An Abstract Memory Functor for Verified C Static Analyzers
Sandrine Blazy, Vincent Laporte, David Pichardie

To cite this version:

HAL Id: hal-01339969
https://inria.hal.science/hal-01339969
Submitted on 25 Oct 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives| 4.0 International License
Abstract interpretation provides advanced techniques to infer numerical invariants on programs. There is an abundant literature about numerical abstract domains that operate on scalar variables. This work deals with lifting these techniques to a realistic C memory model. We present an abstract memory functor that takes as argument any standard numerical abstract domain, and builds a memory abstract domain that finely tracks properties about memory contents, taking into account union types, pointer arithmetic and type casts. This functor is implemented and verified inside the Coq proof assistant with respect to the CompCert compiler memory model. Using the Coq extraction mechanism, it is fully executable and used by the Verasco C static analyzer.
We chose this representation because it is independent of the static analysis of C programs. The design of our memory functor is modular and inspired from Astrée’s design [22].

Our abstract memory functor is integrated into a formally verified static analyzer that is not dealing precisely with memory [3], it operates over an intermediate representation of the CompCert compiler.

The remainder of this paper is organized as follows. First, section 2 recalls the main features of the existing CompCert memory model and introduces the specification of our abstract memory functor. Then, section 3 describes the memory functor we implemented and section 4 reviews its soundness proof. Section 5 describes some improvements that we made to our abstract memory functor, in order to enhance the precision of static analysis. Section 6 describes the experimental evaluation of our analyzer. Section 7 discusses related work, followed by conclusions and perspectives.

2. Background

The value analysis discussed in this paper builds up on the work of [3, 15] on verified static analysis. The architecture of the analyzer is similar to what has been described in [3], but they are using a naive memory abstraction, where for instance, the contents of array cells are not tracked. The Verasco verified static analyzer uses the naive memory abstraction, where for instance, the contents of array is similar to what has been described in [3], but they are using a recent publication [15]. This paper goes into the details of the architecture and also adapted to static analysis [3]. Indeed programs are a semantics preservation property.

The analyzer operates over the CFG intermediate language, that has been added to the CompCert front-end (see Figure 2). We chose this representation because it is independent of the architecture and also adapted to static analysis [3]. Indeed programs are represented by their control-flow graph (hence the name of this language) with explicit program points $l$, and expressions are side-effect free C expressions. More generally, CFG is a low-level imperative language structured like C into expressions, statements and functions. Contrary to C, arithmetic operators are not overloaded and address computations as well as memory access (using load and store operations) are explicit. Local variables can only hold scalar values and they do not reside in memory. Instead, each CFG function declares the size of a stack-allocated block, allocated in memory at function entry and automatically freed at function return. The expression $\text{addrstack}(i)$ returns a pointer within that block at constant offset $i$.

The syntax of CFG is defined in Figure 3. Floating-point operators are omitted in the figure, as our analysis does not compute any information about floats. Expressions include reading local variables, constants, arithmetic operations, and memory loads. Statements include assignment to local variables, memory stores (to a memory location specified by a pointer expression), conditional and unconditional jumps (to a known program point of the same function) and function calls and returns. Memory load expressions and store statements take as parameter a chunk $\kappa$ describing the type of the data being transferred, its size, and signedness.

A function is a finite map from nodes (abstract program points) to statements and each statement lists explicitly the nodes of its predecessor. A program is composed of a list of global variables

<table>
<thead>
<tr>
<th>Constants:</th>
<th>$c ::= i$</th>
<th>integer constant</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f$</td>
<td>floating-point constant</td>
</tr>
<tr>
<td></td>
<td>$\text{addrsym}(id, i)$</td>
<td>address of a symbol + an offset</td>
</tr>
<tr>
<td></td>
<td>$\text{addrstack}(i)$</td>
<td>stack pointer + a given offset</td>
</tr>
</tbody>
</table>

| Chunks: | $\kappa ::= \text{Mint8} \text{Signed}$ | 8-bit integers |
|         | $\text{Mint8} \text{Unsigned}$ | 8-bit integers |
|         | $\text{Mint16} \text{Signed}$ | 16-bit integers |
|         | $\text{Mint16} \text{Unsigned}$ | 16-bit integers |
|         | $\text{Mint32}$ | 32-bit int. or pointers |
|         | $\text{Mfloat32}$ | floats |
|         | $\text{Mfloat64}$ | floats |

| Expressions: | $e ::= i$ | variable identifier |
|             | $c$       | constant |
|             | $a_1 \ op_2 a_2$ | binary arithmetic op. |
|             | $\text{op}_3 e$ | unary arith. operation |
|             | $e_1 \ op_2 e_2$ | binary arith. operation |
|             | $e_1 ? e_2 : e_3$ | conditional expression |
|             | $\text{load}(\kappa, e)$ | memory load |

| Unary op.: | $\text{op}_1 ::= \text{cast} \text{unsigned}$ | 8-bit zero extension |
|           | $\text{cast} \text{signed}$ | 8-bit sign extension |
|           | $\text{cast} \text{unsigned}$ | 16-bit zero extension |
|           | $\text{cast} \text{signed}$ | 16-bit sign extension |
|           | $\text{negint}$ | 0 if null, 1 if non-null |
|           | $\text{notbool}$ | boolean opposite |
|           | $\text{notint}$ | bitwise complement |

| Binary op.: | $\text{op}_2 ::= + | - | * | / | \%$ | arithmetic integer op. |
|            | $\langle \text{int} \rangle$ | bitwise operators |
|            | $\langle \text{int} \rangle$ | unsigned operators |
|            | $\text{cmp}(\circ)$ | int. signed comparisons |
|            | $\text{cmpu}(\circ)$ | integer unsigned comp. |

| Comparisons: | $\circ ::= \langle < | <= | > | >= | == | != \rangle$ | relational operators |
|             | $\langle \text{int} \rangle$ | relational operators |

| Statements: | $\iota ::= \text{assign}(id, e, l)$ | assignment |
|            | $\text{store}(\kappa, e, l)$ | memory store |
|            | $\text{skip}(l)$ | no operation (go to 1) |
|            | $\text{if}(e, l_{\text{true}}, l_{\text{false}})$ | if statement |
|            | $\text{call}(\iota, e, e^*, l)$ | function call |
|            | $\text{return}(e^*)$ | function return |

| Control-flow graph: | $g ::= l \rightarrow i$ | finite map |

| Functions: | $F ::= \{ \text{sig} \}$ | signature |
|            | $\text{id}^*$ | parameters |
|            | $n$ | size of stack data block |
|            | $l$ | label of first instruction |
|            | $g$ | code |

| Programs: | $P ::= \{ \text{dcl}^* \}$ | global variables |
|           | $F^*$ | functions, |
|           | $\text{main} = \text{id}$ | entry point |

---

Figure 2. The CFG intermediate language inside the CompCert toolchain

Figure 3. Abstract syntax of CFG
(with their initialization data), a list of functions, and the name of the function that is the entry point of the program.

