HAL
open science

The CompCert Memory Model, Version 2

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, Gordon Stewart

To cite this version:

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, Gordon Stewart. The CompCert Memory Model, Version 2. [Research Report] RR-7987, INRIA. 2012, pp.26. hal-00703441

HAL Id: hal-00703441

https://inria.hal.science/hal-00703441

Submitted on 1 Jun 2012

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

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

The CompCert Memory Model, Version 2

Xavier Leroy, Andrew W. Appel, Sandrine Blazy, Gordon Stewart

RESEARCH REPORT
N° 7987
June 2012
Project-Teams Gallium and Celtique

ISSN 0249-6399 ISRN INRIA/RR--7987--FR+ENG

The CompCert Memory Model, Version 2

Xavier Leroy[*][†], Andrew W. Appel[‡] [§], Sandrine Blazy[¶][†],
Gordon Stewart[‡] [§]

Project-Teams Gallium and Celtique

Research Report n° 7987 — June 2012 — 26 pages

Abstract: A memory model is an important component of the formal semantics of imperative programming languages: it specifies the behavior of operations over memory states, such as reads and writes. The formally verified CompCert C compiler uses a sophisticated memory model that is shared between the semantics of its source language (the CompCert subset of C) and intermediate languages. The algebraic properties of this memory model play an important role in the proofs of semantic preservation for the compiler. The initial design of the CompCert memory model is described in an article by Leroy and Blazy (J. Autom. Reasoning 2008). The present research report describes version 2 of this memory model, improving over the main limitations of version 1. The first improvement is to expose the byte-level, in-memory representation of integers and floats, while preserving desirable opaqueness properties of pointer values. The second improvement is the integration of a fine-grained mechanism of permissions (access rights), which supports more aggressive optimizations over read-only data, and paves the way towards shared-memory, data-race-free concurrency in the style of Appel's Verified Software Toolchain project.

Key-words: Memory models, formal semantics, verified compilation, CompCert

[*] INRIA Paris-Rocquencourt, project-team Gallium
[†] Supported in part by Agence Nationale de la Recherche, project Arpège U3CAT, grant ANR 08-SEGI-021
[‡] Princeton University
[§] Supported in part by the Air Force Office of Scientific Research (grant FA9550-09-1-0138) and the National Science Foundation (grant CNS-0910448).
[¶] Université de Rennes 1, IRISA, project-team Celtique

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Le modèle mémoire CompCert version 2

Résumé : Le modèle mémoire est un composant important de la sémantique formelle d'un langage impératif de programmation: il spécifie le comportement des opérations sur les états mémoire, tels que les lectures et les écritures. Le compilateur formellement vérifié CompCert C utilise un modèle mémoire élaboré, qu'il partage entre les sémantiques de son langage source (le sous-ensemble CompCert de C) et de ses langages intermédiaires. Les propriétés algébriques de ce modèle mémoire jouent un rôle important dans les preuves de préservation sémantique du compilateur. La première version du modèle mémoire CompCert est décrit dans un article de Leroy et Blazy (J. Autom. Reasoning 2008). Ce rapport de recherche décrit la version 2 de ce modèle mémoire, qui résout les principales limitations de la version 1. Première amélioration: il expose la représentation en mémoire, au niveau de l'octet, des entiers et des flottants, tout en préservant les propriétés utiles d'opacité des pointeurs. Seconde amélioration: il intègre un mécanisme de permissions (droits d'accès) à grain fin, qui autorise le compilateur à effectuer des optimisations plus agressives sur les données en lecture seule, et constitue un premier pas vers le parallélisme à mémoire partagée bien synchronisé, dans le style du projet Verified Software Toolchain d'Appel.

Mots-clés : Modèles mémoires, sémantiques formelles, compilation vérifiée, CompCert

1 Introduction

The imperative programming paradigm views programs as sequences of commands that incrementally update a *memory state*, also called *store*. A *memory model* is a specification of memory states and operations over memory, such as reads and writes. Such a memory model is a prerequisite to giving formal semantics to imperative programming languages, verifying properties of programs, and proving the correctness of program transformations.

