Formal Verification of a Memory Model for *C*-Like Imperative Languages

Sandrine Blazy and Xavier Leroy

INRIA Rocquencourt
78 153 Le Chesnay cedex, France
{Sandrine.Blazy, Xavier.Leroy}@inria.fr

Abstract. This paper presents a formal verification with the *Coq* proof assistant of a memory model for *C*-like imperative languages. This model defines the memory layout and the operations that manage the memory. The model has been specified at two levels of abstraction and implemented as part of an ongoing certification in *Coq* of a moderately-optimising *C* compiler. Many properties of the memory have been verified in the specification. They facilitate the definition of precise formal semantics of *C* pointers. A certified *OCaml* code implementing the memory model has been automatically extracted from the specifications.

1 Introduction

Formal verification of computer programs – be it by model checking, program proof, static analysis, or any other means – obviously requires that the semantics of the programming language in which the program is written be formalized in a way that is exploitable by the verification tools used. In the case of program proofs, these formal semantics are often presented as operational semantics or specialized logics such as Hoare logic. The need for formal semantics is even higher when the program being verified itself operates over programs: compilers, program analyzers, etc. In the case of a compiler, for instance, no less than three formal semantics are required: one for the implementation language of the compiler, one for the source language, and one for the target language. More generally speaking, formal semantics "on machine" (that is, presented in a form that can be exploited by verification tools) are an important aspect of formal methods.

Formal semantics are relatively straightforward in the case of declarative programming languages such as pure functional or logic languages. Many programs that require formal verification are written in imperative languages, however. These languages feature assignments to variables and in-place modification of data structures. Giving semantics to these imperative constructs requires the development of an adequate *memory model*, that is, a formal description of the memory layout and the operations over it. The memory model is often one of the most delicate parts of a formal semantics for an imperative programming language: an excessively concrete memory model (*e.g.* representing the memory as a

single array of bytes) can fail to validate algebraic laws over loads and stores that are actually valid in the programming language and thus make program proofs more difficult; an excessively abstract memory model can fail to account for *e.g.* aliasing or partial overlap between memory areas, thus causing the semantics to be incorrect.

This paper reports on the design, formalization and verification, using the *Coq* proof assistant, of a memory model for *C*-like imperative languages. In addition to being widely used for programming safety-critical software, *C* and related languages are challenging from the standpoint of the memory model, because they feature both pointers and pointer arithmetic, on the one hand, and isolation and freshness guarantees on the other. For instance, pointer arithmetic can result in aliasing or partial overlap between the memory areas referenced by two pointers; yet, it is guaranteed that the memory areas corresponding to two distinct variables or two successive calls to `malloc` are disjoint. This stands in contrast with both higher-level imperative languages such as *Java*, where two distinct references always refer to disjoint data, and lower-level languages such as machine code, where unrestricted address arithmetic invalidates all isolation guarantees.

The memory model presented here is used in the formal verification of a moderately-optimising compiler that translates a large subset of the *C* programming language down to *PowerPC* assembly code [13]. The memory model is used by the formal semantics of all languages manipulated by the compiler: the source language (large subset of *C*), the target language (subset of *PowerPC* assembly), and 5 intermediate languages that bridge the semantic gap between source and target. Certain passes of the compiler perform non-trivial transformations on memory allocations and accesses: for instance, the `auto` variables of a *C* function, initially mapped to individually-allocated memory blocks, are at some point mapped to sub-blocks of a single stack-allocated activation record, which at a later point is extended to make room for storing spilled temporaries. Proving the correctness (semantic preservation) of these transformations require extensive reasoning over the memory model, using the properties of this model given further in the paper.

The remainder of this paper is organized as follows. Section 2 presents how we have formally verified a compiler with the *Coq* proof assistant. Section 3 describes the formal verification of our memory model. Section 4 explains how *OCaml* code has been automatically generated from this verification. Section 5 discusses related work. Finally, section 6 concludes.

2 Certification of a *C*-like Compiler

The formal verification of a compiler is the formal proof of the following equivalence result: any source program that terminates on some final memory state is compiled into a program that also terminates and produces the same memory state. Usually, such an equivalence result relies on a more general notion of equivalence between memory states. But, our memory model aims at facilitating this

correctness proof and it is designed in such a way that the memory states are the same at the end of the execution of source and compiled programs. The correctness result is not proved directly but in several steps. Each step corresponds to a transformation (that is, either a translation or an optimisation) achieved by the compiler. Each correctness proof of a transformation proceeds by induction on the execution of the original program using a simulation lemma: if the original program executes one statement, the transformed program executes zero, one or several statements.

Our compiler treats a large subset of *C*. It compiles any *C* program in which jump statements (*i.e.* `goto`, `setjmp` and `longjmp`) are not allowed, and functions have a fixed number of arguments. The expression evaluation order is defined in the compiler: expressions are evaluated from left to right, thus leaving less freedom to the compiler. Furthermore, as dynamic allocation of variables is done explicitly in *C* by calling the library functions `malloc` and `free`, the semantics of these functions is not defined in our formal semantics and there is no garbage collector in the compiler. The proof that these functions ensure lack of dangling pointers is thus out of the scope of this paper.

The formal verification of the memory model belongs to an ongoing formal verification with the *Coq* proof assistant of this compiler, and it consists of:

- a formal specification at several levels of abstraction a memory model,
- a formal proof about many properties of this memory model,
- the automatic generation from the specification of a certified code that verifies the same properties as the formal specification.

The *Coq* proof assistant [1, 4] consists mainly of a language called *Gallina* for writing formal specifications and a language for developing mathematical proofs to verify some properties on the formal specifications. *Gallina* relies on the *Calculus of Inductive Constructions*, a higher-order typed $\lambda-$calculus with dependent types and capabilities for inductive definitions. Proving a simple property consists in writing interactively proof commands that are called tactics. Tactics may also consist of user-defined tactics, thus making it possible to decompose a property into simpler reasoning steps and to reuse proof scripts.

