An Abstract Memory Functor for Verified C Static Analyzers

Sandrine Blazy
Université Rennes 1 – IRISA, France
sandrine.blazy@irisa.fr

Vincent Laporte
IMDEA Software Institute, Spain
vlaporte@imdea.org

David Pichardie
ENS Rennes – IRISA, France
david.pichardie@ens-rennes.fr

Abstract

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.

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

Keywords abstract interpretation, Coq proof assistant, points-to analysis, numerical analysis, low-level memory

1. Introduction

Static analysis by abstract interpretation [7] has been successfully used to analyze very large C programs [9] and to prove the absence of run-time errors of some class in them. This achievement is based on an advanced *value analysis* that tracks the set of values manipulated by a C program. The analysis faces several technical challenges: it must use coarse enough abstractions to scale to about hundred thousand program variables but also keep enough semantic precision to infer good enough program invariants. In this paper, we stress that efficiency and precision are not the only challenges there.

Indeed, soundness is specially difficult to achieve because the analysis finely tracks the memory-manipulating instructions of the C language, taking into account byte representations of numbers and pointer arithmetic, and also isolation and freshness guarantees of variables. Hence the semantic correctness of the static analysis itself is questionable. As shown by previous works in compiler [18] and static analyzer verification [3], such tools can be programmed and proved correct inside a proof assistant like Coq [25], with respect to the formal semantics of the transformed or analyzed programming language, here a variant of the ISO C 99 language, namely CompCert-C. We follow this methodology here and specially target the *abstract memory functor* of a C value analysis.

```
1:  #define N 256
2:
3:  struct node { struct node *next; int value; };
4:  struct node list[N];
5:
6:  int main(void) {
7:    int i = 0;
8:    struct node *p = 0;
9:    for ( i = 0 ; i < N - 1 ; ++i ) {
10:     list[i].next = &(list[i + 1]);
11:     list[i].value = 2048 - i;
12:   }
13:   i = 0;
14:   for ( p = &(list[0]) ; p != 0 ; p = p->next ) {
15:     int d = p->value;
16:     i = i + d;
17:   }
18:   return 0;
19: }
```

Figure 1. Example C program using a linked list

Value analysis is a well-known static analysis that computes for each variable its abstract value, e.g., an interval approximating its possible values. More generally, it infers *relations* between the possible values of the program variables. Abstract values are defined by numerical abstract domains that are one of the main components of a static analyzer. Many numerical abstract domains have been defined in the literature to track various kinds of information on variables [6, 8, 12, 13, 16, 21]. Moreover, static analysis of programs with pointers requires dedicated treatments, known as *points-to* analyses. Indeed, such an analysis needs to predict pointer targets in order to infer something about a value accessed through pointers. On the other hand, points-to analysis in presence of pointer arithmetic requires a value analysis [22].

In this paper, we present an abstract memory domain that is able to handle points-to information. This domain is a *functor*: it is parameterized by a numerical abstract domain. This abstract memory functor is integrated into a static analyzer handling programs such as the program of

Figure 1. This program builds a linked list (first for loop) within a zero-initialized array list and then computes the sum of all elements in the list (second for loop). When analyzing this program, the abstract memory functor automatically infers invariants like the following. Every time the execution reaches line 15, pointer p targets an element of the array list: its offset is within the bounds of the array and properly aligned to a node boundary. The computed invariant implies that this program is memory-safe.

All results presented in these papers have been mechanically verified using the Coq proof assistant. The complete Coq development is available online [1]. The paper makes the following contributions:

- It provides a verified abstract memory functor that is central to 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 current abstract memory functor but only briefly introduced it in a recent publication [15]. This paper goes into the details of the development and the proof, and presents some recent extensions. We recall in this section the design of the analyzer.

2.1 The CompCert Compiler and its CFG Language

The analyzer is built on top of the CompCert compiler [18]. This compiler from C to assembly is programmed, specified, and proved in Coq. It is structured in successive passes and several different intermediate languages. Every intermediate language comes with a formal semantics, so that the correctness of the compilation is stated as 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 over-

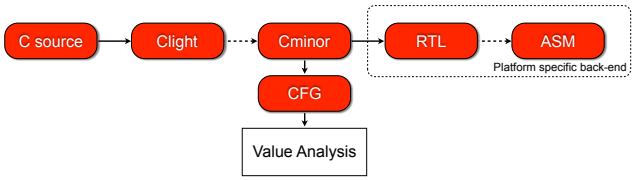

Figure 2. The CFG intermediate language inside the CompCert toolchain

loaded 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 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 κ 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 successors. A program is composed of a list of global variables (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. The evaluation (eval-expr $ge\ \rho\ m\ a\ v$) states that expression a evaluates to value v. It involves a global environment ge of type genv, 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[1]. Therefore, a pointer is a value (vptr b ofs) that represents the address of the byte at offset ofs in the block b. Other values stored in memory are integers, floats and the special value vundef representing the contents of uninitialized memory. An access to the content of a block, either to read (load operation) from it or to

[1] CFG does not handle 64-bit machine integers.