For high-level, type-safe languages such as ML or the sequential fragment of Java, the memory model is simple and amounts to a finite map from abstract memory locations to the values they contain. At the other end of the complexity spectrum, we find memory models for shared-memory concurrent programs with data races and relaxed (non sequentially consistent) memory, where much effort is needed to capture the relaxations (e.g. reorderings of reads and writes) that are allowed and those that are guaranteed never to happen [1].

In this work, we focus on memory models for the C programming language and for compiler intermediate languages, in the sequential case but with planned extensions to data race-free concurrency. The C language and the intermediate languages we consider feature both low-level aspects such as pointers, pointer arithmetic, and nested objects, and high-level aspects such as separation and freshness guarantees. 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. A very abstract memory model, such as the popular Burstall-Bornat model [7, 6], can fail to account for desirable features of the languages we are interested in, such as casts between incompatible pointer types. A very concrete memory model, such as the hardware view of memory as an array of bytes indexed by addresses that are just machine integers [15], fails to enforce separation and freshness guarantees, and makes it impossible to prove the correctness of standard compiler passes and even of late, link-time or loading-time placement of code and data in memory.

In the context of the CompCert project [11], where we formally verify the correctness of a realistic, optimizing C compiler, we developed and formalized in the Coq proof assistant a memory model for the CompCert C dialect of the C programming language and for the various intermediate languages of the CompCert compiler. Version 1 of the CompCert memory model is described in details in an article by Leroy and Blazy [12] and summarized in section 2. In the years following this publication, several limitations of this "v1" memory model appeared, some related to low-level programming idioms used in embedded systems, others related to the extension of CompCert towards race-free concurrent programming as investigated in the Verified Software Toolchain project of Appel *et al* [2].

The limitations mentioned above and described in section 3 led us to refine the CompCert memory model in two directions. One is to expose the byte-level machine representation of integers and floating-point numbers, while keeping abstract the machine representation of pointers, as required to preserve crucial invariance properties of invariance by generalized renaming of block identifiers. The other direction is to add fine-grained *permissions*, also known as access rights, on every byte of the memory state, giving precise control on which memory operations are permitted on these bytes. For instance, the in-memory representation of a C string literal can be given read-only permissions, allowing reads but preventing writes and deallocation.

This new memory model was introduced in release 1.7 of CompCert, then further refined in release 1.11. This technical report documents the CompCert 1.11 memory model, which we call the CompCert memory model version 2.

2 The CompCert memory model, version 1

We first review version 1 of the CompCert memory model, referring the reader to Leroy and Blazy [12] for full details.

In this model, memory states m are collections of *blocks*, each block being an array of abstract bytes. (See figure 1.) Pointers are represented by pairs (b, i) of a block identifier b and a byte offset i within this block. Each block b has two integer bounds, `low_bound` m b and `high_bound` m b. Valid offsets within block b range between `low_bound` m b inclusive and `high_bound` m b exclusive. (The size of block b is therefore `high_bound` m b − `low_bound` m b bytes.)

Our semantics for CompCert C and the Clight intermediate language [4] associate a different block to every global variable of the program, to every addressable local variable of every active invocation of a function of the program, and to every invocation of `malloc`. For local variables, fresh blocks are allocated at function entry point and deallocated when the function returns.

Pointer arithmetic modifies the offset part of a pointer value, keeping its block identifier part unchanged:

$$(b, i) + n \stackrel{\text{def}}{=} (b, i + n)$$

As a consequence, blocks are separated by construction: from a pointer to block b, no amount of pointer arithmetic can create a pointer to block $b' \neq b$; pointer arithmetic can only create other pointers within block b, or illegal pointers outside b's bounds.

As an abstract data type, memory states are presented as a type `mem`, a constant `empty` : `mem` denoting the empty memory state, and the 4 operations

$$\begin{array}{rcl}
\texttt{alloc} & : & \texttt{mem} \to \texttt{Z} \to \texttt{Z} \to \texttt{mem} \times \texttt{block} \\
\texttt{free} & : & \texttt{mem} \to \texttt{block} \to \texttt{mem} \\
\texttt{load} & : & \texttt{mem} \to \texttt{memory_chunk} \to \texttt{block} \to \texttt{Z} \to \texttt{option val} \\
\texttt{store} & : & \texttt{mem} \to \texttt{memory_chunk} \to \texttt{block} \to \texttt{Z} \to \texttt{val} \to \texttt{option mem}
\end{array}$$

`alloc` m l h allocates a fresh block of size $h - l$ bytes, with low bound l and high bound h. It returns the updated memory state and the block identifier for the fresh block.