Coq provides a way to structure specifications in large units called modules. The *Coq* module system [7] reuses the main features of the *OCaml* module system. A module is a collection of definitions of types, values and modules. It consists of two parts: a signature and an implementation. The signature of a module is an abstract specification of the components that must occur in all possible implementations of that module. The type of a module is its signature. Modules can be parametrised by modules. Parametrised modules are called functors (*i.e.* functions from modules to modules). One way to build modules is to apply a functor. The other way is to build it definition by definition. A module may be associated with a signature to verify that the definitions of the module are compatible with the signature. Properties may be defined in modules. When a property is defined in the signature of a module, it must be proved in any implementation of this module. The property is thus called an axiom (resp. theorem) in the signature (resp. implementation) of the module.

Coq provides also an automated mechanism for extracting functional programs from specifications [14]. The extraction from a *Coq* function or proof removes all logical statements (*i.e.* predicates) and translates the remaining content to a program written in *OCaml*. As the extracted program verifies the same properties as the *Coq* specification, the extracted code is called the certified code. The *Coq* extraction mechanism handles the module system: *Coq* modules are extracted to *OCaml* modules, *Coq* signatures are extracted to *OCaml* signatures, and *Coq* functors are extracted to *OCaml* functors.

3 Formal Specification

This section describes the formal verification in *Coq* of our memory model. It specifies the memory layout and the operations that manage the memory. This formal specification is written at two levels of abstraction:

- The abstract specification is suitable for most of imperative languages. It defines a general memory model, parametrised by some characteristics of the language it applies to (*e.g.* the values of the language), and properties that need to be verified by a more concrete specification.
- The concrete specification is devoted to *C*-like languages with pointer arithmetic. It implements the operations defined in the abstract specification, and proves that they satisfy the abstract specification. The properties that have been stated in the abstract specification are proved in the concrete specification. Other properties are also stated (and proved) in the concrete specification.

This section presents two concrete specifications. The first one is devoted to an infinite memory model of a *C* compiler. The second one defines a finite memory model that corresponds to the first concrete specification. In this paper, we will use familiar mathematical notation to present our development in *Coq*. For instance, inductive definitions will be presented in BNF format and *Coq* arrows will be replaced by either conjunctions or implications.

3.1 Abstract Specification

The abstract specification defines the memory layout in terms of records and maps. Several types are left unspecified. The operations that manage the memory are only defined by their types. Some axioms are also defined in the abstract specification.

Memory Layout Figure 1 describes the types that specify the memory layout. The memory is separated into four areas that do not overlap:

- the free memory called mem_{free} that can be allocated during the execution of a program,

- the null memory called mem_{null} that is not accessible during the execution of a program,
- the memory called mem_{data} that stores data,
- the memory called mem_{code} that stores code, *i.e.* the procedures of a program. [1]

The type of memory is called *Tmem*. It is a record whose four fields represent the four areas. Each area is represented by a map (that is, a partial finite function) of type $Tmem_i$ from blocks identifiers *Tblock* to blocks $Tblock_i$, where i denotes a memory area. *Tblock* is an ordered type and $\leq$ denotes an order relation on *Tblock*. A block consists of a low bound, a high bound and a map from offsets *Tofs* (*i.e.* cells identifiers) to memory cells $Tcell_i$. $Tcell_i$ is equipped with a comparison relation that we write $=$. The high and low bounds of a block are block identifiers. The contents of the cells in a block depend on the area the block belongs to. Usually, each cell of the data area stores a value on a given number of bytes. Each cell of the code area stores a procedure (*i.e.* a C function). Each cell of the null area stores either a deallocated cell or a null cell that has never been deallocated.

The types that are left unspecified in the abstract specification are related to the way blocks and cells are addressed (*cf. Tblock* and *Tofs*) and to the contents of memory cells (*cf.* $Tcell_i$, $\forall i \in \{data, free, null\}$ and *Tprocedure*). The four areas of the memory are handled in a similar way. For space reasons, this paper focuses on the memory area that stores data.

MEMORY LAYOUT:

$$\begin{aligned} Tmem \quad &::= \{mem_{data} := Tmem_{data} \\ &\quad ; mem_{free} := Tmem_{free} \\ &\quad ; mem_{null} := Tmem_{null} \\ &\quad ; mem_{code} := Tmem_{code} \ \} \end{aligned}$$

MEMORY AREAS:

$$\forall i \in \{data, free, null, code\},\ Tmem_i \ ::= \texttt{Map}\,(Tblock, Tblock_i)$$

MEMORY BLOCKS:

$$\begin{aligned} \forall i \in \{data, free, null\},\ Tblock_i \ &::= \{high := Tblock \\ &\quad ; low := Tblock \\ &\quad ; contents := \texttt{Map}\,(Tofs, Tcell_i) \ \} \\ Tblock_{code} \quad &::= Tprocedure \end{aligned}$$

Fig. 1. Abstract specification of the memory layout: type definitions.

[1] In the sequel of this paper, we use the word *procedure* to denote a C function. The word *function* is reserved to *Coq* mathematical functions that are defined in the specification.

Figure 2 defines some relations between blocks and memory and some of their properties. The relation called valid_data_block states that a block b is valid with respect to a memory m if it has been allocated in the area of m that stores data (*i.e.* it belongs to the domain of the map $m.memdata$.). This relation is often used as a precondition in the operations that manage the memory (see for instance the definition of load in figure 5). The axiom called valid_not_valid_diff states that any block is either valid or not.

The relation called block_agree is an agreement relation between blocks. Two blocks belonging to two memories agree between two bounds called *lo* and *hi* if they share a same identifier b and if each of their cells that is between the bounds *lo* and *hi*, stores the same value. This relation is an equivalence relation: it verifies the three axioms called block_agree_refl, block_agree_sym and block_agree_trans.