Constants:	$c ::= i$		integer constant
	$\mid f$		floating-point constant
	$\mid$ `addrsymbol`(id, i)		address of a symbol + an offset
	$\mid$ `addrstack`(i)		stack pointer + a given offset
Chunks:	$\kappa ::=$ `Mint8signed`		
	$\mid$ `Mint8unsigned`		8-bit integers
	$\mid$ `Mint16signed`		
	$\mid$ `Mint16unsigned`		16-bit integers
	$\mid$ `Mint32`		32-bit int. or pointers
	$\mid$ `Mfloat32`		
	$\mid$ `Mfloat64`		floats
Expressions:	$e ::= id$		variable identifier
	$\mid c$		constant
	$\mid a_1\ op_2\ e_2$		binary arithmetic op.
	$\mid op_1\ e$		unary arith. operation
	$\mid e_1\ op_2\ e_2$		binary arith. operation
	$\mid e_1 ?\ e_2 : e_3$		conditional expression
	$\mid$ `load`(κ, e)		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 op.
	$\mid$ `<<` $\mid$ `>>` $\mid$ `&` $\mid$ `|` $\mid$ `^`		bitwise operators
	$\mid$ `/`$_u$ $\mid$ `%`$_u$ $\mid$ `>>`$_u$		unsigned operators
	$\mid$ `cmp`$(\odot)$		int. signed comparisons
	$\mid$ `cmpu`$(\odot)$		integer unsigned comp.
Comparisons:	$\odot ::=$ `<` $\mid$ `<=` $\mid$ `>`		
	$\mid$ `>=` $\mid$ `==` $\mid$ `!=`		relational operators
Statements:	$\iota ::=$ `assign`(id, e, l)		assignment
	$\mid$ `store`(κ, e, e, l)		memory store
	$\mid$ `skip`(l)		no operation (go to l)
	$\mid$ `if`(e, l_{true}, l_{false})		if statement
	$\mid$ `call`$(sig, id^?, e, e^*, l)$		function call
	$\mid$ `return`$(e)^?$		function return
Control-flow graph:	$g ::= l \mapsto \iota$ finite map		
Functions:	$F ::= \{sig$		signature
	$; id^*$		parameters
	$; n;$		size of stack data block
	$; l$		label of first instruction
	$;\ g\ \}$		code
Programs:	$P ::= \{dcl^*$		global variables
	$; F^*$		functions,
	$;$ `main` $= id\}$		entry point

Figure 3. Abstract syntax of CFG

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 mapped to a permission (also known as access right). The permissions are, in increasing order:

None dangling pointer;

Values:	$v ::=$ `vint` i $\mid$ `vfloat` f $\mid$ `vptr` $b\ i$ $\mid$ `vundef`	
Permissions:	$\pi ::=$ `None` $\mid$ `NonEmpty` $\mid$ `Readable` $\mid$ `Writable` $\mid$ `Freeable`	
Memory states:	$m ::= b \mapsto (lo, hi, \pi, i \mapsto v)$	
Operations over values		
load-result(κ, v) = v'		Reflects the 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 $\in$ { 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 $[lo, hi[$
`Mem.free` $m\ b =$ `Some` m'		Free (invalidate) the block b
`Mem.load` $\kappa\ m\ b\ i =$ `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` $\kappa\ m\ b\ i\ v =$ `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` $m\ b\ i = \pi$		Permission of block b at offset i
`Mem.size_chunk` $\kappa = i$		Size (number of bytes) that κ holds
Global env.:	$ge ::= (id \mapsto b)$	map from global variables to block references
	$\times (b \mapsto F)$	and map from function references to function definitions
Local env.:	$\rho ::= id \mapsto b$	map from local variables to block references
Operations over global environments		
`symbol`$(ge, id) =$ `Some` b		Return the block b corresponding to the global var. or function name id
Evaluation of expressions		
eval-expr $ge\ \rho$ m a v		Expression a evaluates to value v.

Figure 4. Concrete memory model, semantics of expressions

NonEmpty valid pointer, but only comparisons are permitted (e.g., pointer to a function);

Readable loads are also permitted (e.g., constant static data);

Writable stores are also permitted (e.g., global variable);

Freeable full permissions, free is also permitted (e.g., stack allocated local variable).