`free` m b deallocates the block b, returning the updated memory state in which block b has bounds $(0, 0)$ and can therefore no longer be read from or written to.

`store` m τ b i v stores value v with type τ in block b at offset i. Values are the discriminated union of 32-bit machine integers, 64-bit floats, pointer values, and a special `Vundef` value denoting an unknown value:

$v ::= \texttt{Vint}(i) \mid \texttt{Vfloat}(f) \mid \texttt{Vptr}(b, i) \mid \texttt{Vundef}$

Memory types τ indicate the size, type and signedness of the value being stored:

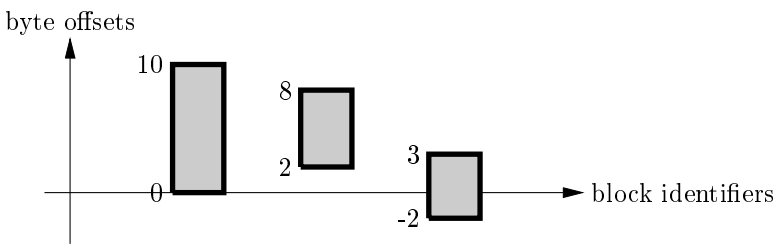


Figure 1: General shape of memory states

$\tau ::=$ `Mint8signed` | `Mint8unsigned`	8-bit integers
| `Mint16signed` | `Mint16unsigned`	16-bit integers
| `Mint32`	32-bit integers or pointers
| `Mfloat32`	32-bit, single-precision floats
| `Mfloat64`	64-bit, double-precision floats

Each type τ comes with a size $|\tau|$ in bytes and a natural alignment $\langle\tau\rangle$.

As shown by the `option mem` return type, memory stores can fail because they perform bounds and alignment checking. The store succeeds and returns `Some` m' (where m' is the updated memory state) if and only if bounds and alignment constraints are respected:

$$\langle\tau\rangle \text{ divides } i \ \wedge\ \texttt{low_bound}\ m\ b \leq i \ \wedge\ i + |\tau| \leq \texttt{high_bound}\ m\ b$$

Otherwise, `store` returns `None`.

`load` $m\ \tau\ b\ i$ reads a value of type τ from block b at starting offset i. The load succeeds and returns `Some` v (where v is the value read) if and only if bounds and alignments constraints are respected, as in the case of `store`. Otherwise, `None` is returned.

The following "good variable" laws characterize most of the semantics of these 4 memory operations.

Load after alloc: if `alloc` $m\ l\ h = (m', b)$,

- `load` $m'\ \tau'\ b'\ i' =$ `load` $m\ \tau'\ b'\ i'$ if $b' \neq b$
- If `load` $m\ \tau\ b\ i =$ `Some` v, then $v =$ `undef`

Load after free: if `free` $m\ b = m'$,

- `load` $m'\ \tau'\ b'\ i' =$ `load` $m\ \tau'\ b'\ i'$ if $b' \neq b$
- `load` $m\ \tau\ b\ i =$ `None`.

Load after store: if `store` $m\ \tau\ b\ i\ v =$ `Some` m',

- Disjoint case: `load` $m'\ \tau'\ b'\ i' =$ `load` $m\ \tau'\ b'\ i'$ if $b' \neq b$ or $i' + |\tau'| \leq i$ or $i + |\tau| \leq i'$
- Compatible case: `load` $m'\ \tau'\ b\ i =$ `Some(convert` $\tau'\ v$) if $|\tau'| = |\tau|$
- Incompatible case: if `load` $m'\ \tau'\ b\ i =$ `Some` v' and $|\tau'| \neq |\tau|$, then $v' =$ `undef`
- Overlapping case: if `load` $m'\ \tau'\ b\ i' =$ `Some` v' and $i' \neq i$ and $i' + |\tau'| > i$ and $i + |\tau| > i'$, then $v =$ `undef`