The relation called extends states that a memory *m2* extends another memory *m1* if each valid block b of *m1* is also a block of *m2*. More precisely, if b identifies a valid block $(m1.memdata)(b)$ of *m1*, then it identifies also a bigger block $(m2.memdata)(b)$ of *m2* (*i.e.* a block such that its cells are included in the cells of $(m2.memdata)(b)$) and both blocks agree between the bounds of the smallest block $m1(b)$. The picture of figure 2 shows an example of two such blocks. The compilation process relies on a run-time stack of memory blocks called stack frames. At the beginning of the compilation process of a program, a stack frame is allocated for each instance of a called procedure. Information that are computed in further steps of the compilation process are stored in stack frames and reused in further steps of the process. The relation called extends is useful to specify the extension of stack frames during the compilation process.

Memory Management The main operations that manage the memory are alloc, free, load and store. They are specified in the figure 3, where alloc, load and store are related to the memory area that stores data. Similar operations related to the memory area that stores code have also been specified. Each operation that manage the memory may fail (*e.g.* alloc may fail if there are no free cells left). Thus, its results is of type $\texttt{option}(\tau)$. The values of such a type are either `None` (when the operation fails) or $\texttt{Some}(v)$ where v is of type τ.

load and store operations are parametrised by memory chunks. A memory chunk indicates the size and the type of accessed data. Its type is called *Tchunk* and is left unspecified in the abstract specification. Memory chunks ensure that each load operation follows a store operation that supplied the value retrieved by the load. For instance, when an operation such as (store *chunk1 m1 b ofs1* = Some *m2*) is followed by an operation such as (load *chunk2 m b ofs2*) then the load does not fail only if *chunk1*, *chunk2*, *ofs1* and *ofs2* are compatible.

The functionalities of the memory management operations are the following:

- alloc is the function that allocates a block with given bounds. If it does not fail, this function yields a newly allocated block and the modified memory.
- free is the function that deallocates a given block of data.
- load is the function that given a memory chunk fetches the value stored in a given block of data.

Definition valid_data_block (m: $Tmem$) (b: $Tblock$) := $b \in$ `domain`($m.memdata$).

Axiom valid_not_valid_diff:
  $\forall$ (m: $Tmem$) (b b': $Tblock$),
  valid_data_block m $b \wedge \neg$(valid_data_block m b') $\Rightarrow$ $b \neq b'$.

Definition block_agree (b: $Tblock$) (lo hi: $Tblock$) ($m1$ $m2$: $Tmem$) :=
  $\forall$ $ofs \in [lo,hi]$,
  $((m1.memdata)(b).contents)(ofs) = ((m2.memdata)(b).contents)(ofs)$.

Axiom block_agree_refl:
  $\forall$ (m: $Tmem$) (b: $Tblock$) (lo hi: $Tblock$),
        block_agree b lo hi m m.

Axiom block_agree_sym:
  $\forall$ ($m1$ $m2$: $Tmem$) (b: $Tblock$) (lo hi: $Tblock$),
  block_agree b lo hi $m1$ $m2$ $\Rightarrow$
        block_agree b lo hi $m2$ $m1$.

Axiom block_agree_trans:
  $\forall$ ($m1$ $m2$ $m3$: $Tmem$) (b: $Tblock$) (lo hi: $Tblock$),
  block_agree b lo hi $m1$ $m2 \wedge$ block_agree b lo hi $m2$ $m3$ $\Rightarrow$
        block_agree b lo hi $m1$ $m3$.

Definition extends ($m1$ $m2$: $Tmem$) :=
  $\forall$ (b: $Tblock$),
    valid_data_block $m1$ b $\Rightarrow$
          $(m2.memdata)(b)$.low $\leq (m1.memdata)(b).low \wedge$
          $(m1.memdata)(b).high \leq (m2.memdata)(b).high \wedge$
          block_agree b $(m1.memdata)(b).low$ $(m1.memdata)(b).high$ $m1$ $m2$

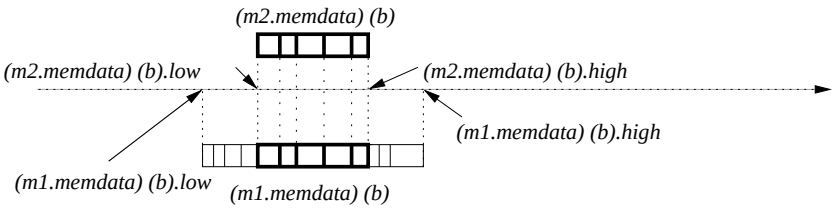

Fig. 2. Abstract specification of the memory layout: properties.

– store is the function that given a memory chunk stores a value in a given block of data. The load (resp. store) function fails if the value to load (resp. store) is not compatible with the memory chunk and the offset (*e.g.* if the memory chunk is to large). As these functions are left unspecified at the abstract level, this property consists of axioms such as loaded_block_is_valid and loaded_block_is_in_bounds that will be proved once the functions will be defined.

The axiom called loaded_block_is_in_bounds uses a property called in_bounds that defines when a value may be loaded from or stored in the two bounds of a block. in_bounds is used as a precondition that triggers loads and stores in memory. As block identifiers and offsets are left unspecified in the abstract specification, in_bounds is also left unspecified. It is a relation, *i.e.* a function that yields values of a type called Prop. This *Coq* type is used to define logical propositions.

Other properties of the operations that manage the memory express that the relations between blocks are preserved by the memory management operations. For instance, the axiom called valid_block_store expresses that the load operation does not invalidate valid blocks. More precisely, it states that if a value v is stored in a memory *m1.memdata*, any block b that was valid before the operation remains valid after. The axiom called store_agree states that the store operation preserves the agreement relation. The axiom load_extends states that the load operation preserves the extension relation. Figure 3 shows only some axioms of the specification. Similar axioms have been defined for all memory management operations.