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 [3]. The fixpoint iterator takes as input the code of a function and an abstract semantics of the CFG

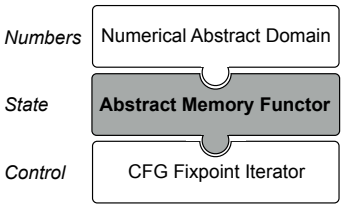

Figure 5. Modular architecture of the CFG value analyzer

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.

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 x 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 ab) operator assumes that an expression e evaluates to a non-zero value within all concrete states represented by ab. The (assign x e ab) operator models the assignment of the value of an expression e to a local variable x within an abstract state ab. The (store κ e_1 e_2 ab) operator models the assignment of the value of an expression e_2 to a memory location pointed by the value of expression e_1 with chunk κ within an abstract state ab. The (forget x ab) operator performs a non-deterministic assignment to the local variable x in the abstract state ab and can be used for instance to model input statements.

The t+⊥ 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.

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[2]. For instance, the (Store κ e_1 e_2) predicate transformer relates a set E of concrete states (ρ, m) — before executing a store instruction — to the set of concrete states (ρ, m') — after the store — that is obtained by storing in memory m a value v at address (vptr b i) with chunk κ; the address results from the evaluation of the left-hand-side expression e_1 in the environment ρ, and the value v results from the evaluation of the right-hand-side expression e_2 in the same environment ρ. Our implementation of the memory functor is proved sound with respect to these specifications; this leads to the theorems given at the bottom of Figure 6.

[2] In these definitions, notation $\rho[x \mapsto v]$ represents the environment that is identical to ρ excepted on variable x that is mapped to the value v.

Interface of the implemented memory functor

```
Class abs-mem-dom (t: Type) : Type := {
  mem-wl : weak-lattice t   (* abstract domain lattice structure *)
; range : t → ident → signedness → strided-interval+⊥
            (* consult the range of a local variable *)
; assume : expr → t → t+⊥
            (* assume an expression evaluates to non-zero value *)
; assign : ident → expr → t → t+⊥
            (* assignment to a local variable *)
; store : chunk → expr → expr → t → t+⊥
            (* assignment to a memory cell *)
; forget : ident → t → t+⊥
            (* non-deterministic assignment to a local variable *)
}.
```

Soundness of the memory functor

```
Definition Assume (q: expr) (E: 𝒫 (env×mem)) : 𝒫 (env×mem) :=
  { (ρ, m) | ∃ v, (ρ, m) ∈ E ∧ eval-expr ρ m q v ∧ bool-of-val v true }.

Definition Assign (x: positive) (q: expr) (E: 𝒫 (env×mem)) :
                  𝒫 (env×mem) :=
  { (ρ', m) | ∃ ρ v,
              (ρ, m) ∈ E ∧ eval-expr ρ m q v ∧ ρ' = ρ[ x ↦ v ] }.

Definition Store (κ: chunk)(e1 e2: expr)(E: 𝒫 (env×mem)) :
                 𝒫 (env×mem) :=
  { (ρ, m') | ∃ m b i v, (ρ, m) ∈ E ∧ eval-expr ρ m e1 (vptr b i) ∧
              eval-expr ρ m e2 v ∧ Mem.store κ m b i v = Some m' }.

Definition Forget (x: positive) (E: 𝒫 (env×mem)) : 𝒫 (env×mem) :=
  { (ρ', m) | ∃ ρ v, (ρ, m) ∈ E ∧ ρ' = ρ[ x ↦ v ] }.

Theorem assume-sound: ∀ e ab, Assume e (γ ab) ⊆ γ (assume e ab).
Theorem assign-sound: ∀ x e ab, Assign x e (γ ab) ⊆ γ (assign x e ab).
Theorem store-sound: ∀ κ e1 e2 ab,
                     Store κ e1 e2 (γ ab) ⊆ γ (store κ e1 e2 ab).
Theorem forget-sound: ∀ x ab, Forget x (γ ab) ⊆ γ (forget x ab).
```

Figure 6. Specification of the abstract memory functor

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].

We show in Figure 7 the signature of numerical domains, including the syntax and semantics of numerical expressions ne[3]. This semantics is defined by means of an evaluation function written ($neval_\rho$ ne), and parameterized by a numerical environment ρ which maps variables to machine integers. This function returns a set of machine integers as some constructions are not deterministic (e.g., the Nintunknown

[3] The Coq keyword Fixpoint defines a recursive function.

