*t') := (* omitted definition *)
```

A reduction is made of an operator ρ and a proof that this operator is a sound reduction.

```
Record reduction (A B:Type) (N1:num_dom A) (N2:num_dom B) := {
  ρ: A+⊥ → B+⊥ → (A * B)+⊥;
  ρ_sound: ∀ a b, (γ a) ∩ (γ b) ⊆ γ (ρ a b)                  }
```

[4] Otherwise the analyzer may encounter two equivalent values without noticing it and lose precision.

Each operator of this functor is implemented by first using the operator of both input domains and then reducing the result with ρ. We hence ensure that each encountered value is systematically of the form ρ `a b` but we do not prove this fact formally, avoiding the heavy manipulation of quotients. Note also that, for soundness purposes, we do not need to prove that reduction actually reduces (i.e., returns a lower element in the abstract lattice)!

6 Memory Abstraction

The last layer of our modular architecture connects the CFG abstract interpreter with numerical abstract domains. It aims at translating every C feature into useful information in the numerical world. On the interpreter side, the interface with this *abstract memory model* is called `mem_dom`. It consists in trees made of CFG expressions and four basic commands `forget`, `assign`, `store` and `assume`.

```
Record mem_dom (t:Type) := { (* abstract domain with concretization
   to local environment and global memory *)
  mem_adom: adom t (env * mem);
  (* consult the range of a local variable *)
  range: t → ident → sign_flag → itv+⊥;
  range_sound: ∀ ab e m x i,
    (e,m) ∈ γ ab → (e[x] = vint(i) ∨ ∃ b, e[x] = vptr(b,i)) →
    i ∈ (ints_in_range (range ab x));
  (* project the value of a local variable *)
  forget: ident → t → t;
  forget_sound: ∀ x ab, Forget x (γ ab) ⊆ γ (forget x ab);
  (* assign a local variable *)
  assign: ident → expr → t → t;
  assign_sound: ∀ x e ab, Assign x e (γ ab) ⊆ γ (assign x e ab);
  (* assign a memory cell *)
  store: memory_chunk → expr → expr → t → t;
  store_sound: ∀ κ l r ab,
    Store κ l r (γ ab) ⊆ γ (store κ l r ab);
  (* assume an expression evaluates to non-zero value *)
  assume: expr  → t → t;
  assume_sound: ∀ e ab, Assume e (γ ab) ⊆ γ (assume e ab)
}.
```

Our final analyzer is parameterized by a structure of this type.

```
value_analysis (t:Type) :  mem_dom t →
   program → node → (ident → sign_flag → Interval.itv +⊥)
```

A structure of type `mem_dom` is built with a functor of the following form.

```
  AbMem.make (t:Type) : int_dom t → mem_dom (t*type_info)
```

The numerical abstraction is associated with a flow sensitive type information (of type `type_info`) that we compute at the same time. This type information tries to recover some information to disambiguate integer and pointer values. The abstract domain is built using the product functor presented in Section 3. The concretization function of the numeric domain is lifted from a concretization of type `t` $\rightarrow$ $\mathcal{P}$`(var→int)` to a concretization of type `t` $\rightarrow$ $\mathcal{P}$`(env * mem)` with the following definition. [5]

[5] The types `env` and `mem` are introduced in Section 2.

```
Definition gamma_mem (ab:t) := λ (e,m):(env*mem).
    ∃ ρ:var → int, ρ ∈ (γ ab) ∧
      (∀ x i, (e[x] = vint(i) ∨ ∃ b, e[x] = vptr(b,i)) → ρ x = i).
```

For each transfer function that takes as argument a C expression, we convert it into a numerical expression in order to feed the numerical abstract domain. For instance, the `assign` operator takes the following form.

```
Definition assign (id:ident) (e:expr) (ab:t*type_info): t*type_info :=
  let (nm,tp) := ab in
  (* convert expression e into a numeric form using type infos *)
  match convert tp e with
    | None ⇒ forget id ab (* if we fail, we just project  *)
    | Some ne ⇒
       (* otherwise we call the numerical assignment operator *)
       (num.assign id ne nm, ... (* type info update omitted *))
end.