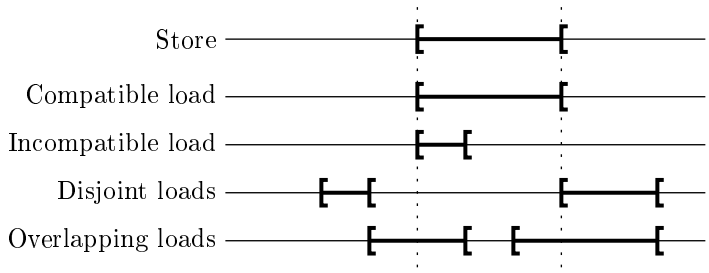


Figure 2: The 4 cases of the load-after-store algebraic laws

The 4 cases of the "load after store" laws are depicted in figure 2. The disjoint case corresponds to a load outside the memory area affected by the store. In the compatible case, we load exactly from the memory area affected by the store, in which case we obtain the value just stored, after conversion to the destination type τ':

$$
\begin{aligned}
\texttt{convert}\ (\texttt{Vint}(n))\ \texttt{Mint8unsigned} &= \texttt{Vint}(\text{8-bit zero extension of } n) \\
\texttt{convert}\ (\texttt{Vint}(n))\ \texttt{Mint8signed} &= \texttt{Vint}(\text{8-bit sign extension of } n) \\
\texttt{convert}\ (\texttt{Vint}(n))\ \texttt{Mint16unsigned} &= \texttt{Vint}(\text{16-bit zero extension of } n) \\
\texttt{convert}\ (\texttt{Vint}(n))\ \texttt{Mint16signed} &= \texttt{Vint}(\text{16-bit sign extension of } n) \\
\texttt{convert}\ (\texttt{Vint}(n))\ \texttt{Mint32} &= \texttt{Vint}(n) \\
\texttt{convert}\ (\texttt{Vptr}(b,i))\ \texttt{Mint32} &= \texttt{Vptr}(b,i) \\
\texttt{convert}\ (\texttt{Vfloat}(f))\ \texttt{float32} &= \texttt{Vfloat}(f \text{ normalized to single precision}) \\
\texttt{convert}\ (\texttt{Vfloat}(f))\ \texttt{float64} &= \texttt{Vfloat}(f) \\
\texttt{convert}\ v\ \tau &= \texttt{Vundef} \quad \text{in all other cases}
\end{aligned}
$$

In the two remaining cases, "incompatible" and "overlapping", the bytes being read by the `load` include some but not all of the bytes written by the `store`, possibly combined with other bytes that were not affected by the `store`. To specify the result of the `load` in these two cases, we would need to expose the byte-level representation of values in memory: are integers stored in big-endian or little-endian representation? what is the byte-level encoding of floats? etc. This is something we elected not to do in version 1 of the memory model. Therefore, we just say that the value read is `Vundef`, as if we were reading from an uninitialized memory area. This is one of the design decisions that we revisited in version 2 of the memory model.

3 Assessment of the memory model, version 1

3.1 Capability: Accounting for ISO C99 and popular C programming idioms

The CompCert memory model version 1 correctly models the memory behavior of C programs that conform to the ISO C99 standard [10]. As specified in section 6.5 of the C99 standard, a C "object" (memory-resident data) has an effective type, which is either its declared static type, if any, or the type of the latest assignment into this object, if it has no declared static type. A conformant program always accesses an object through an l-value whose type is compatible with that of the effective type of the object. Compatibility includes addition or removal of qualifiers, as well as changes in signedness. This corresponds to the "compatible" case of the load-after-store laws:

$$\text{if } \texttt{store}\ m\ \tau\ b\ i\ v = \texttt{Some}\ m' \text{ and } |\tau'| = |\tau| \text{ then } \texttt{load}\ m'\ \tau'\ b\ i = \texttt{Some}(\texttt{convert}(\tau', v))$$

Indeed, two C types t, t' that are compatible in the sense of the C99 standard encode into CompCert memory chunks τ, τ' that have the same size (and differ only in signedness). Moreover, the C semantics guarantee that the value v being stored with type t has been normalized (casted) to type t before storing. In this case, CompCert's conversion-at-load-time $\texttt{convert}(\tau', v)$ behaves like the C type cast $(t')\ v$.