The semantics of CFG expressions is defined by a relation called eval-expr and introduced in Figure 4. More precisely, the evaluation (eval-expr ge ρ m a v) states that expression a evaluates to value v. It involves a global environment ge of type gen, a local environment ρ and the current concrete memory m. In the rest of this paper, we will often omit the global environment.

This concrete memory is shared among all intermediate languages of the compiler. More precisely, the concrete memory model of CompCert defines the behavior of the C memory state, which encompasses the heap (dynamically allocated by calls to malloc), the stack (holding local variables that cannot be allocated to machine registers), and global variables. This memory state (of type mem) is a collection of blocks, each being an array of abstract bytes and having a unique identifier. The memory state is defined in Figure 4 as a map from block references to bounds and contents. In memory, fresh blocks are allocated (for global variable allocation or local stack allocation) and released (local stacks are freed when leaving a function).

A particular byte in a given block can be referred to by an (unsigned) 32-bit machine integer. Therefore, a pointer is a value (vptr b of b) that represents the address of the byte at offset of b in the block b. Other values stored in memory are integers, floats, and the special value undefined representing the contents of uninitialized memory. An access to a block, either to read (load operation) from it or to write to it (store operation) is defined by a pointer and a chunk κ (defined in Figure 3). This chunk describes how the transferred value is encoded as a sequence of abstract bytes (on stores) or decoded from such a sequence (on loads) using the decode-val function.

Many memory operations are partial functions (e.g., dereferencing a dangling pointer is not permitted). This is modeled through a permission system: each pointer (i.e., each block-offset pair) is given a permission system: each pointer (i.e., each block-offset pair) is given a permission (None, Readable, Writable, Freeable). The permission is propagated from the block to its offset.

This permission system is very fine-grained: a different permission may also be given to every offset in a block. However, blocks are like arrays: they have bounds such that there is no permission outside the bounds and all offsets within the bounds have the same permission.

2.2 Analyzer Architecture and Numerical Domains

The analyzer has three main components shown in Figure 5: an abstract numerical domain, an abstract memory functor, and a fixpoint solver. The fixpoint solver takes as input the code of a function and an abstract semantics of the CFG language. It then computes a flow-sensitive invariant (a post-fixpoint) that is sound with respect to the given semantics. It uses Bourdoncle’s algorithm (4) to employ widening operators sparsely in the control-flow graph and to iterate abstract transfer functions with an efficient and precise strategy. The iterator also relies on decreasing iterations to enhance precision.

### Values

Values: v ::= vint i | vfloat f | vptr b i | undefined

### Permissions

Permissions: π ::= None | NonEmpty | Readable | Writable | Freeable

### Memory states

Memory states: m ::= b → (lo hi, π, i → v)

### Operations over values

- load-result(κ, v) = v′: Reflected effect of storing a value with a given memory chunk, then reading it back with the same chunk.
- bool-of-val v b with b ∈ {true, false}: Truth values. Non-zero integers are treated as true. The integer 0 (also used to represent the null pointer) is false.

### Operations over memory states

- Mem.alloc m lo hi = (m′, b): Allocate a fresh block with bounds [lo, hi] and a free block.
- Mem.free m b = Some m′: Free (invalidate) the block b.
- Mem.load κ m b i = Some v: Read consecutive bytes (as determined by κ) at block b, offset i of m. If successful, return the contents of these bytes as value v.
- Mem.store κ m b i v = Some m′: Store v as one or several consecutive bytes (as det. by κ) at offset i of block b. If successful, return an updated memory m′.
- Mem.perm m b i = π: Permission of block b at offset i.
- Mem.size_chunk κ = i: Size (number of bytes) that κ holds.

### Global env.

- ge ::= (id → b): map from global variables to block references.
- ρ ::= id → b: map from local variables to block references.

### Evaluation of expressions

- eval-expr ge ρ m a v: Expression evaluates to value v.

**Figure 4.** Concrete memory model, semantics of expressions

**Figure 5.** Modular architecture of the CFG value analyzer

The abstract memory functor is a Coq data-type whose abstract values ab represent sets of concrete memory states. Its signature is given in the first part of Figure 6, where the type of abstract values is called t. The domain is equipped with a lattice structure (mem-wl) and a query operator (range ab × s) that returns an over-approximation of the values of local variable x in any state represented by ab, as an interval enriched with congruence information.
of type strided-interval). The result depends on the signedness interpretation choice $s$.

The domain also features operators (called abstract transformers) that model CFG statements and perform the actual propagation of the analysis, guided by the fixpoint iterator: assume, assign, store and forget. The (assume $e a b$) operator assumes that an expression $e$ evaluates to a non-zero value within all concrete states represented by $a b$. The (assign $x e a b$) operator models the assignment of the value of an expression $e$ to a local variable $x$ within an abstract state $a b$. The (store $\kappa e_1 e_2 a b$) operator models the assignment of the value of an expression $e_1$ to a memory location pointed by the value of expression $e_2$ with chunk $\kappa$ within an abstract state $a b$. The (forget $x a b$) operator performs a non-deterministic assignment to the local variable $x$ in the abstract state $a b$ and can be used for instance to model input statements.

The $t \vdash \perp$ notation refers to the type $t$ extended with an extra Bot element whose concretization is empty: it means that a contradiction has been found and that no concrete state can satisfy the given constraints. For instance, when analyzing a conditional branch, the analyzer may prove that the condition never holds, i.e., that the branch cannot be taken, hence returns Bot.

The analysis will be restricted to programs without functions calls or, equivalently, programs without recursion nor function pointers in which all function calls can be inlined before the analysis. This is a standard hypothesis for a static analyzer operating over C programs. Therefore, there is no operator in this interface that enables us to model function calls.

Numerical constants

of type int and a non-relational interval domain, enhanced with congruence information.