3.2 Implementation of an Infinite Memory

This section presents an implementation of our memory model that is devoted to a *C*-like compiler. The implementation of values and addresses is adapted to *C* pointer arithmetic and the implementation of memory chunks follows the *C* arithmetic types. In this implementation, the memory is unlimited and thus the allocation never fails. New properties of the memory management are added in this implementation.

For each language manipulated by our compiler, we have encoded in *Coq* operational semantics rules that detail how the memory is accessed and modified during the execution of a program. For instance, the evaluation of a procedure respects the following judgements of the source and target languages of the compiler (called respectively *C* and *PPC*):

$G_c \vdash p_c, lv, m \Rightarrow v, m'$ states that in the global environment G_c and the memory m, the evaluation in *C* of the procedure p_c called with the list of values lv of its arguments computes a value v. The memory at the end of the evaluation in *C* is m'.

$G_{ppc} \vdash r, m \dashrightarrow r', m'$ states that in the global environment G_{ppc}, the evaluation in *PPC* of the current function updates the set of registers r into r' and the memory m into m'.

Memory management operations:

alloc : $Tmem \to Tblock \to Tblock \to \texttt{option}\,(Tmem * Tblock)$
free : $Tmem \to Tblock \to \texttt{option}\,(Tmem)$
load : $Tchunk \to Tmem \to Tblock \to Tofs \to \texttt{option}\,(Tvalue)$
store : $Tchunk \to Tmem \to Tblock \to Tofs \to Tvalue \to \texttt{option}\,(Tmem)$

Relation between blocks and memory chunks:

in_bounds : $Tchunk \to Tofs \to Tblock \to Tblock \to \texttt{Prop}$

Some properties of memory management operations:

Axiom loaded_block_is_valid:
  $\forall$ ($chunk$: $Tchunk$) (m: $Tmem$) (b: $Tblock$) (ofs: $Tofs$) (v:$Tvalue$),
  load $chunk$ m b ofs = Some v $\Rightarrow$
    valid_data_block m b.

Axiom loaded_block_is_in_bounds:
  $\forall$ ($chunk$: $Tchunk$) (m: $Tmem$) (b: $Tblock$) (ofs: $Tofs$) (v: $Tvalue$),
  load $chunk$ m b ofs = Some v $\Rightarrow$
    in_bounds $chunk$ ofs $(m.memdata)(b).low$ $(m.memdata)(b).high$.

Axiom valid_block_store:
  $\forall$ ($chunk$: $Tchunk$) ($m1$ $m2$: $Tmem$) (b b': $Tblock$) (ofs: $Tofs$) (v: $Tvalue$),
  store $chunk$ $m1$ b' ofs v = Some $m2$ $\wedge$
  valid_data_block $m1$ b $\Rightarrow$
    valid_data_block $m2$ b.

Axiom store_agree:
  $\forall$ ($chunk$: $Tchunk$) ($m1$ $m2$ $m1'$ $m2'$: $Tmem$) (b b': $Tblock$)
    (lo hi: $Tblock$) (ofs: $Tofs$) (v: $Tvalue$),
  block_agree b lo hi $m1$ $m2$ $\wedge$
  store $chunk$ $m1$ b' ofs v = Some $m1'$ $\wedge$
  store $chunk$ $m2$ b' ofs v = Some $m2'$ $\Rightarrow$
    block_agree b lo hi $m1'$ $m2'$.

Axiom load_extends:
  $\forall$ ($chunk$: $Tchunk$) ($m1$ $m2$: $Tmem$) (b: $Tblock$) (ofs: $Tofs$) (v: $Tvalue$),
  extends $m1$ $m2$ $\wedge$
  load $chunk$ $m1$ b ofs = Some v $\Rightarrow$
    load $chunk$ $m2$ b ofs = Some v.

Fig. 3. Abstract specification of the memory management.

These semantics rely on the memory management operations. For instance, in the dynamic semantics of *PPC*, references to variables correspond to explicit loads and stores. There are 13 load instructions and 10 store instructions in *PPC*. In the dynamic semantics of *C*:

- A block of memory is allocated for each declared variable. The cells of the block that stores an array consist of the elements of the array.
- Such a block is deallocated at the end of the scope of the variable.
- The evaluation of a left value loads a value from memory.
- The execution of any assignment statement is based on the load and store operations.

Memory Layout Figure 4 defines the types that were left unspecified in the abstract specification in figure 1. The blocks and the offsets of a block are identified by integers. The sizes of stored values are one, two, four and eight bytes. Values are either undefined values, or integers or floats or non null pointer values. The undefined value `Vundef` is a junk value that represents the value of uninitialised variables. A value of type pointer is either the integer `0` (that represents the `NULL` pointer) or a pair of a block identifier (that is, the address of the first cell of the block) and an offset between the block and the cell the pointer points to. This representation of pointers is adapted to *C* pointer arithmetic. For instance, the expression `(Vptr b ofs) + (Vint i)` evaluates to the pointer value `(Vptr b Vint (ofs + i))` if this evaluation does not fail. In other words, the only integers i that can be added to a pointer value are those such that `(ofs + i)` is in the bounds of the block `b`. Another example is the comparison between pointers: two pointers that are not `NULL` may be compared only if they point to a same block.

ADDRESSES:

Tblock	$::= \mathbb{Z}$	
Tofs	$::= \mathbb{Z}$	

VALUES:

$Tcell_{data}$	$::= Tsize * Tvalue$	a data cell is a pair of a size and a value
Tsize	$::= \{1, 2, 4, 8\}$	number of bytes of a cell
Tvalue	::= `Vint` *Tinteger*	integer
	| `Vfloat` *Tfloat*	float
	| `Vptr` *Tblock Tofs*	pointer (a block and an offset)
	| `Vundef`	undefined value

Fig. 4. An implementation of the memory layout.