Besides standard-conformant C programs, the CompCert memory model can give meaning to several popular C programming idioms that have undefined behavior according to the C standards. For example, in CompCert, the representation of pointer values is independent of

their static pointer types. Therefore, casting from any pointer type to any other pointer type and then back to the original pointer type is well defined and behaves like the identity function:

```
int x = 3;
*((int *) (double *) &x) = 4;    // equivalent to "x = 4;"
```

For another example, CompCert memory blocks are accessed at byte offsets within a block. The CompCert C and Clight semantics compute the byte offsets corresponding to array elements or struct fields before performing memory accesses [4]. This makes it possible to give semantics to non-conformant programs that access elements of arrays or structs via nonstandard casts or pointer arithmetic. For example:

```
struct { int x, y, z; } s;
s.y = 42;
((int *) &s)[1] = 42;
*((int *) ((char *) &s + sizeof(int))) = 42;
```

All three assignments above are well defined in CompCert C and have the same semantics, namely storing the integer 42 at offset 4 in the block associated with variable `s`.

The same tolerance applies to non-discriminated unions. Consider:

```
union point3d {
    struct { double x, y, z; } s;
    double d[3];
};
```

For any object `p` of type `union point3d`, its three coordinates can be accessed indifferently as `p.s.x`, `p.s.y`, `p.s.z` or as `p.d[0]`, `p.d[1]`, `p.d[2]`.

3.2 Limitation: No access to in-memory data representations

Systems or library C codes often cannot be written in standard-conformant C because they need to operate over the in-memory representation of data, at the level of individual bytes or bits. For example, changing the endianness of an integer (converting it from little-endian to big-endian representation, or conversely) is often written as follows:

```
unsigned int bswap(unsigned int x)
{
    union { unsigned int i; char c[4]; } src, dst;
    int n;
    src.i = x;
    dst.c[3] = src.c[0]; dst.c[2] = src.c[1];
    dst.c[1] = src.c[2]; dst.c[0] = src.c[3];
    return dst.i;
}
```

In this example, the memory objects for `src` and `dst` are accessed simultaneously with types `int` and `char`. This is not supported by version 1 of the CompCert memory model: we fall in the "incompatible" and "overlapping" cases of the load-after-store laws, therefore `src.c[0]`, ..., `src.c[3]` read as `Vundef` values, and likewise `dst.i` reads as `Vundef` instead of the byte-reversed value of `x` as expected.

The example above is not too serious because it can be rewritten into standard-conformant code:

```
unsigned int bswap(unsigned int x)
{
    return (x & 0xFF) << 24 | (x & 0xFF00) << 8
         | (x & 0xFF0000) >> 8 | (x & 0xFF000000) >> 24;
}
```

More delicate examples arise in floating-point libraries that need to exploit the IEEE 754 bit-level representation of floating-point numbers to implement basic float operations. For instance, taking the absolute value of a single-precision IEEE 754 float amounts to clearing the top bit of its representation:

```
float fabs_single(float x)
{
    union { float f; unsigned int i; } u;
    u.f = x;
    u.i = u.i & 0x7FFFFFFF;
    return u.f;
}
```

Giving semantics to this function using the CompCert memory model version 1, we obtain that it always returns `Vundef` instead of the expected float result: the read `u.i` after the write `u.f` falls in the "incompatible" case of the load-after-store laws.

Sometimes, "bit surgery" over floating-point numbers must be performed by the compiler itself, to implement primitive C operations for which the microprocessor provides no dedicated instructions. For example, the PowerPC 32 bits architecture lacks an instruction to convert a 32-bit integer to a double-precision float. This conversion must be implemented by machine code equivalent to the following C code:

```
double double_of_signed_int(int x)
{
    union { double d; unsigned int i[2]; } a, b;

    a.i[0] = 0x43300000;  a.i[1] = 0x80000000;
    b.i[0] = 0x43300000;  b.i[1] = 0x80000000 + x;
    return b.d - a.d;
}
```

This code exploits not only the fact that the PowerPC is a big-endian architecture, but also the bit-level IEEE 754 representation: the bit pattern of `a`, namely, `0x4330000080000000` represents the float $2^{52} + 2^{31}$, and the bit pattern of `b` represents the float $2^{52} + 2^{31} + (\texttt{double})\texttt{x}$; moreover, the floating-point subtraction between these two floats is exact, resulting in $(\texttt{double})\texttt{x}$. Again, version 1 of the CompCert memory model fails to give the intended semantics to this code, predicting an `Vundef` result instead.