Each abstract transformer of the abstract memory functor comes with a specification (given in the second part of Figure 6); it is defined with respect to a predicate transformer that expresses the concrete semantics as a strongest post-condition. For instance, the $(\text{Store } \kappa e_1 e_2 a b)$ predicate transformer relates a set $E$ of concrete states $(\rho, m)$ — before executing a store instruction — to the set of concrete states $(\rho, m)$ — after the store — that is obtained by storing in memory $m$ a value $v$ at address $(\text{vptr } b i)$ with chunk $\kappa$: the address results from the evaluation of the left-hand-side expression $e_1$ in the environment $\rho$, and the value $v$ results from the evaluation of the right-hand-side expression $e_2$ in the same environment $\rho$. Our implementation of the memory functor is proved sound with respect to these specifications; this leads to the theorems at the bottom of Figure 6.

The implementation of the abstract memory functor takes as parameter a (relational) numerical abstract domain whose abstract values (of type num) represent sets of functions from (abstract) cells to numerical values (machine integers, of type int). It is used to represent the numerical part of the contents of the abstract cells: the values of integers and the offsets of pointers. In our experiments, the functor has been instantiated with a relational polyhedra domain [10] and a non-relational interval domain, enhanced with congruence information [3].
3. Design of the Memory Functor

Relational numerical domains are well suited to represent numerical environments. The operators of such domains always refer to the variables through their names (as in “assign \( x + 2 \cdot y \) to \( z \)”). Such a domain cannot be directly applied to the analysis of languages like C in which: 1) values are not only numerical but also comprise pointers; and 2) variables are referred to by expressions which may involve pointer arithmetic (as in “assign \((x + 2 \cdot y)\) to \((z+4)\)”, where \(p\) denotes the dereference of pointer \(p\)). Such an assignment may modify several memory locations: the exact variable targeted by a pointer expression may not be known statically. The analyzer must account for this uncertainty and consider that any of the possibly targeted locations has been updated.

Programs to analyze manipulate data that is stored in various locations: registers, local and global variables. The numerical content of these locations is represented as a point in a (relational) numerical domain. The numerical content refers to the actual value of integers and the offset part \(i\) of pointers (\(vptr\) \(b\), \(i\)). One of the main tasks of the memory functor is to translate the queries that it receives in terms of CFG expressions (with pointer arithmetic and memory loads) into queries for the numerical domain, in terms of purely numerical expressions.

Let us consider the example program of Figure 8 to be analyzed.

The analyzer operates on the CFG intermediate representation but the program is written here in concrete C syntax for the sake of readability. Note that scaling (i.e. multiplication of the offset by the size of target type) is explicit in CFG expressions (but hidden here, when incrementing pointers). In this program, the two global variables \((\text{arrays} S\) and \(T)\), are initialized at the beginning of the main function. This program builds a pointer \(p\) to some element of the arrays \(S\) and \(T\), depending on the values of \(b1\) and \(b2\) that can be seen as input in this example (as suggested by the assignment to \(\text{any\_bool()}\)). It then uses the value of this element as an offset in array \(S\) (pointer \(x\)) and returns the value referenced by \(x\).

The picture in Figure 8 schematically shows a possible concrete state when reaching line 7. Variables \(S\) and \(T\) are allocated respectively to blocks 100 and 101. Local variables are stored in registers, and registers \(b1\) and \(b2\) contain respectively integers 0 and 1. Therefore, after the ++\(p\) statement, \(p\) points to offset 4 in block 101. We will focus on line 7. When analyzing this line, the abstract memory functor is asked to model the assignment to variable \(x\) of the value resulting from the evaluation of expression \(x + \star p\). This query is expressed using CFG expressions, which cannot be directly given to the numerical domain. One difficulty in answering this query lies in the load expression; the memory functor needs to predict what are the locations that may be targeted by pointer \(p\).

To do so, the abstract memory functor embeds a generic numerical domain (as shown on the right of Figure 9) and a points-to domain (at the top). Queries that are sent to the memory functor flow to (plain arrows) the points-to and numerical domains. On the way, they are converted to load-free queries for the points-to domain (using load-free expressions of type \(\text{pespr}\) defined in Figure 10) and numerical queries for the numerical domain (using numerical expressions of type \(\text{nexpr}\), previously defined in Figure 7). These conversions use (dashed arrows) the information provided by these domains: the numerical conversion uses the points-to information to distinguish integers from pointers and the load-elimination uses the both points-to and numerical information to resolve loaded addresses.

3.1 Abstract Memory Cells

Abstract memory cells are the first component of the abstract memory functor. In CFG, when a program accesses an array element or a structure field (as in \(S[1] = 1\)), the variable (here \(S\)) is not fully involved but only a chunk of it. Therefore, we introduce a notion of abstract cell (and the corresponding data-type \(\text{acell}\)) to represent

---

3 The Coq keyword \texttt{Fixpoint} defines a recursive function.

4 In Coq, predicates are functions returning values of type \texttt{Prop}, the type of truth types True and False.
locations that are accessed by the program. Such an abstract cell is either a chunk \( \kappa \) at some offset \( i \) in a global variable \( s \) or a register \( x \).

Abstract cells (type \( \text{acell} \)): \( a ::= \text{ACglobal} \, s \, \kappa \, i \mid \text{ACreg} \, x \)

The memory chunk \( \kappa \) in the global-variable location describes how this cell is accessed (in particular, it gives the size of the cell and how to interpret its contents). This memory chunk simplifies the view of the memory. Otherwise when reading a cell, we would have to decode its contents. In contrast, we chose to perform the trans-coding on stores.

There is no such meta-data for the register location, since the contents of registers are accessed directly and-as (i.e., without transformation nor reinterpretation of the data).

For instance, the abstract memory cells involved in the analysis of our example program are: \( \text{ACglobal} \, S \, \text{Mint32} \, 0 \) and \( \text{ACglobal} \, T \, \text{Mint32} \, 0 \), that represent the two elements of array \( S \); \( \text{ACglobal} \, T \, \text{Mint32} \, 0 \) and \( \text{ACglobal} \, T \, \text{Mint32} \, 0 \), that represent the two elements of array \( T \); \( \text{ACreg} \, b_1 \), \( \text{ACreg} \, b_2 \), \( \text{ACreg} \, p \), and \( \text{ACreg} \, x \), that represent the four register variables of this program.

These abstract cells are used as variables of the numerical and points-to domains; they operate as if the program was manipulating abstract cells rather than physical memory locations, and compute invariants about the contents of these cells. The role of the abstract memory functor is to make this illusion correct, by converting queries about CFG expressions into queries about abstract cells.

Notice that abstract cells may overlap: two abstract cells may refer to the same address with different chunks, or to overlapping chunks of the memory. For instance, if the program wrote a 64-bit integer at the address of array \( T \), the cell \( \text{ACglobal} \, T \, \text{Mint64} \, 0 \) would be used to describe this access; and this cell overlaps with the two other cells about \( T \). Most often, the abstract domains will not compute invariants about overlapping cells. Would this happen (and it will when the analyzed program uses unions, as will be described in section 3.1), the conjunction of these invariants would apply to the concrete part of memory they represent. Therefore, we need to take care of overlapping cells on store statements (all possibly written cells have to be updated) rather than on load expressions (reading only one of the read cells always returns a sound invariant).