Numerical constants nc ::= Nintconst i | Nintunknown

Num. expressions (type nexpr)

ne ::= Nv v | Nc nc
| Nuop op_1 ne | Nbop op_2 ne ne
| Ncond ne ne ne

Evaluation of numerical expressions

Fixpoint neval_ρ (ne:nexpr): $\mathcal{P}$(int) := match ne with
| Nc (Nintconst i) ⇒ { i }
| Nc (Nintunknown) ⇒ (λn:int, True)
| Nv v ⇒ { ρ(v) }
| Nuop op_1 ne ⇒ $\bigcup_{\text{n} \in \text{neval}_\rho\ \text{ne}}$ (eval-nunop op_1 n)
| Nbop op_2 ne_1 ne_2 ⇒ $\bigcup_{\text{n} \in (\text{neval}_\rho\ \text{ne}_1)\times(\text{neval}_\rho\ \text{ne}_2)}$ (eval-nbop op_2 n)
| Ncond ne ne_1 ne_2 ⇒ $\bigcup_{\text{k} \in \text{neval}_\rho\ \text{ne}}$ neval_ρ(k=0 ? ne_2 : ne_1)
end.

Interface of numerical domains

```
Class ab-env (var num: Type) : Type := {
  wl: weak-lattice num
; gamma : gamma-op num (var → int)
; adom : adom num (var → int) wl gamma;
; range : nexpr var → num → signedness → strided-interval+⊥
; assume : nexpr var → num → num+⊥
; assign : var → nexpr var → num → num+⊥
}.

Theorem range-sound : ∀ ne ρ ab,
    ρ ∈ γ ab → neval_ρ ne ⊆ ints-in-range (range ne ab).
Theorem assign-sound: ∀ x ne ρ n ab,
    ρ ∈ γ ab → n ∈ neval_ρ ne → ρ[x ↦ n] ∈ γ (assign x ne ab).
Theorem assume-sound: ∀ ne ρ ab,
    ρ ∈ γ ab → 1 ∈ neval_ρ ne → ρ ∈ γ (assume ne ab).
```

Figure 7. Signature of numerical domains

integer constant) or have no semantics (e.g., the division by zero). Sets are represented by predicates[4]: for instance, (λn:int, True) is the set of all integers; notation {n} represents the singleton whose element is n; notation $\bigcup_{\text{a} \in \text{A}}$ (f a) represents the union of all sets (f a) for every a in the set A.

A relational numerical domain is specified by the record ab-env consisting of the following definitions. A *concretization relation* (type gamma-op) links abstract values to the concrete values they represent. The query operator range returns a strided interval; this is required to ensure the precision of the memory functor which relies on this operator to approximate pointer dereferences.

The abstract transformers range, assume and assign are similar to those of the abstract memory functor; however, they rely on numerical expressions of type nexpr. As there is neither loads nor pointers in numerical expressions, there is no need for a store operator. Moreover, since there is a constant Nintunknown denoting unknown numerical values, we do not need a forget operator for the numerical domains. The properties range-sound, assume-sound and assign-sound express the soundness of the abstract transformers. The numerical abstract domain is also parameterized by a type var of variables, that we will instantiate with a type acell of abstract memory cells, defined later in section 3.1.

[4] In Coq, predicates are functions returning values of type Prop, the type of truth types True and False.

```
1:  int S[2], T[2];
2:  int main(void) {
3:    int b1 = any_bool(), b2 = any_bool();
4:    int *x = S, *p = T;
5:    S[0] = T[1] = 0; S[1] = T[0] = 1;
6:    if (b1) p = S; if (b2) ++p;
7:    x = x + *p;
8:    return *x;
9:  }
```

Example of concrete state at line 7

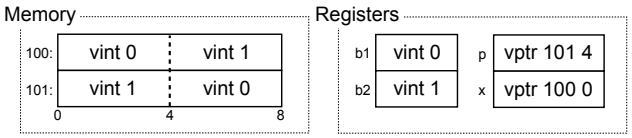

Figure 8. An example C program to be analyzed

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 · 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 · 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 (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 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

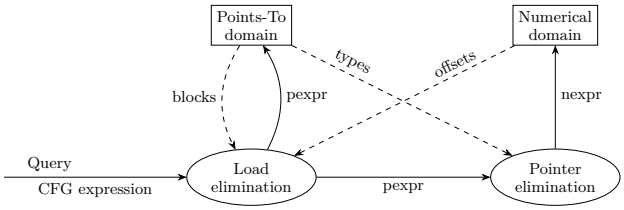

Figure 9. Sketch of the abstract memory functor

functor is asked to model the assignment to variable x of the value resulting from the evaluation of expression x + *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 pexpr defined in Figure 10) and numerical queries for the numerical domain (using numerical expressions of type 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 acell) to represent locations that are accessed by the program. Such an abstract cell is either a chunk κ at some offset i in a global variable s or a register x.

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

The memory chunk κ 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-is (i.e., without transformation nor reinterpretation of the data).