Usually, properties of memory layouts are classified into separation, adjacency and containment properties [26]. This is also the kind of properties of our memory

model. Separation and adjacency of memory blocks are valid in our model by construction. By construction, each memory block belongs to only one memory area. Two different blocks are also separated by construction since a cell of a block can not be accessed from another block. The containment property we use is the extends relation.

Memory Management The memory chunks that were left unspecified in figure 3 are implemented in figure 5 in the following way: integers are stored on either one, two or four bytes, and floats are stored on either four or eight bytes. Integers that are stored on one or two bytes are either signed or unsigned. Pointer values are implemented by integers stored on four bytes.

The alloc and free functions never fail. The allocation method is linear. load *chunk m b ofs* fails when *b* does not identify a block of the data area of *m* and when the property in_bounds *chunk ofs b* is not true. The load function calls the load_result function in order to load each cell that needs to be loaded in the block *b* from the offset *ofs*. The load_result function fetches a value in memory and casts this value to a value of a type defined by a memory chunk, when the memory chunk is compatible with the value. Memory chunks determine also if a block needs to be filled with digits. For instance, when an integer that is stored on one or two bytes is loaded, it is automatically extended to four bytes (by the function called load_result), either by adding zeroes if the integer is unsigned, or by replicating the sign bit if the integer is signed (see the function cast1signed called by load_result). The load_result function fails if the memory chunk is not compatible with the value, for instance if it attempts to load a float value when the memory chunk corresponds to an integer. For space reasons, the definition of this function is not fully detailed in figure 5.

Some new properties of the operations that manage the memory are defined in the implementation. They have not been defined in the abstract specification because they rely on the implementation of the memory management operations. These properties express that the memory blocks remember correctly the stored values. More precisely:

1. If an operation updates a block of a memory area by storing a value in it, then the content of this block becomes this value,
2. and the other blocks of memory are not modified.
3. A block which is modified by an operation belongs to the memory that results from the modification.

These properties are often called the good variable properties [25]. Our certification uses them in order to prove analogous properties on stack frames built by the compiler. As these properties are related to memory blocks consisting of memory cells, their proof relies on analogous properties for memory cells.

Figure 6 specifies some of the good variable properties. In the two theorems called load_store_same and load_store_other, a value v is stored in a memory *m1* at the offset *ofs1* of a block *b1*, given a memory chunk called *chunk*. The resulting memory is called *m2*. The first theorem called load_store_same states that v is

MEMORY CHUNKS:

Tchunk ::=	`Mint1signed`	signed integer stored on one byte
	| `Mint1unsigned`	unsigned integer stored on one byte
	| `Mint2signed`	signed integer stored one on two bytes
	| `Mint2unsigned`	unsigned integer stored on two bytes
	| `Mint4`	integer stored on four bytes
	| `Mfloat4`	float stored on four bytes
	| `Mfloat8`	float stored on eight bytes

MEMORY MANAGEMENT OPERATIONS:

```
Definition size_chunk (chunk: Tchunk) := ...
    (* number of bytes corresponding to chunk, e.g. 4 for Mint4 *)

Definition in_bounds (chunk: Tchunk) (ofs: Tofs) (lo hi: Tblock) :=
  lo ≤ ofs ∧ ofs + size_chunk chunk ≤ hi.

Definition load_result (chunk: Tchunk) (v: Tvalue) :=
  match chunk, v with
    | Mint1signed, Vint n : Some (Vint (cast1signed n))
        (* values are casted in order to fit the memory chunks *)
    | ...
    | Mint4, Vptr b ofs :    Some (Vptr b ofs)
    | Mfloat4, Vfloat f :    Some (Vfloat (singleoffloat f))
    | ...
    | _, _ : None
        (* erroneous cases, e.g. an integer chunk such as Mint4 and a float value *)
  end.

Definition load (chunk: Tchunk) (m: Tmem) (b: Tblock) (ofs: Tofs) :
  if valid_data_block m b ∧ in_bounds chunk ofs m(b).low m(b).high
  then load_result chunk ...
        (* the second parameter is the value that is found in cell b at offset ofs *)
  else None.
```

Fig. 5. An implementation of the memory management.

also the value that is loaded in $m2$ at the address where it has been stored. The second theorem called load_store_other states that the store operation of v (in a block *b1* at the offset *ofs1*) does not change any other value of the memory, *i.e.* any other value that is fetched either in another block $b2$ or in the same block *b1* but at another valid offset *ofs2*. An offset is valid in a block if there are enough remaining cells in the block in order to store a value form this offset.

Other properties are related to the high and low bounds of memory blocks. They express the compatibility between the bounds of a block and the offset from where a value is stored or loaded in that block. For instance, the theorem low_bound_store of figure 7 states that if a value v is stored in a memory $m1$, then the resulting memory $m2$ has the same low bound as $m1$. Finally, a few other relations between the memory management operations. For instance, the

```
Theorem load_store_same:
  ∀ (chunk: Tchunk) (m1 m2: Tmem) (b1: Tblock) (ofs1: Tofs) (v: Tvalue),
  store chunk m1 b1 ofs1 v = Some m2 ⇒
        load chunk m2 b1 ofs1 = Some (load_result chunk v).

Theorem load_store_other:
  ∀ (chunk1 chunk2: Tchunk) (m1 m2: Tmem) (b1 b2: Tblock)
    (ofs1 ofs2: Tofs) (v: Tvalue),
  store chunk1 m1 b1 ofs1 v = Some m2 ∧
  (b1 ≠ b2 ∨ ofs2 + size_chunk chunk2 ≤ ofs1 ∨ ofs1 + size_chunk chunk1 ≤ ofs2)
        ⇒ load chunk2 m2 b2 ofs2 = load chunk2 m1 b2 ofs2.
```