3.2 Pointer Expressions

Pointer expressions are the second component of the memory model. They correspond to an intermediate representation between CFG expressions (see Figure 5) and numerical expressions (see Figure 7). Indeed, these are CFG expressions without loads or equivalently numerical expressions with pointers. They are defined in Figure 10 and use the same constants and the same operators as plain CFG expressions. They are parameterized by the type of the variables they may contain: the type of a pointer expression \( p \) is \( \text{pexpr} \, \text{var} \), where \( \text{var} \) denotes the type of the variables. For instance, in this section, we use pointer expressions of type \( \text{pexpr} \, \text{acell} \).

The concrete semantics of these expressions is very similar to the one of CFG expressions, except that it does not depend on a memory (for loads) but only on the permissions (for pointer comparisons). It is defined in Figure 10.

3.3 Points-to Abstract Domain

In order to precisely understand CFG expressions involving loads and pointers, the memory functor needs to 1) distinguish cells holding integers from cells holding pointers, and 2) predict the set of blocks a pointer may target. We thus attach to each cell a type (integer (Int), pointer (Ptr) or unknown (All)) and, if it is a pointer, a finite set of blocks.

This set is either a subset of the finite sets of all global variables, or the special All value that denotes any block and is used, for instance, to abstract pointers to stack-allocated variables. We finally get a semi-lattice (of type pointsto+T) that can be pictured as Figure 11. The \( t+T \) notation (for any type \( t \)) refers to the type \( t \) extended with an extra All element; the other values are tagged with the constructor Just that we often omit. In the example program, when reaching line 7 the points-to domain state is as depicted below.

\[
\begin{align*}
\text{ACglobal} \, S \, \text{Mint32} \, 0 & \mapsto \text{Int} ; \\
\text{ACglobal} \, T \, \text{Mint32} \, 0 & \mapsto \text{Int} ; \\
\text{ACreg} \, b_1 & \mapsto \text{Int} ; \\
\text{ACreg} \, p & \mapsto \text{Ptr}((\text{S} ; \text{T})) ; \\
\text{ACreg} \, x & \mapsto \text{Ptr}((\text{S}))
\end{align*}
\]

This domain features in particular a forward evaluation function eval-ptr, of the following type (where Map[\(A, B\)] denotes the type of finite maps from type \(A\) to type \(B\)),

\[
\text{eval-ptr} : \text{Map[\(\text{acell}, \text{pointsto}\)}] \rightarrow \text{pexpr} \, \text{acell} \rightarrow \text{pointsto}+\text{T}.
\]

Given an abstract state \( \rho \) mapping each cell to its points-to approximation, and a pointer expression \( p \) (eval-ptr \( \rho \) \( p \)) computes the points-to information corresponding to the result of the evaluation of this expression.

The type of the abstract memory functor is thus a pair made of a points-to information, and a point in the numerical domain.

Definition \( t : \text{Type} := (\text{Map[\(\text{acell}, \text{pointsto}\)}] \times \text{num}) \).

3.4 Conversion

To analyze a statement as the one on line 7 in Figure 8 the abstract memory functor needs to translate the CFG expression \( x + *p \) into possibly many numerical expressions to hand them over to the numerical domain. This translation is decomposed into two interleaved tasks (illustrated in Figure 9) that we define in this section: elimination of the memory loads, and elimination of the pointers.

Load Elimination The first phase of the conversion, implemented by the function convert, is detailed at the top of Figure 12. It produces a set of pointer expressions (denoted by (Just \( pes \))) out of one CFG expression (see Figure 3) or gives up and returns All. It traverses an
expression, converts its sub-expressions, and combines all possible results (as in a regular set monad). When encountering an expression of the form load(x,e), the abstract memory functor computes an over-approximation of the set of cells that may be designated by expression e.

To compute this set of cells, the loads from sub-expression e are recursively eliminated; this yields a set of expressions without loads. Then each of these expressions is given to: 1) the points-to domain which computes a set of blocks (i.e., names of global variables); and 2) the numerical domain (after pointer-elimination) that computes a concrete set of offsets. The Cartesian product of these two sets produces a concrete set of cells. This logic is implemented by the deref-pexpr function. It is defined at the bottom of convert, and makes use of the function nconvert to eliminate pointer expressions.

**Pointer Elimination** The expressions resulting from the load elimination may still contain pointers: pointer constants or pointer arithmetic. The function nconvert produces a numerical expression (of type nexpr) from a points-to information and a pointer expression of type (pexpr acell). Its code is described at the bottom of Figure 12.

This pointer information pt is also a parameter of the previously defined functions convert and deref-pexpr.

To convert a pointer expression, it is recursively traversed and its structure is mostly kept (e.g., a negation of an integer expression is mapped to the negation of the converted sub-expression). When converting constants (function nconvert), integers are kept, and pointer values are replaced by their offsets. Indeed, the block identifier of the pointer value was taken into account by the deref-pexpr function during the previous load elimination. Non-trivial cases occur when converting boolean expressions (e.g., (Puop boolval pe)): all (non-null) pointers are true. For instance, the boolval unary operator casts its argument into a boolean (i.e., integer zero or one). To correctly convert such an expression, we need to distinguish whether the argument is an integer or a pointer (this information is available thanks to the points-to domain, through the pt argument):

- if it is an integer, the conversion goes on as usual;
- if it is a pointer, the whole expression is mapped to the constant expression that evaluates to integer one; indeed, all (non-null) pointers are true;
- if its type is not known at analysis time, the result is soundly over-approximated by the expression that may evaluate to integers zero and one.

In our example, the CFG expression to convert is (in a more readable syntax): \(x + 4 \times \text{load Mint32 p}\). The load to eliminate is the dereferencing of p.

1. This sub-expression is converted to a set of pointer expressions, namely to the singleton (Pv (ACreg p)).
2. All expressions in this set (there is only one) are evaluated (in parallel) in the points-to domain and in the numerical one. In the points-to domain, cell (ACreg p) is mapped to the value Ptr\{\{S,T\}\} which represents all pointers to some cell within the blocks of global variables S and T. Evaluation in the numerical domain may result in the precise set \{0;4\}. Such precision is achieved using a product of interval and congruence domains.
3. The Cartesian product of these sets yields a set of four abstract cells: (ACglobal S Mint32 0); (ACglobal S Mint32 2); (ACglobal T Mint32 0); and (ACglobal T Mint32 4).