For instance, the abstract memory cells involved in the analysis of our example program are: (ACglobal S Mint32 0) and (ACglobal S Mint32 4), that represent the two elements of array S; (ACglobal T Mint32 0) and (ACglobal T Mint32 4), that represent the two elements of array T; (ACreg b1), (ACreg b2), (ACreg p), and (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

Pointer expressions (type pexpr var) pe ::= Pv v | Pc cst | Puop op_1 pe | Pbop op_2 pe pe | Pcond pe pe pe

Fixpoint peval (ρ:var → val) (pe:pexpr var): $\mathcal{P}$(val) := match pe with
- | Pv v ⇒ { ρ v }
- | Pc cst ⇒ eval-constant cst
- | Puop op$_1$ pe ⇒ $\bigcup_{n \in \text{peval } \rho \text{ pe}}$ (peval-unop op$_1$ n)
- | Pbop op$_2$ pe$_1$ pe$_2$ ⇒ $\bigcup_{n \in (\text{peval } \rho \text{ pe}_1 \times \text{peval } \rho \text{ pe}_2)}$ (peval-bop op$_2$ n)
- | Pcond b pe$_1$ pe$_2$ ⇒ $\bigcup_{k \in \text{peval } \rho \text{ b}}$ ($\bigcup_{k' \in \text{bool-of-val k}}$ (peval ρ (if k' then pe$_1$ else pe$_2$)))

end.

Figure 10. Semantics of pointer expressions

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 (ACglobal T 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 5.1), 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 3) 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 pe is pexpr var, where var denotes the type of variables. For instance, in this section, we use pointer expressions of type pexpr 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+$\top$) that can be pictured as Figure 11. The t+$\top$ notation (for any type t) refers to the type t extended with an extra All

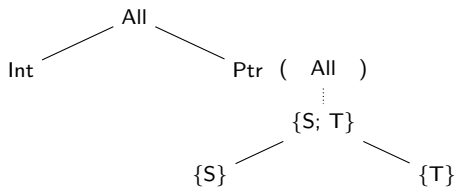

Figure 11. Semi-lattice of types and points-to values

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.

```
ACglobal S Mint32 0 ↦ Int;     ACglobal S Mint32 4 ↦ Int;
ACglobal T Mint32 0 ↦ Int;     ACglobal T Mint32 4 ↦ Int;
ACreg b1 ↦ Int;                ACreg b2 ↦ Int;
ACreg p ↦ Ptr({S; T});         ACreg x ↦ Ptr({S})
```

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),

```
eval-ptr: Map[acell, pointsto] → pexpr acell → pointsto+⊤.
```

Given an abstract state ρ mapping each cell to its points-to approximation, and a pointer expression pe, (eval-ptr ρ pe) 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 : Type := (Map[acell, pointsto] × 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(κ,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.

First conversion: load elimination

```
Fixpoint convert pt (e:expr) : set (pexpr acell)+⊤ := match e with
  | id          ⇒ {Pv pt (ACreg id)}
  | c           ⇒ {Pc pt c}
  | (op e)      ⇒ {Puop op pe | pe ∈ convert pt e}
  | (ℓ op r)    ⇒ {Pbop op x y | x ∈ (convert pt ℓ), y ∈ (convert pt r)}
  | load(κ,e) ⇒ map Pv (⋃ pe ∈ convert pt e  (deref-pexpr pt κ pe))
end.

Definition deref-pexpr pt (κ: chunk) (pes: set (pexpr acell)) : set acell+⊤ :=
    ⋃ pe ∈ pes { (ACglobal s κ ofs) | ofs ∈ concretize (nconvert pt pe) ∧
                                      ∃ bs, eval-ptr pt pe = Ptr bs ∧ s ∈ bs }
```

Second conversion: pointer elimination

```
Definition ncconvert (c: constant) : nconstant := match c with
  | n                ⇒ Nintconst n
  | (addrsymbol s ofs) ⇒ Nintconst ofs
end.

Fixpoint nconvert pt (e: pexpr acell) : nexpr acell := match e with
  | (Pv x)              ⇒ Nv x
  | (Pc cst)            ⇒ Nc (ncconvert cst)
  | (Puop negint pe)    ⇒ Euop negint (nconvert pt pe)
  | (Puop boolval pe)   ⇒ Nuop boolval (nconvert pt pe)
                          when eval-ptr pt pe = Int
  | (Puop boolval pe)   ⇒ ne-true   when eval-ptr pt pe = Ptr _
  | (Puop boolval pe)   ⇒ any-bool  otherwise
  | (Pbop + ℓ r)        ⇒ Nbop + (nconvert pt ℓ) (nconvert pt r)
  | (Pbop cmpu(c) ℓ r)  ⇒ Nbop (cmpu c) (nconvert pt ℓ) (nconvert pt r)
                          when eval-ptr pt ℓ = eval-ptr pt r = Int
  | (Pbop cmpu(!=) ℓ r) ⇒ ne-true
                          when eval-ptr pt ℓ = Int
                           and eval-ptr pt r = Ptr _
end.
```

Figure 12. Conversions (excerpt)

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 ncconvert), 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;

```
1:  int x, y, z;
2:  int main(void) {
3:    int *p = any_bool() ? &x : &y;
4:    if ( x < z && y < z)
5:      if (0 < *p) assert (1 < z);
6:    return 0;
7:  }
```

Figure 13. Test case

- 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×(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 4); (ACglobal T Mint32 0); and (ACglobal T Mint32 4).