Fig. 6. Some good-variable properties

theorem called store_alloc states that a value may be stored from a given offset in a newly allocated block if the memory chunk and the offset are compatible with the bounds of this block.

```
Theorem low_bound_store:
  ∀ (chunk: Tchunk) (m1 m2: Tmem) (b b': Tblock) (ofs: Tofs) (v: Tvalue),
  store chunk m1 b ofs v = Some m2 ⇒
        (m2.memdata)(b').low = (m1.memdata)(b').low.

Theorem store_alloc:
  ∀ (chunk: Tchunk) (m1 m2: Tmem) (b lo hi: Tblock) (ofs: Tofs) (v: Tvalue),
  alloc m1 lo hi = Some (m2, b) ∧
  in_bounds chunk ofs lo hi ⇒
        ∃ m3| store chunk m2 b ofs v = Some m3.
```

Fig. 7. Other properties of memory management operations

3.3 Implementation of a Finite Memory

The execution of a source program may exceed the memory of the target machine. Thus, we have implemented another memory model where the size of memory cells and the number of blocks in each memory area are finite. The only difference with the previous model relies in the implementation of the alloc operation: the allocation of a block fails if there is no free cell left. Thus, the theorems such as store_alloc that are defined in the first implementation still hold in this second implementation. When the allocation does not fail, it behaves as the allocation of the infinite memory. This is shown in figure 8. The theorem alloc_finite_to_infinite results from the definition of both allocation operations.

The compilation of a program fails as soon as an allocation fails. As each step of the compilation process allocates memory blocks, there are many opportunities for the compiler to fail. In the memory that stores data, the evolution of block

ABSTRACT SPECIFICATION:

alloc : $Tmem \to Tofs \to Tofs \to \texttt{option}\,(Tmem * Tblock)$

TWO IMPLEMENTATIONS:

```
Definition alloc1 (m:Tmem) (lo hi: Tblock) :=
    Some ...    (* never fails *)

Definition alloc2 (m:Tmem) (lo hi: Tblock) :=
  if (* no free cell left *) then None
  else alloc1 m lo hi.

Theorem alloc_finite_to_infinite:
  ∀ (m1 m2: Tmem) (b lo hi: Tblock),
  alloc2 m1 lo hi = Some (m2, b) ⇒ alloc1 m1 lo hi = Some (m2, b).
```

Fig. 8. Reuse of the allocation operation

allocation during the compilation process is the following. For each instance of a called procedure:

- The dynamic semantics of C allocates a block for each declared variable.
- The translation from C to the first intermediate language L_1 of the compiler allocates a single block for all the local variables of the procedure that are either of array type or whose addresses are taken. Thus the number of allocated blocks decreases but the size of each block increases.

In the case of the translation from C to L_1, the size of all allocated blocks in the data area is the same in the semantics of C and L_1. In other translations from one intermediate language L_i to another intermediate language L_j, the number of allocated blocks increases slightly. The translation allocates indeed the blocks that correspond to the blocks of L_i but also other blocks that are built by the translation of long expressions made up of several variables and function calls.

Concerning the memory area that stores code, each translation of the compilation process computes information that need to be stored in memory. At the end of the process, all the information have been computed and the target code may be emitted. If for instance a translation from one intermediate language L_i to another intermediate language L_j occurs, the semantics of L_i allocates as many blocks as the dynamic semantics of L_j. However, the blocks allocated by the dynamic semantics of L_j are becoming bigger. For instance, the return address of a called procedure is only known (and stored) at the end of the compilation process. As the translations do not preserve the contents of memory blocks, they may fail because they translate blocks into bigger blocks. Thus:

- During the compilation process, any translation fails when it translates a block into a bigger block.
- The execution of a translated program may fail, although the execution of the program does not fail.

With such a finite memory model, we prove the following correctness result for each translation: if the translation of a program does not fail, if that program terminates on some final memory state, and if the translated program also terminates, then it terminates on the same memory state. This property is weaker than the property we prove for an infinite memory model.

Instead of defining a more precise memory model, we intend to perform a static analysis that will track the amount of allocated memory for a given compilable program and compute an approximation of this amount if the control flow graph of the program is acyclic. We will then have to prove an equivalence result between the execution of the program and its execution in a stack discipline language where only one block is allocated. This will require the definition of such a language and the proof of semantic equivalence between this language and the corresponding language of the compiler.

4 From Formal Specifications to Code

This section gives an overview of the architecture of the *Coq* development. Figures 9 and 10 show the *Coq* modules that have been built in order to formally verify the memory model. *OCaml* modules have been automatically generated from them. The generated modules have the same architecture as the *Coq* modules. The *Coq* extraction mechanism removes the axioms and theorems, and more generally the terms of type `Prop`.

The abstract specification consists of the three signature modules of figure 9. They are declared with the keyword Module Type. The module called *MEM_PARAMS* collects the parameters of the memory model. These are *Coq* variables of type Set that *Coq* uses to type abstract specifications. They define the contents and the addressing of memory cells and are left abstract in the signature modules. The module called *MEM_LAYOUT* specifies the memory layout. It defines the functions and axioms that are detailed in figures 1 and 2. These definitions refer to unspecified types (*e.g. Tblock*) that are declared in a module called *MemP* of type *MEM_PARAMS*. The module called *MEM_OPS* specifies the memory management operations. It defines the functions and axioms that are detailed in figure 3.