```

Removing some ambiguity between pointers and integers is mandatory for soundness. As an example, consider the unsigned equality expression `(x ==u y)`. For the sake of precision of the analysis, it is important to convert it into a simple numerical equality `x == y` before using the `assume` operator of the numerical abstract domain. However if `x` contains a numerical value and `y` a pointer, the first formula is always false while assuming the second formula in the numerical world would lead to a spurious assumption about the offset of the pointer in `y`.

7 Experimental Evaluation

Our verified value analyzer takes as input a CFG program and outputs ranges for every variable at every point of the program. Our formal development adds about 7,500 lines of Coq code (consisting of 4,000 lines of Coq functions and definitions and 3,500 lines of Coq statements and proof scripts) and 200 lines of OCaml to the 100,000 lines of Coq and 1,000 lines of OCaml provided in CompCert 1.11.

We have conducted some experiments to evaluate the precision and the efficiency of our analyzer. Indeed, an analyzer that always returns "top" is easily proved correct, but useless. It is therefore important to distinguish between bounded and unbounded variables. Moreover, a precise but non-scalable analyzer has limited applicability. In order to evaluate the precision and efficiency of our value analysis, we use the OCaml extracted code to compile our benchmark programs into CFG programs and to run our analyzer on them.

We compare our analyzer to two interval-based analyzers operating over C programs: a state-of-the-art industrial tool, Frama-C [13], and an implementation of a value-range analyzer [24]. Frama-C is an industrial-strength framework for static analysis, developed at CEA. It integrates an abstract interpretation-based interprocedural value analysis on interval domains with congruence, k-sets and memory analysis. It operates over C programs and has a very deep knowledge of its semantics, allowing it to slice out undefined behaviors for more precise results. It currently does not handle recursive functions. The value-range analyzer, which will be referred to as Wrapped is described in [24]. It relies on LLVM and operates

over its intermediate representation to perform an interval analysis in a signedness-agnostic manner, using so-called "wrapped" intervals to deal with machine integer issues such as overflows while retaining precision. It is an intraprocedural tool, but can benefit from LLVM's inlining to perform interprocedurally in the absence of recursion.

The 3 tools have been compared on significant C programs from CompCert's test suite. They range from a few dozen to a few thousand statements. To relate information from different analyses, we annotated the programs to capture information on integer variables at function entries and exits and at loops (for iteration variables). This amounts to 545 annotations in the 20 programs considered. For each program point, we counted the number of bounded variables. We consider as bounded any variable whose inferred interval has no more than 2^{31} elements, and hence rule out useless intervals like $\texttt{x} \in [-2^{31}, 2^{31} - 2]$, inferred after a guard like `x<y`. Finally, to be able to compare the results of an interprocedural analysis with those of two intraprocedural analyses with inlining, we considered for each annotation the union of the intervals of all call contexts. Less than 10% of intervals present a union of different intervals, and among those several preserve the boundedness for all contexts. Overall, its impact on the results is negligible.

The results are shown in Figure 5, which displays the number of bounded variables per program and per analyzer. In total, Frama-C bounded 398 variables, our analyzer got 355, and Wrapped ended up with 305. The main differences between our analyzer and Frama-C, especially on the larger benchmarks (`lzw`, `arcode` and `lzss`) result from global variable tracking and congruence information. Such reasoning is not handled by our analyzer. On the other hand, the precision of our product of signed and unsigned domains allows us to bound more variables (e.g., on `fannkuch`), where Wrapped also obtains a good score, mainly due to variables bounded as $[0, 2^{31} - 1]$ and similar values. Some issues with the inlining used by Wrapped explain its worse results in `fft`, `knucleotide` and `spectral`.