Then, the conversion proceeds and builds an expression (Pv (ACglobal x Mint32 0)) + 4 × (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 getchar 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.

```
assume e ab := ⨆ { let pt := pt-assume ab pe in
                   (pt, assume (nconvert pt pe))
                 | pe ∈ convert pt ab (Euop boolval e) }
```

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.

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.

```
1:  int x = 0, y = 1;
2:  int *p = any_bool() ? &x : &y;
3:  if ( *p ) { /* p points to y */ }
4:  else { /* p points to x */ }
```

The conversion of the expression *p yields two expressions: one corresponding to variable x, one corresponding to variable y. The numerical domain is then able to prove that assuming that x is true leads to a contradiction. However, this information is not propagated to the points-to domain. Indeed no information about the choices that are made during the conversion is kept. To address this limitation, the conversion could produce, rather than a set of expressions, a set of pairs made of an expression and an abstract environment, where the abstract environment is the original one refined with the choices done during conversion. This would also address the precision loss in expressions like *x + *x. Such an improvement is left as future work.

3.5.3 Assign and Store

Both operators assign and store, whose implementation is given in Figure 14, share the same logic: evaluate an expression, and store its result at some location. The difference between the two is that the destination is definitely known in the case of assign whereas it is denoted by an expression in the case of store. Therefore, this second operator needs first to compute an over-approximation of the set of cells that may be designated by this expression (thanks to deref-pexpr, also used in the load elimination phase). Then, both operators rely on the more general assign-cells that updates a set of cells.

Abstract cells may overlap. Therefore, an explicit update to one cell may hide implicit updates to overlapping cells. To soundly model this property, we conservatively forget anything that is known about any overlapping cell. To do so we rely on an auxiliary function overlapping-cells that computes the set of all cells that are distinct from all cells in a given set but overlap with some of them.

```
Definition assign (x: ident) (e: expr) (ab: t) : t+⊥ :=
assign-cells None { ACreg x } e ab.

Definition store (κ: chunk) (ℓ r: expr) (ab: t) : t+⊥ :=
let cells := deref-pexpr ab κ (convert ab ℓ) in
  assign-cells (Some κ) cells r ab.

Definition assign-cells κ (dst: set acell) (e: expr) ((pt, nm): t) : t+⊥ :=
let pes := convert pt nm e in
let ab' := set-map-reduce dst (λ c,
   let (ty, nm') := set-map-reduce pes (λ pe,
      (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 14. Updating cells: assign and store

To assign to a set of cells the result of a given expression e, this expression is first converted to a set of pointer expressions. Then, 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

```
Concrete cells (type ccell):      c ::= CCmem b κ i | CCreg x

Record fmem: Type := { f-of-fmem:> ccell → val;
                       perm: block → Z → permission }.

Definition disjoint-ranges i κ i' κ' : Prop :=
        i' + Mem.size_chunk κ' ≤ i ∨ i + Mem.size_chunk κ ≤ i'.

Definition disjoint-cells (c c': ccell) : Prop :=
  match c, c' with
  | CCmem b κ ofs, CCmem b' κ' ofs' ⇒
                b' ≠ b ∨ disjoint-ranges ofs κ ofs' κ'
  | CCreg i, CCreg i' ⇒ i ≠ i'
  | _, _ ⇒ True
  end.

Definition fmem-update (c: ccell) (v: val) (f f': fmem) : Prop :=
  f' c = v ∧ (∀ c', disjoint-cells c c'→ f' c' = f c').
```

Figure 15. Specification of concrete memory

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 4) 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 f of type fmem can be used as a function of type ccell → val. Therefore, reading the content of a cell c in a memory f amounts to the function application written (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 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[5].

The abstract transformers of the memory functor can then be specified against this concrete view of the memory, as shown in Figure 16. The forget-sound property reads as follows. The concretization of the result of (forget x ab) contains (at least) all concrete memories obtained after updating an initial memory f (in the concretization of ab) 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 CFG expression e.

Again, the specification of the store transformer is similar, except that the updated cell (CCmem b κ (unsigned i)) is built from any result (vptr b i) of the evaluation of the address expression e_1, and that the stored value v is transformed according to the chunk κ specifying the memory access (as defined by CompCert's load_result function, see Figure 4). In addition, this transformer may use the fact that the destination cell is *writable* (i.e., that the offset i is correctly aligned and that there are sufficient permissions to write there).

[5] 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 5.1 for a discussion.