Then, the conversion proceeds and builds an expression (Pv (ACglobal \(x \times \text{Mint32 0}\))) + \(4 \times (\text{Pv c})\), for each cell c from the above set.

### 3.5 Abstract Transformers

Conversion enables the implementation of the abstract transformers that model CFG statements: forget, assume, assign, and store. We describe their design in this section.

#### 3.5.1 Forget

The forget operator over-approximates any statement that may overwrite a temporary in an unknown way (e.g., a call to an input function such as getc in C). During the analysis, we need a more general operator that enables us to forget the content of any cell. We therefore provide a forget-cells function that forgets anything about a set of cells. It removes them from the points-to map and calls the corresponding forget operator of the numerical domain. This operation involves no conversion.

#### 3.5.2 Assume

The purpose of the (assume e ab) transformer is to refine an abstract state ab to take into account the fact that expression e evaluated to a true value. It is defined as follows.

\[
\text{assume e ab} := \top \cup \{ \text{let pt := pt-assume ab pe in} \}
\]

The expression e is first prefixed by the operator boolval to ensure that the result is actually cast to a boolean. The resulting expression is then converted using convert to a pointer expression in order to eliminate the loads in the expression. This conversion may fail, in which case nothing new can be learned. Otherwise, the conversion returns a set of pointer expressions pe.
When the analysis reaches line 5, it knows that the guard \(0 < *p\) is first converted to a set of pointer expressions. Then, from all cells in a given set but overlap with some of them, that computes the set of all cells that are distinct overlapping-cells about any overlapping cell. To do so we rely on an auxiliary function model this property, we conservatively forget anything that is known cells.

\[
\text{assign-cells} \quad (\kappa : \text{setacell}) \quad (e : \text{expr}) \quad : (\kappa, (\text{pt}, \text{nm}) : \text{t}) \quad : \quad \text{t+} := \text{let pes := convert pt nm e in let ab' := set-map-reduce dst (λ c, let (ty, nm') := set-map-reduce pes (λ pe, \text{eval-ptr pt pe, assign c (nconvert pt (ensure-cast-for-chunk κ pe)) nm})) in (map-assign pt c ty, nm')) in forget-cells (overlapping-cells dst) ab'}.
\]

### Figure 13. Test case

All these expressions are then given to the pt-assume transformer of the points-to domain and to the assume transformer of the underlying numerical domain (recall that nconvert casts a pointer expression into the corresponding numerical expression). The pt-assume operator is in charge of refining the points-to information. It cannot do anything useful yet. It will be discussed in more detail later (see section \([5.2]\)). Finally, all resulting abstract states are joined together.

Consider as example the analysis of the program shown on Figure [13] using a relational numerical domain (e.g., polyhedra) that is able to infer, after the first if, that \(z\) is larger than both \(x\) and \(y\). When the analysis reaches line [5] it knows that the guard \(0 < *p\) is true. However, pointer \(p\) targets either \(x\) or \(y\), so conversion of the guard yields two numerical expressions: one about \(x\), the other about \(y\). Since in both cases the numerical domain is able to infer that \(z\) is larger than one, in spite of uncertainty about the target of \(p\), the analysis is able to prove the assertion on line [5].

This implementation is not the most precise one. To illustrate the loss of precision, consider the following C snippet.

```c
int main(void) {
int x = 0, y = 1;
int *p = any_bool() ? &x : &y;
if (*p) { /* p points to y */ }
else { /* p points to x */ }
return 0;
}
```

### Figure 14. Updating cells: assign and store

for each destination cell \(c\) and each pointer expression \(pe\) resulting from the conversion, the assignment of this expression to that cell is performed, in parallel, in the points-to domain and in the numerical one. All these parallel assignments are then joined together. This results in a strong update if the destination set is a singleton; a weak update otherwise. This parallel evaluation followed by a join of all results is implemented by the set-map-reduce combinator. Finally, all overlapping cells are erased.

As we perform trans-coding on stores, the pointer expressions are changed to add an explicit cast (ensure-cast-for-chunk) so that the numerical domain knows that the value has to be trans-coded. For instance, when adding a cast to 8-bit unsigned integer, the numerical domain truncates the abstract value to the interval \([0; 255]\).

### 4. Soundness

We have seen so far the implementation of the abstract memory functor. We now move on its soundness proof. We first introduce an intermediate specification of the concrete memory in section \([4.1]\) then discuss the concretization relation of the points-to domain in section \([4.2]\) and of the whole abstract memory functor in Section \([4.3]\). Finally we present the central lemmas of the soundness proof in section \([4.4]\).

#### 4.1 Functional Memory

The (concrete) memory model of CompCert has been defined to enable the specification of various programming language semantics, and the verification of the compiler. Unfortunately, it is not very convenient for the verification of the abstract domain. Therefore, we introduce an intermediate specification of the concrete memory which is structurally closer to the memory abstract domain (see Figure [15]). This concrete memory is made of cells. Each such cell represents a location from which some data can be fetched. A memory is then simply a total function from cells to values. The value \(vundef\) represents uninitialized data and thus enables us to see the memory as a total function.

We distinguish between two kinds of cells (type ccell): cells that are in memory (global variables and local variables whose address is taken); and temporary variables (also known as registers). The similarity with the abstract cells is intentional.

To be able to correctly define the semantics of the CFG expressions in this model, we need to keep track of a little bit more information. Indeed, pointer comparison in CompCert behaves differently whether its arguments are valid pointers or not. The validity of pointers is defined in term of permissions (see Figure [9]) attached to each memory address. Therefore, the memory is defined as the following record type. The first field, namely f-of-fmem, is a coercion (as denoted by \(>:\)); this means that a value of type fmem can
The soundness of the points-to domain is established against a value \( v_{undef} \). In particular, it is the only abstract value that represents the bogus value \( 0 \). This invariant enables us to prove that some value cannot be \( v_{undef} \). This is particularly useful for proving progress (as it is used as a function of type \( c : ccell \rightarrow \text{val} \). Therefore, reading the content of a cell \( c \) in a memory \( f \) amounts to the function application \( f(c) \). The second field called \( perm \) expresses the validity of pointers.

Writing to the memory is a little bit more intricate. Indeed, memory cells can overlap. We introduce a notion of disjointedness (i.e., non-overlap) with the predicates called disjoint-ranges and disjoint-cells. Overlapping cells are either the same temporary or overlapping chunks of the same memory block. Then a store can be specified by the relation called \( \text{fmem-update} \) between memories. It says that the memory resulting from the store, \( f' \) holds the new value \( v \) at the target cell \( c' \), and agrees with the original memory \( f \) for all cells disjoint from \( c \).