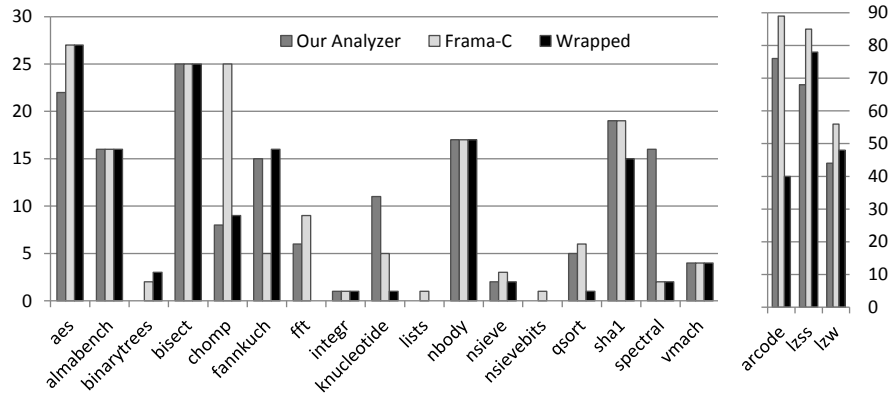

Fig. 5. Number of bounded intervals (bounded per program and analyzer).

We also compared the execution times of the analyses. Overall, our analysis runs faster than Frama-C because we track less information, such as pointers and global variables. For programs without these features, both analyses run in roughly the same time, from a few tenths of seconds for the smaller programs up to a few seconds for the larger ones. Wrapped's analysis is faster than the others. On a larger benchmark (over 3,000 lines of C code and about 10,000 CFG instructions after inlining) our analysis took 34 seconds to perform.

It is hard to draw final conclusions about the precision of our tool from these experiments. Frama-C, for instance, is likely to perform better on specific industrial critical software for which it has been specially tuned. Nevertheless we give evidence that our analyzer performs non-trivial reasoning over the C semantics, close to that of state-of-the-art (non-verified) tools.

8 Related Work

While mechanization of research paper proofs attracts an increasing number of practitioners, it should not be confused with the activity of developing a formally verified compiler or static analyzer. Our work is initially inspired by the achievement of the CompCert compiler [20] but we target the area of abstract-interpretation-based static analyzers.

Previous work on mechanized verification of static analyses has been mostly based on classic data flow frameworks: Klein and Nipkow instantiate this framework for inference of Java bytecode types [19]; Coupet-Grimal and Delobel [9] and Bertot *et al.* [3] for compiler optimizations, and Cachera *et al.* [7] for control flow analysis. Vafeiadis *et al.* [28] rely on a simple data flow analysis to verify a fence elimination optimization for concurrent C programs. Compared to these prior works, our value analysis relies on fixpoint iterations that are accelerating with widening operators. Cachera and Pichardie [8] and Nipkow [25] describe a verified static analysis based on widenings but their technique is restricted to structured programs and targets languages without machine arithmetic nor pointers. Leroy and Robert [22] have developed a points-to analysis in the CompCert framework. This static analysis technique is quite orthogonal to what we formalize here. Their verified tool is not compared to any existing analyzer. Hofmann *et al.* [17] provide a machine-checked correctness proof in Coq for a generic post-fixpoint solver named RLD. The formalized algorithm is not fully executable and cannot be extracted to OCaml code.

Of course the area of non-verified static analysis for C programs is a broader topic. In our context, the most relevant and inspiring works are the static analyses devoted to a precise handling of signed and unsigned integers [27,24] and the Astrée static analyzer [11]. Our current formalization is directly inspired by Astrée's design choices, trying to capture some of its key interfaces. Our current abstract memory model is aligned with the model developed by Miné [23] because we connect a C abstract semantics with a generic notion of numerical abstract domain. Still our treatment of memory is simplified since we only track values of local variables in the current implementation of our analyzer.

9 Conclusion

This work provides the first verified value analysis for a realistic language as C. Implementing a precise value analysis for C is highly error-prone. We hope that our work shows the feasibility of developing such a tool together with a machine-checked proof. The precision of the analysis has been experimentally evaluated and compared on several benchmarks. The paper's technology performs comparably to existing off-the-shelf (unverified!) tools, Frama-C [13] and Wrapped [24]. Our contribution is also methodological. Our formalization, its lightweight interfaces and its proofs can be easily reused to develop different formally verified analyses.