```
Class fmem-dom (ρ:env) (t: Type) (D:pre-mem-dom t)
                (G:gamma-op t fmem): Prop :=
{
  range-sound: ∀ (ab: t) (f: fmem) (x: ident), f ∈ γ ab →
  match f (CClocal x) with
      vint i | vptr _ i ⇒ i ∈ ints-in-range (range ab x)
    | _ ⇒ True
  end
;
  assume-sound : ∀ (e: expr) (ab: t) (f: fmem) (v: val),
    f ∈ γ(ab) →
    eval-expr ρ f e v →
    bool-of-val v true →
    f ∈ γ(assume e ab)
;
  assign-sound : ∀ (x: ident) (e: expr) (ab: t) (f: fmem) (v: val),
    f ∈ γ(ab) →
    eval-expr ρ f e v →
    fmem-update (CCreg x) v f ⊆ γ(assign x e ab)
;
  store-sound : ∀ (κ: chunk) (e1 e2: expr) (ab: t) (f: fmem)
                  (b: block) (i: int) (v: val),
    let c := CCmem b κ (unsigned i) in
      f ∈ γ(ab) →
      eval-expr ρ f e1 (vptr b i) →
      eval-expr ρ f e2 v →
      c ∈ writable-cell f →
      fmem-update c (load-result κ v) f ⊆ γ(store κ e1 e2 ab)
;
  forget-sound : ∀ (x: ident) (ab: t) (f: fmem) (v: val),
    f ∈ γ ab →
    fmem-update (CCreg x) v f ⊆ γ(forget x ab)
}.
```

Figure 16. Specification of our abstract memory functor

4.2 Points-to Domain

The soundness of the points-to domain is established against a concretization relation defined as follows. The Int abstract value represents all machine integers. The abstract values of the form (Ptr bs) represent all pointers whose block is in the set bs (the offset is not constrained). The trivial abstract type All is related to any value. In particular, it is the only abstract value that represents the bogus value vundef.

This invariant enables us to prove that some value cannot be vundef. This is particularly useful for proving progress (as it 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 ncompat 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 pre-gamma) for the abstract memory functor, where concrete values are functions from abstract cells to values.

The set (pre-gamma (pt, nm)) is the intersection of the concretization γ(pt) of the points-to map and of the set of all memories related (by the point-wise lifting of the ncompat relation) to some function ν in the concretization γ(nm) of the numerical abstraction.

```
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 κ o ⇒
      do_opt b ← symbol ge g;
      Some(CCmem b κ o)
  | ACreg i ⇒ Some(CCreg i)
  end.

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

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

Figure 17. Concretization relation

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 δ_0. It is defined using a monadic style (hence the 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 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 ∘ 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 v 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 ∈ ⋃ pe ∈ 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 v of the original expression pe, the resulting expression evaluates to some number n that is compatible with value v. The compatibility relation ncomp was previously defined in Figure 17. 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),
  (∀ x : acell, ncompat (ρ x) (ν x)) →
  ∀ (pe : pexpr acell) (v : val),
  v ∈ peval ρ pe →
  ∃ n : int, n ∈ (neval ν (nconvert pt pe) ∩ 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 v the expression e may evaluate to in f, for every destination cell a (related to the concrete cell c through δ_0), the concrete memories obtained by updating f at c with value (load_result κ v) 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 →
  ∀ v, eval-expr m e v →
  ∀ a, a ∈ dst →
  ∀ c, δ0 a = Some c →
  fmem-update c (load-result κ v) 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 runtime 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.

```
1:  union { int i; uint8_t b[4]; } u;
2:  u.i = 0x12345678;
3:  switch (u.b[0]) {
4:    case 0x12:
5:      return BIG_ENDIAN;
6:    case 0x78:
7:      return LITTLE_ENDIAN;
8:  }
9:
10: void memcpy(void *dest, void *src, size_t n) {
11:   uint8_t *d = dest;
12:   const uint8_t *s = src;
13:   int i;
14:   for ( i = 0 ; i < n ; ++i ) {
15:     d[i] = s[i];
16:   }
17: }
```

Figure 18. Type punning examples

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 floats[6]. The first are similar to casts (with the cast8unsigned 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 (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.

```
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.