The abstract transformers of the memory functor can then be specified against this concrete view of the memory, as shown in Figure 15. The forget-sound property reads as follows. The concretization of the result of \( \text{forget} \times a b \) (contain at least) all concrete memories obtained after updating an initial memory \( f \) at the target cell with any value \( v \). The specification for the assign transformer is very similar, except that the value \( v \) is taken among the possible results of the evaluation of the CPG expression \( e \).

Figure 15. Specification of concrete memory

is done, for instance, in the Verasco static analyzer [15] and to precisely analyze programs with type-punning (see section 5.1).

4.3 Concretization Relation

The relational numerical domain comes with a concretization to functions from (abstract) cells to numerical values (machine integers). The concretization relation is defined in Figure 17. Using the nocompat relation, it can be given a concretization to functions from cells to values (including pointers). The type domain, that maps abstract cells to an abstraction of their types, also concretizes to a function from abstract cells to values. We can then define the concretization relation (called \( \text{pre-gamma} \)) for the abstract memory functor, where concrete values are functions from abstract cells to values.

The set \((\text{pre-gamma}(pt,nm))\) is the intersection of the concretization \( \gamma(pt) \) of the points-to map and of the set of all memories related (by the point-wise lifting of the nocompat relation) to some function \( v \) in the concretization \( \gamma(nm) \) of the numerical abstraction.

In order to relate an abstract memory to a concrete memory, we still have to relate abstract cells to concrete cells. This is done through an allocation partial function, that maps abstract cells to concrete cells. One such allocation function is given in Figure 17 and called \( s_0 \). It is defined using a monadic style (hence the \( \text{do_opt} \) notation that simplifies the syntax of propagating None values).

This function is not so far from the identity function (modulo the correspondence between blocks identifiers and variable names): when we will introduce summarized cells (in section 5.3), this function will be generalized.

Given such an allocation function, we can relate memories (mem-rel relation) as follows: related cells are mapped to equal values (and cells that are related to nothing have unconstrained

Footnote 3: This does not specify the value of overlapping cells, and is therefore not sufficient for a precise analysis of programs performing type-punning; see section 5.3 for a discussion.
Definition ncompat (v : val) (j : int) : Prop :=
match v with
| vint i | vptr _ i ⇒ i = j
| vfloat _ | vundef ⇒ True
end.

Definition pre-gamma ((pt, nm): t) (f:acell → val) gamma-op t (acell → val) :=
f ∈ γ(pt) ∧
∃ (ν: acell → int), ν ∈ γ(nm) ∧ ∀ c, ncompat (f c) (ν c).

Definition allocation : Type := acell → option ccell.

Definition δ₀ (ac: acell): allocation :=
match ac with
| ACglobal g k o ⇒
do_opt b ← symbol ge g;
Some(CCmem b k o)
| ACreg i ⇒ Some(CCReg i)
end.

Definition mem-rel (δ: allocation) (m: fmem) (ρ: acell → val) : Prop :=
∀ c a, δ(a) = Some c ⇒ ρ(a).

Instance gamma (ab:t) (m:fmem) : gamma-op t fmem :=
(mem-rel δ₀ m) ⊆ (pre-gamma ab).

Figure 17. Concretization relation

value). Notice that, given an allocation function δ and a memory m, the set (mem-rel δ m) is not empty: there is in it at least the function m o δ', where δ' allocates every unallocated cell to a dummy one (e.g., δ' a := match δ a with Some c ⇒ c | None ⇒ CCReg 1 end, where (CCReg 1) denotes an arbitrary valid register name).

Then we can define the concretization relation gamma from the abstract memory to concrete memories. A memory m is in the concretization of an abstract memory ab whenever all functions in the pre-gamma concretization of ab are related to m. By using a for-all quantification (i.e., set inclusion) when a there-exists (i.e., non-emptiness of the intersection) would be plausible as well, we insist on the fact that we do not care about the value of the non-allocated cells, and that the abstract memory functor should not compute anything about these cells.

4.4 Key Lemmas

So as to depict how the soundness proof of the abstract memory functor is carried on, this section highlights the most important lemmas; their proofs are done in Coq and not described in this paper.

4.4.1 Conversion

There are two tasks during conversion: load elimination and pointer elimination. Given an expression e, the first one produces (in case of success) a set pes of expressions such that, collectively, the resulting expressions may evaluate to all possible values ν of the original expression e. This property is expressed by the following lemma.

Lemma convert-sound (env:env) (m: fmem) (ρ: acell → val) (ab: t) (e: expr) (pes: P (pexpr acell)) :
mem-rel δ₀ m ρ →
m ∈ γ(ab) →
convert ab e = Just pes →
∀ v, eval-expr env m e v → v ∈ ∪ pes ∈ pes (peval ρ pe).

The second task produces, for each pointer-expression, a purely numerical expression. Numerical expressions evaluate to numerical values (machine integers) whereas pointer expressions may also evaluate to pointers. Therefore, the soundness property of this second task states that for each possible value ν of the original expression pe, the resulting expression evaluates to some number n that is compatible with value ν. The compatibility relation ncomp was previously defined in Figure[7] It relates in particular numbers to themselves and pointers to their offsets.

Lemma nconvert-sound : ∀ (pt : Map[acell, pointsto]) (ρ : acell → val),
(∀ x : acell, ρ(x) ∈ γ (pt[x])) →
(∀ (ν : acell → int), ∃ ncompat (ρ x) (ν x)) →
(∀ (pel: pexpr acell) (v : val),
v ∈ peval ρ pel →
∀ n : int, n ∈ (eval ν (nconvert pt pel) ∩ ncompat v).

4.4.2 Assign

The soundness lemma for the (assign-cells dst e ab) operation reads as follows. For every concrete memory f and value ν the expression e may evaluate to in f, for every destination cell a (related to the concrete cell c through δ₀), the concrete memories obtained by updating f at c with value (load_result κ ν) are in the concretization of the resulting abstract state. The stored value is trans-coded according to chunk κ.

Lemma assign-cells-sound (κ: option chunk) (dst: set acell) (e: expr) (ab: t) :
∀ m, m ∈ γ ab →
∀ ν, eval-expr m e ν →
∀ a, a ∈ dst →
∀ c, δ₀ a = Some c →
fmem-update c (load_result κ ν) m ⊆ γ(assign-cells κ dst e ab).