Now that the main interfaces are defined, we expect to improve our analyzer in several challenging directions. First, we want to replace the current memory abstraction with a domain similar to Miné's memory model [23]. Verifying such a domain raises specific challenges not only in terms of semantic proofs but also in terms of efficient implementation of the transfer functions. Without special care, the domain may not be able to scale to large enough programs. We also intend to connect relational abstract domains to the interface for numerical environments. We would like to develop efficient validation techniques following Besson *et al.* [4] approach and test their efficiency on large programs. The last and important challenge concerns floats. Astrée relies on subtle reasoning and manipulation on floats. CompCert has recently been enhanced with a fully verified implementation of floating-point arithmetic [5] and we hope to be able to incorporate them in our own value analysis.

Acknowledgements

We thank Antoine Miné, David Monniaux and Xavier Rival for many fruitful discussions on the Astrée static analyzer. We thank Jacques-Henri Jourdan and Xavier Leroy for integrating the CFG language into the CompCert compiler.

References

1. A. W. Appel. Verified software toolchain. In *Proc of ESOP 2011*, volume 6602 of *LNCS*, pages 1–17. Springer, 2011.
2. A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In *Proc. of TPHOLs 2007*, volume 4732, pages 5–21, 2007.
3. Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Proc. of TYPES 2006*, volume 3839 of *LNCS*, pages 66–81. Springer, 2006.
4. F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Certified result checking for polyhedral analysis of bytecode programs. In *Proc. of TGC 2010*, volume 6084 of *LNCS*, pages 253–267. Springer-Verlag, 2010.
5. S. Boldo, J. Jourdan, X. Leroy, and G. Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In *Proc. of ARITH 21*. IEEE Computer Society Press, 2013. To appear.
6. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of FMPA 1993*, volume 735 of *LNCS*, pages 128–141. Springer, 1993.

7. D. Cachera, T. P. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.
8. D. Cachera and D. Pichardie. A certified denotational abstract interpreter. In *Proc. of ITP-10*, volume 6172 of *LNCS*, pages 9–24. Springer, 2010.
9. S. Coupet-Grimal and W. Delobel. A uniform and certified approach for two static analyses. In *Proc. of TYPES 2004*, volume 3839 of *LNCS*, pages 115–137, 2004.
10. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
11. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyzer. In *Proc. of ESOP 2005*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of POPL 78*, pages 84–97. ACM Press, 1978.
13. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - a software analysis perspective. In *Proc. of SEFM 2012*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.
14. D.Pichardie. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes 1, 2005. In French.
15. G. Gonthier. The Four Colour Theorem: Engineering of a Formal Proof. In *ASCM 2007*, volume 5081 of *LNCS*, page 333. Springer, 2007.
16. G. Gonthier. Engineering mathematics: the odd order theorem proof. In *Proc. of POPL'13*, pages 1–2. ACM, 2013.
17. M. Hofmann, A. Karbyshev, and H. Seidl. Verifying a local generic solver in Coq. In *Proc. of SAS'10*, pages 340–355. Springer, 2010.
18. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. SeL4: formal verification of an operating-system kernel. *Comm. of the ACM*, 53(6):107–115, June 2010.
19. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM TOPLAS*, 28(4):619–695, 2006.
20. X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
21. X. Leroy, A.W. Appel, S. Blazy, and G. Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012.
22. X. Leroy and V. Robert. A formally-verified alias analysis. In *Proc. of CPP 2012*, volume 7679 of *LNCS*, pages 11–26. Springer, 2012.
23. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of LCTES'06*, pages 54–63. ACM, Jun. 2006.
24. J. Navas, P. Schachte, H. Søndergaard, and P. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In *Proc. of APLAS 2012*, volume 7705 of *LNCS*. Springer, 2012.
25. T. Nipkow. Abstract interpretation of annotated commands. In *Proc. of ITP'12*, volume 7406 of *LNCS*, pages 116–132. Springer, 2012.
26. S. Rideau and X. Leroy. Validating register allocation and spilling. In *Proc. of CC 2010*, volume 6011 of *LNCS*, pages 224–243. Springer, 2010.
27. A. Simon and A. King. Taming the Wrapping of Integer Arithmetic. In *Proc. of SAS 2007*, volume 4634 of *LNCS*, pages 121–136. Springer, 2007.
28. V. Vafeiadis and F. Zappa Nardelli. Verifying fence elimination optimisations. In *Proc. of SAS'11*, pages 146–162. Springer, 2011.