```
Definition pun-u8 (f: cell → val) := ∀ b ofs κ,
  (align-chunk κ | ofs) →
  ∃ bytes,
    f (CCmem b κ ofs) = decode-val κ bytes ∧
    length bytes = size_chunk κ ∧
    ∀ i (LT: i < length bytes),
    f (CCmem b Mint8unsigned (ofs + i)) =
          decode-val Mint8unsigned (get i 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

[6] Fast computations of approximations of inverse square roots are a notable example of floating-point operations performed on the bit-level representation of floats.

```
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: }
```

Figure 19. C program checking for null pointers

bytes of bytes from which this value is decoded, such that reading in memory at the same address the i^{th} Mint8unsigned chunk yields the decoding of the i^{th} byte in the sequence. In this property, the get function takes as argument a proof LT that there are at least 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.

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

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 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.

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).

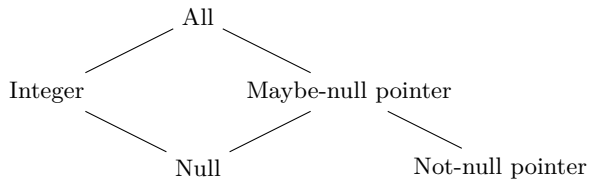

Figure 20. Lattice of abstract types to handle null pointers

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 not-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 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.

Instance gamma : gamma-op t fmem := λ ab m,
  $\forall\ \delta,\ \delta \in$ allocation-spec $\rightarrow$ mem-rel δ m $\subseteq$ 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

Program	Size (instr.)	Analysis time (s)
aes	960	6.0
almabench	217	5.0
fft	203	0.02
fftw	88	6.5
integr	49	0.02
nbody	117	10
sha3	287	121
siphash24	272	0.2
core1	154	0.04
core2	142	0.04
core4	198	0.05
hash (256)	457	1.6
hash (512)	523	2.2
scalarmult	961	22
stream	920	507
random (1)	45	1.1
random (2)	3952	90
random (3)	4491	1.5
sat (20)	4642	0.8
sat (50)	66 322	90
blowfish	177	31
des	229	2.05
donna	1217	484
rc4	93	4.05
salsa20	341	4.52
snow	871	2.44
tea	121	3.18

Table 1. Verasco analysis times

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 1 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.

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 dataflow 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 [3] used a very coarse memory abstraction, without tracking memory contents. Other previous works on formal verification of abstract interpretation [2, 24] 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

[1] Companion website, 2016. http://www.irisa.fr/celtique/ext/abstract-memory.

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

[3] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Proceddings of Static Analysis Symposium (SAS)*, volume 7935 of *LNCS*, pages 324–344. Springer, 2013.

[4] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of FMPA*, volume 735 of *LNCS*, pages 128–141. Springer, 1993.

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

[6] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *Symposium on Principles of Programming Languages* (*POPL*), pages 238–252. ACM, 1977.

[8] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In 5^{th} *Symposium on Principles of Programming Languages* (*POPL*), pages 84–97, Tucson, Arizona, 1978. ACM.

[9] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE Analyser. In *Proceedings of European Symposium On Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.

[10] Alexis Fouilhé, David Monniaux, and Michaël Périn. Efficient generation of correctness certificates for the abstract domain of polyhedra. In *Proceedings of Static Analysis Symposium (SAS)*, volume 7935 of *LNCS*, pages 345–365. Springer, 2013.

[11] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 512–529. Springer, 2004.

[12] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4): 165–190, 1989.

[13] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *Proceedings of TAPSOFT'91*, pages 169–192. Springer, 1991.

[14] ISO. The ANSI C standard *(*C99*)*. Technical Report WG14 N1124, ISO/IEC, 1999. URL http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf.

[15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *Proc. of the 42^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2015.

[16] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, 1976.

[17] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine, and compiler. *ACM Transactions Programming Languages and Systems*, 28(4): 619–695, 2006.

[18] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[19] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[20] Xavier Leroy and Valentin Robert. A formally-verified alias analysis. In *Proceedings of Certified Proofs and Programs (CPP)*, volume 7679 of *LNCS*, pages 11–26. Springer, 2012.

[21] Antoine Miné. *Weakly relational numerical abstract domains*. Ph.D. thesis, École Polytechnique, 2004.

[22] Antoine Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *Proc. of The ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (*LCTES'06*), pages 54–63. ACM, 2006.

[23] Durica Nikolic and Fausto Spoto. Inferring complete initialization of arrays. *Theoretical Computer Science*, 484:16–40, 05 2013.

[24] Tobias Nipkow. Abstract interpretation of annotated commands. In *Proceedings of Interactive Theorem Proving (ITP)*, volume 7406 of *LNCS*, pages 116–132. Springer, 2012.

[25] The Coq development team. *The Coq proof assistant reference manual*. Inria, 2012. URL http://coq.inria.fr. Version 8.4.

[26] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Conference on Programming Languages Design and Implementation (PLDI)*, volume 46, pages 283–294. ACM, 2011.