5. Analysis Extensions

The abstract memory functor presented so far can be improved in both precision and performance. We describe in this section three enhancements that we have integrated to our development and proved correct: a precise analysis of programs using unions, an acute handling of null pointers, and a summarization of arrays.

5.1 Unions and Type Punning

Some low-level programs perform so-called type punning. This means accessing some data of a given type (say a 32-bit integer) as if it was of a different type (say an array of four 8-bit integers). This is usually used to access the bit-level representation of some data. For instance, the program on the top of Figure[18] is a C snippet that discovers at run-time the endianness of the architecture. Such
puns are only loosely specified by the C standard. Most of them are however precisely defined in the CompCert semantics.

There are several kinds of puns. They all involve reading some data with a different chunk than the one that was used to write it. We will focus on the following two cases.

- Reading at the same address with a chunk of the same size. This covers changes in signedness and manipulations of the bit-level representation of float[6] The first are similar to casts (with the cast\text{unsigned} CFG operator, for instance) whereas the last are no standard C operations and are not handled by the numerical abstract domains.

- Reading one byte of a multi-byte data, as in the implementation of the memcpy function given at the bottom of Figure 18.

In case of type punning, a cell whose content has to be read is not bound in the abstract domain. Indeed, when a cell is written, the (abstract) contents of all overlapping cells are forgotten. To precisely approximate the read value, the cell needs to be realized[22], that is, bound to some non-trivial abstract value derived from the values of overlapping cells.

We expect the following property from the \text{(realize c ab)} function, meant to realize the cell c in the abstract state ab: all memories in the concretization of some abstract value ab before realization, are in the concretization after realization.

\text{Lemma realize-sound (c: acell) (ab: t) : γ(ab) ⊆ γ(realize c ab).}

However, this is hardly provable, since in the functional view of the concrete memory, the values of overlapping cells are not constrained. One way to address this issue is to further constrain the fmem type, using the following property.

\text{Definition pun-\text{u8} (f: cell \rightarrow val) := \forall b ofs κ, (align-chunk κ | ofs) → ∃ bytes, f (CCmem b κ ofs) = decode-val \text{bytes} ∧ length \text{bytes} = size_chunk κ ∧ \forall i (LT: i < length \text{bytes}, f (CCmem b Mint\text{sunsigned} (ofs + i)) = decode-val \text{Mint\text{sunsigned}} (get i \text{bytes LT :: nil}).}

It states that for every value read through some chunk κ at a properly aligned offset ofs, there is a sequence called bytes of bytes from which this value is decoded, such that reading in memory at the same address the \text{ith} Mint\text{sunsigned} chunk yields the decoding of the \text{ith} byte in the sequence. In this property, the get function takes as argument a proof LT that there are at least \text{i} elements in the list, to be sure to be able to return a meaningful value.

This property is added as invariant of the type fmem. This enables us to prove the soundness of realization functions that bind new cells in the abstract memory (i.e., in both points-to and numerical domains) using information derived from what is known about overlapping cells. Such functions are called optimistically when expressions dereference unbound cells, so as to try to bind them to some non-trivial value.

Realization needs to take place during conversion. Here is an example of code where the variable x is first written as a 32-bit integer, then read as an 8-bit unsigned integer.

```c
1. int x;
2. int main(void) {
3.   int *p = 0;
4.   x = 0;
5.   if (any_bool()) p = &x;
6.   if (p != 0)
7.     *p = 1;
8.   int z = x;
9.   return z;
10. }
```

\text{Figure 19. C program checking for null pointers}

```c
1. int t[2] = { 1, 2, 3, 4 };
2. union { int i; uint8_t c[4]; } x;
3. x.i = 1;
4. int v = t[x.c[0]];
```

\text{8Fast computations of approximations of inverse square roots are a notable example of floating-point operations performed on the bit-level representation of floats.}

To understand what cells are targeted by array accesses, we need to numerically evaluate the subscript expression. This requires the 8-bit cell to be realized during conversion. The conversion function is therefore adapted to manipulate the abstract state as in the state monad, rather than as an input value.

5.2 \text{The Case of Null Pointers}

The analyzer described so far is not able to prove that, after a test ensuring that a pointer is not null, the said pointer is actually not null. Consider for instance the program of Figure 19 where pointer p is either null or points to variable x, depending on some unknown input. The assignment on line 7 is guarded by the condition of line 6 so that it is safe and it updates variable x.

The problem comes from the fact that zero and null pointers are the same value (in CompCert as in standard C99 [14, section 6.3.2.3]). Indeed, in our example, variable p may have two distinct types: either integer or pointer. In other circumstances, the memory functor soundly ignores variables whose type cannot be inferred: a safe program is not expected to store values of different types in a single variable.

The null pointer is an exception to this typing rule: a pointer can be used as an integer without explicit cast, provided it is null (and symmetrically). We therefore refine two components of the memory functor, as we describe below.

\text{Richer Types} Currently, the points-to domain loses precision when joining the abstraction for a null value (an integer) and a non-null pointer. This domain is therefore refined to still precisely handle maybe-null pointers. The lattice of abstract values is given in Figure 20. Values with pointer-type are also associated to a points-to set, as before (not shown on the picture).

\text{Assume} When analyzing a guard such as (p != 0), if p is known to be a maybe-null pointer, its abstract type has to be refined to the non-null pointer with the same points-to set.

More generally, we introduce a backward evaluation function in the points-to domain, which computes for each expression and abstract value representing the result of the evaluation of this expression in some abstract state, a more precise abstraction for the evaluation state. This backward evaluation function corresponds to the implementation of the pt-assume function introduced in section 3.5.2.
5.3 Summarized Cells

Summarized cells are abstract memory cells which represent, at once, several concrete memory cells. Their use can increase the efficiency of the analyzer (e.g., smaller memory footprint, faster analysis) and its expressiveness (dynamically allocated memory, including malloc, variable-length arrays, and recursion).

For the sake of simplicity, we consider the following setting: only global variables can be summarized, and whether a variable is summarized or not is fixed throughout the analysis. The analysis is therefore parameterized by a global function \( \text{is\-strong} \) that statically classifies global variables into strong ones and summarized ones.

Summarization only makes sense for indexed collections of elements of a same type, as arrays. A summary represents all elements of a given collection at once. Each summary has a size, which corresponds to the size of the elements of the collection.