The figure 10 shows the modules that implement the signature modules. For instance, the module *MEM_PARAMS_IMPL* implements the module *MEM_PARAMS* (see figures 4 and 5). The module called *MAKE_MEM_LAYOUT* is the functor that builds a module of type *MEM_LAYOUT* from a module of type *MEM_PARAMS*. All axioms that have been defined in the signature modules are proved in the implementation modules (thus becoming theorems). For instance, figure 10 shows the proof script of the theorem called valid_not_valid_diff. This is a very simple proof script that consists of a few *Coq* tactics. In this example, the proof script unfolds the definitions and prove by contradiction that *b* can not be equal to *b'*. More generally, these tactics can be user defined and correspond to the steps

```
Module Type MEM_PARAMS.
  Parameters Tchunk, Tofs, Tcell, Tvalue: Set.
  ...
End MEM_PARAMS.

Module Type MEM_LAYOUT.
  Declare Module MemP : MEM_PARAMS.

  Record Tblockdata := {high: Tblock ; low: Tblock ; contents: Map (Tblock, Tcell)}.
  ...
  Record Tmem := {memdata: Map (Tblock, Tblockdata); ... }.

  Definition valid_data_block (m:Tmem)(b:Tblock) := ∃ v, m.memdata(b) = Some v.

  Axiom valid_not_valid_diff:
    ∀ m b b', valid_data_block m b ∧ ¬(valid_data_block m b') ⇒ b ≠ b'.
End MEM_LAYOUT.

Module Type MEM_OPS.
  Declare Module MemP: MEM_PARAMS.
  Declare Module MemL: MEM_LAYOUT.

  Parameter load: Tchunk → Tmem → Tblock → Tofs → option Tvalue.
  ...

  Axiom loaded_block_is_valid:
    ∀ chunk m b ofs v, load chunk m b ofs = Some v ⇒ valid_data_block m b.
  ...

End MEM_OPS.
```

Fig. 9. Architecture of the specification (signature modules)

that would be used in a hand proof. They are reused to prove interactively the theorems.

Our memory model consists of several thousands lines of *Coq* specifications and proofs. The compilable *OCaml* modules that have been automatically extracted from the *Coq* specifications implement the operations that manage the memory.

5 Related Work

Several low-level memory models (often called architecture-centric models) have been defined. They are dedicated to hardware architectures and study the impact of features such as write buffers or caches, especially in multiprocessor systems. For instance, [22] uses a term rewriting system to define a memory model that decomposes load and store operations into finer-grain operations. This model formalises the notions of data replication and instruction reordering. It aims as defining the legal behaviours of a distributed shared-memory system that relies on execution trace of memory accesses. These memory models are lower-level

```
Module MAKE_MEM_LAYOUT (P: MEM_PARAMS)
        <: MEM_LAYOUT with Module MemP := P.
  ...
  Theorem valid_not_valid_diff:
      ∀ m b b', valid_data_block m b ∧ ¬(valid_data_block m b') ⇒ b ≠ b'.
  Proof. intros; red; intros; subst b; contradiction. Qed.
End MAKE_MEM_LAYOUT.

Module MEM_PARAMS_IMPL <: MEM_PARAMS.
  Definition Tblock:= ℤ .
  Inductive Tchunk := Mint1signed    | Mint1unsigned    | ...
  ...
End MEM_PARAMS_IMPL.

Module MEM_LAYOUT_IMPL <: MEM_LAYOUT :=
  MAKE_MEM_LAYOUT MEM_PARAMS_IMPL.

Module MEM_OPS_IMPL <: MEM_OPS.
  Module MemP := MEM_PARAMS_IMPL.
  Module MemL := MEM_LAYOUT_IMPL.

  Definition load (chunk: Tchunk) (m: Tmem) (b:Tblock) (ofs: Tofs) : option Tvalue
     := ...

  Theorem loaded_block_is_valid: ∀ chunk m b ofs v,
    load chunk m b ofs = Some v ⇒ valid_data_block m b.
  Proof. ... Qed.

  Theorem load_store_same:   ∀ chunk m1 m2 b1 ofs1 v,
    store chunk m1 b1 ofs1 v = Some m2 ⇒
        load chunk m2 b1 ofs1 = Some (load_result chunk v).
  Proof. ... Qed.
End MEM_OPS_IMPL.
```

Fig. 10. Architecture of the specification (implementation modules)

than ours (thus relying on a very different representation of memory) and are not dedicated to *C*-like languages.

Other research has concentrated on the formalisation of properties of programs that manipulate recursive data structures defined by pointers. New logics that capture common storage invariants have also been defined in order to facilitate and automate the proof of properties about pointers. These logics are based on separation logic [5], an extension of Hoare logic where assertions may refer to pointer expressions in a more concise and meaningful way. Two operators facilitate the expression of memory properties in separation logic: a separative conjunction allows one to express the separation of one piece of memory with respect to another, a separating implication allows one to introduce hypotheses about the memory layout. The definition of a refinement calculus for the separation logic is currently investigated [16]. In the near future, separation logic should be implemented, as is Hoare logic in tools dedicated to the B method.

Some ideas of separation logic have been formalised in *Isabelle/HOL* in order to verify the correctness of *Java* programs with pointers [15]. [9] presents a tool for formally proving that a *C* program is free of null pointer dereferencing and out-of-bounds array access. Some of our properties of memory management operations are also stated in [15] and [9].

Another way to prove properties about programs involving pointers is to define type systems that enable compilers to detect errors in programs. Some type systems are dedicated to a specific part of a compiler (*e.g.* assembly code [8]). Type systems for memory management have been applied for low-level memory management [24]. For instance, typed region systems where each memory location has an intended type and an actual type, have been defined to verify garbage collectors.

Much work has been done on verifying the complete correctness of a compiler. [11] and [3] use refinement as a compilation model. In the former, a refinement calculus is defined to support the compilation of programs written in an idealised high-level language into the .NET assembler. The aim of this work is to refine the whole compilation process and this approach is not automated by tools. The latter uses a term rewriting system to reduce programs into normal forms representing target programs.

The translation validation approach [18, 19, 10, 20, 21] aims at validating every run of the compiler, producing a formal proof that the produced target code is a correct implementation of the source code. This approach is based on program checking and on static analysis. It has been applied a lot for validating a variety of compiler optimizations, with a recent focus on loop transformations [27]. In the proof carrying code approach [17, 2, 12], the compiler is rewritten in a certifying compiler that produces both a compiled code and a proof term of some properties (called safety rules) to verify, that have been added in the source program. Safety rules are written in first-order predicate logic extended with predicates for type safety and low-level memory safety. Many specialised type systems have been used in this approach tat has been extensively applied to *Java* bytecode certification.