Cells which belong to a summarized block may represent several concrete cells in the concrete memory. This is made explicit by the allocation functions which choose, for each summarized variable the element that is represented by the summary. So as to ensure that the abstract domain soundly approximates any possible choice, the concretization relation is updated to universally quantify on allocation functions. Since not all functions from abstract to concrete cells make sense as allocation functions, we constrain them to satisfy some properties gathered under the name allocation-spec; for instance, an allocation function should be the identity on registers and be injective.

\[
\text{instance gamma : gamma-op t fmem} \overset{\lambda \text{ab m}}{\rightarrow} \forall \delta, \delta \in \text{allocation-spec \rightarrow mem-rel \delta m} \subseteq \text{pre-gamma(ab)}.
\]

Particular care is required when dealing with summarized cells: as they may represent several concrete locations, only weak updates can be performed on them. For instance, the store \( t[0] = 4 \) in which array \( t \) is summarized must be understood as “some element of \( t \) holds 4” rather than “all elements of \( t \) hold 4”. Similarly, an assume \( 0 < t[0] \) should not be interpreted as “all elements of \( t \) are positive”. The embedded relational numerical domain may not have been designed to deal with such summary variables. Gopan et al. describe an extended interface of numerical domains to this purpose [11]. The simpler approach that we use here is to replace occurrences of summarized cells by an over-approximating interval in queries to the numerical domain. To this end, numerical and pointer expression languages are extended with range constants which represent any integer in the given range or any pointer to the given block with any offset in the given range.

6. Experimental Evaluation

We have described a formally verified abstract memory functor for C value analysis. In itself this functor represents 5000 lines of Coq development that lead to the extraction of 2400 lines of OCaml. This functor has been deployed in a CFG value analyzer (12000 lines of Coq without the abstract memory functor). The whole analyzer is connected to the CompCert compiler (100k lines of Coq).

We report in table [4] on some performances of the Verasco analyzer (version 1.3), that reuses our abstract memory functor, except the backwards evaluation in the points-to domain (defined in section [5.2]) and the array summarization (defined in section [5.3]). On these examples, Verasco was able to prove the absence of run-time errors. For each analyzed program, the table gives its name, its size expressed in number of C#minor instructions, and the time (in seconds) needed by Verasco to analyze it and prove it free of undefined behaviors.

Each block in the table gathers programs from different sources. The first block of rows gathers programs adapted from the CompCert benchmarks; they implement cryptographic primitives or numerical computations. The second block gathers tests from the NaCl cryptographic library. Then there are three interesting programs that have been randomly generated by the Csmith tool [26]. The two following programs check the satisfiability of Boolean formulas with respectively 20\(^2\) and 50\(^2\) variables. The last block lists test programs of the PolarSSL cryptographic library.

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (instr.)</th>
<th>Analysis time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>aes</td>
<td>960</td>
<td>6.0</td>
</tr>
<tr>
<td>almabench</td>
<td>217</td>
<td>5.0</td>
</tr>
<tr>
<td>fft</td>
<td>203</td>
<td>0.02</td>
</tr>
<tr>
<td>fttw</td>
<td>88</td>
<td>6.5</td>
</tr>
<tr>
<td>integr</td>
<td>49</td>
<td>0.02</td>
</tr>
<tr>
<td>nboby</td>
<td>117</td>
<td>10</td>
</tr>
<tr>
<td>sha3</td>
<td>287</td>
<td>121</td>
</tr>
<tr>
<td>siphash24</td>
<td>272</td>
<td>0.2</td>
</tr>
<tr>
<td>core1</td>
<td>154</td>
<td>0.04</td>
</tr>
<tr>
<td>core2</td>
<td>142</td>
<td>0.04</td>
</tr>
<tr>
<td>core4</td>
<td>198</td>
<td>0.05</td>
</tr>
<tr>
<td>hash</td>
<td>457</td>
<td>1.6</td>
</tr>
<tr>
<td>hash</td>
<td>523</td>
<td>2.2</td>
</tr>
<tr>
<td>scalarmult</td>
<td>961</td>
<td>22</td>
</tr>
<tr>
<td>stream</td>
<td>920</td>
<td>507</td>
</tr>
<tr>
<td>random (1)</td>
<td>45</td>
<td>1.1</td>
</tr>
<tr>
<td>random (2)</td>
<td>3952</td>
<td>90</td>
</tr>
<tr>
<td>random (3)</td>
<td>4491</td>
<td>1.5</td>
</tr>
<tr>
<td>sat (20)</td>
<td>4642</td>
<td>0.8</td>
</tr>
<tr>
<td>sat (50)</td>
<td>66322</td>
<td>90</td>
</tr>
<tr>
<td>blowfish</td>
<td>177</td>
<td>31</td>
</tr>
<tr>
<td>des</td>
<td>229</td>
<td>2.05</td>
</tr>
<tr>
<td>donna</td>
<td>1217</td>
<td>484</td>
</tr>
<tr>
<td>rc4</td>
<td>93</td>
<td>4.05</td>
</tr>
<tr>
<td>salsa20</td>
<td>341</td>
<td>4.52</td>
</tr>
<tr>
<td>snow</td>
<td>871</td>
<td>2.44</td>
</tr>
<tr>
<td>tea</td>
<td>121</td>
<td>3.18</td>
</tr>
</tbody>
</table>

Table 1. Verasco analysis times

7. Related Work and Conclusion

Previous works on formal verification of static analysis were restricted to less advanced techniques for memory analysis. Klein and Nipkow verify a Java bytecode verifier where the heap is mainly abstracted by types [17]. Cachera et al. formally verify using Coq an inter-procedural class analysis for Java bytecode programs [5]. Leroy and Robert verify a points-to analysis in the CompCert framework [20]. Contrary to us, their points-to analysis does not interact with numerical abstraction and is thus far less precise.

CompCert itself [19] embeds verified datatflow analyses that support compiler optimizations. Most of them do not track memory values but CompCert has been recently upgraded with an unpublished value analysis, following previous collaboration between Leroy and us [15]. The corresponding memory abstraction is less fine grained than what we describe here and is not compatible with relational abstract numerical domains. The CFG analyzer that is described in [4] used a very coarse memory abstraction, without tracking memory contents. Other previous works on formal verification of abstract interpretation [2] provide pedagogical abstract interpreters for an imperative toy language and do not deal with memory.

The current memory functor can be extended in at least two directions. First, we would like to track more precisely the content
of summarized cells. Summarized cells are only updated with weak updates now and we could get more precision with Gopan et al. technique [11]. This would require extensions on the interface of numerical domains. As far as we know, Astrée does not use this technique yet. Next we could improve our analysis on dynamically allocated memory. The current proof is limited to memory allocated during program initialization (global variables), following usual safety-critical constraints. We would like to lift this restriction but the analysis would then require to deal differently with array initialization [23].

Acknowledgments
This work is supported by Agence Nationale de la Recherche, grant ANR-11-INSE-003.

References