Our work belongs to a project that investigates the feasibility of formally verifying the correctness of a C-like compiler itself. The goal is to write the compiler directly in the *Coq* specification language. Other projects that develop machine-checked proofs of compiler correctness focus on data flow analyses and other compiler transformations [6, 23]. They do not require a memory model as precise as ours.

6 Conclusion

This paper has presented a formalisation and a verification in *Coq* of a memory model for C-like languages. Thanks to the use of *Coq* modules, this formalisation has been specified at two levels of abstraction. Two concrete specifications have been implemented from an abstract specification. They describe an infinite memory and a finite memory. Both memory models have a similar behaviour except in the case of failure of the allocation of memory blocks. A significant part of the specifications and correctness proofs have been factored out through the use of modules. The memory model has been implemented as part of an ongoing certification of a moderately-optimising C compiler. This compiler relies on 7 different languages whose formal semantics refer to the memory model, and on transformations that require extensive reasoning over the memory model. Many properties have been proved and certified programs have been synthesised from the formalisation.

A limitation of our compiler is that the correctness proofs of the transformations use simulation lemmas that apply only when every statement of the source code is mapped to zero, one or several statements of the transformed code. This is not sufficient to prove the correctness of more sophisticated optimisations such as code motion, lifting of loop-invariant computations or instruction scheduling, where computations occur in a different order in the source and transformed code. Because of this limitation, we envision to define a notion of equivalence between memory states and to perform these optimisations on a higher-level intermediate language, whose big-step semantics make it easier to reorder computations without worrying about intermediate computational states that are not equivalent.

Another current focus is the formalisation of non-terminating programs. The languages of our compiler are defined by big-step semantics that hide non-termination of programs. Our correctness proof states that any source program that terminates on some final memory state is compiled into a program that also terminates, produces the same memory state and calls the same functions in the same contexts. Previous experiments in the writing of small-step semantics showed us that they are not adapted for proving on machine properties such as semantic equivalence between languages. We intend to define semantics that collect more information than big-step semantics but that are not as concrete as small-step semantics.

Acknowledgements

We would like to thank C.Dubois and P.Letouzey for fruitful discussions about this work.

References

1. The Coq proof assistant. http://coq.inria.fr.
2. A.W. Appel. Foundational proof-carrying code. In *IEEE Symp. on Logic in Computer Science (LICS)*, page 247, Washington, DC, USA, June 2001.
3. A.Sampaio. *An algebraic approach to compiler design*, volume 4 of *AMAST series in computing*. World Scientific, 1997.
4. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.
5. R. Bornat. Proving pointer programs in Hoare logic. In *5th Conf. on Mathematics of Program Construction*, pages 102–126. Springer-Verlag, 2000.
6. D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a Data Flow Analyser in Constructive Logic. In *Proc. of Europ. Symp. on Programming (ESOP'04)*, number 2986 in Lecture Notes in Computer Science, pages 385–400. 2004.
7. J. Chrząszcz. *Modules in Type Theory with Generative Definitions*. PhD thesis, Warsaw Univerity and University of Paris-Sud, January 2004.
8. D.Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *Int. Conf. on Functional Programming (ICFP)*, pages 175–188, Snowbird, USA, September 2004.
9. J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *6th Int. Conf. on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, November 2004. Springer-Verlag.
10. G. Goos and W. Zimmermann. Verification of compilers. In *Correct System Design, Recent Insight and Advances,* , pages 201–230, London, UK, 1999. Springer-Verlag.
11. G.Watson. Compilation of refinement for a practical language. In *5th Int. Conf. on Formal Engineering Methods (ICFEM)*, volume 2885 of *Lecture Notes in Computer Science*, Singapore, November 2003. Springer-Verlag.
12. N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. *Journal of Automated Reasoning*, 31(3-4):191–229, September 2003.
13. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. 2005. draft, submitted for publication.
14. P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.
15. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In *Automated Deduction (CADE-19)*, volume 2741 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2003.
16. I. Mijajlovic and N. Torp-Smith. Refinement in separation context. In *Second workshop on semantics, program anlysis and computing analysis for memory management (SPACE)*, Venice, Italy, January 2004.
17. G. Necula. Proof carrying code. In *Proc. of Principles Of Progamming Languages Conf. (POPL)*, January 1997.
18. G. Necula. Translation validation for an optimizing compiler. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 83–95, 2000.

19. A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proc. of the 4th Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 151–166, London, UK, 1998. Springer-Verlag.
20. M. Rinard and D. Marinov. Credible compilation with pointers. In *Workshop on Run-Time Result Verification (RTRV)*, Trento, Italy, July 1999.
21. X. Rival. Symbolic transfer function-based approaches to certified compilation. In *Principles Of Progamming Languages Conf. (POPL)*, pages 1–13. 2004.
22. X. Shen, Arvind, and L. Rudolph. Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers. In *ISCA '99: 26th symposium on Computer architecture*, pages 150–161, Washington, DC, USA, 1999.
23. S.Lerner, T.Millstein, E.Rice, and C.Chambers. Automated soundness proofs for dataflow analyses and transformations. In *Principles Of Progamming Languages Conf. (POPL)*, Long Beach, USA, 2005.
24. S.Monnier. Typed regions. In *workshop on semantics, program anlysis and computing analysis for memory management (SPACE)*, Venice, Italy, January 2004.
25. R.D. Tennent and D.R. Ghica and. Abstract models of storage. *Higher-Order and Symbolic Computation*, 13(1/2):119–129, 2000.
26. D. Walker. Stacks, heaps and regions: one logic to bind them. In *Second workshop on semantics, program anlysis and computing analysis for memory management (SPACE)*, Venice, Italy, January 2004. invited talk.
27. Y.Hu, C.Barrett, B.Goldberg, and A. Pnueli. Validating more loop optimizations. In *Workshop on Compiler Optimization Meets Compiler Verification (COCV)*, Edinburgh, UK, 2005.