Finally, some library functions work over byte-level data representations in a highly-portable (but not standard-conformant) manner. This is the case for the following naive implementation of the `memcpy` function from the C standard library:

```
void * memcpy(void * dest, const void * src, size_t n)
{
    for (i = 0; i < n; i++)
        ((char *) dest)[i] = ((const char *) src)[i];
    return dest;
}
```

According to the CompCert memory model version 1, this `memcpy` works as intended if it is passed arrays of `char` (signed or unsigned), or other compound types consisting only of `char` fields. Otherwise, for example if `src` points to an array of `int` or `double`, the read `((const char *) src)[i]` returns `Vundef`, and the destination block `dest` is filled with `Vundef` values.

3.3 Capability: Invariance by memory transformations

In the implementation of the memory model, block identifiers are integers and are assigned consecutively at each `alloc` operation. However, the operations of the memory model and the CompCert C formal semantics are insensitive to this particular choice of block identifiers. For instance, the CompCert C semantics, following the C99 standard, enables programs to test whether two block identifiers are equal (using the `==` pointer comparison), but not whether one identifier was allocated before another one (the `<` comparison is undefined between pointers designating different blocks). Consider two programs that are identical except for the order of definition of some variables:

```
int x = 10;                 int y = 20;
int y = 20;                 int x = 10;
/* some code */             /* same code */
```

When executed according to the CompCert C semantics, the program on the left will bind `x` to (say) block identifier 1 and `y` to block 2, while the program on the right will bind `x` to block 2 and `y` to block 1. Nonetheless, the observable behaviors of the two programs are the same, because both the memory model and the CompCert C semantics are invariant by a *renaming* of block identifiers.

Invariance properties stronger than renamings are necessary to prove the correctness of CompCert's compiler passes. Here are two examples of compilation passes that modify the memory layout of the program in nontrivial ways:

1. In an early simplification pass, producing Cminor intermediate code, local scalar variables whose addresses are never taken (using the `&` operator) are "pulled out of memory" and made to reside in a variable environment separate from the memory state. (This enables much more aggressive optimizations on uses of these variables.) Moreover, to simplify the semantics of Cminor and later intermediate languages, the remaining local variables are packed together as sub-blocks of a single memory block representing the stack frame of the function.

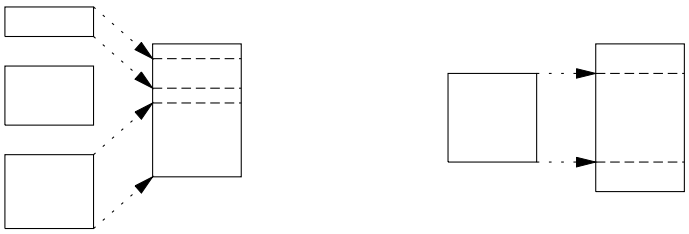
2. In the "stacking" pass performed after register allocation, local variables that could not be allocated to register are "spilled" to memory locations within the stack frame of the current function. This stack frame, therefore, needs to be extended to make room for spilled variables.

The effect of these two program transformations on the memory states are depicted in figure 3.

To reason about such transformations, CompCert introduces the notion of *memory injections*, which are a generalization of renamings of block identifiers. A memory injection is a function F with type

$$F : \texttt{block} \to \texttt{option}(\texttt{block} \times \texttt{Z})$$

Let b be a block identifier in the memory state of the original program. $F(b) = \texttt{None}$ means that this block was "pulled out of memory" by the program transformation. $F(b) = \texttt{Some}(b', \delta)$ means that this block is mapped to a sub-block of block b' in the memory state of the transformed program, said sub-block starting at offset δ.



Memory injections (C#minor to Cminor) — Memory extensions (spilling)

Figure 3: Transformations over memory states in the Compcert compiler

A memory injection F induces a relation $F \vdash v_1 \hookrightarrow v_2$ between the values v_1 of the original program and the values v_2 of the transformed program. This relation corresponds to relocating pointer values as specified by F. It also enables `Vundef` values in the original program to be replaced by more defined values in the transformed program.

$$F \vdash \texttt{Vundef} \hookrightarrow v_2 \qquad F \vdash \texttt{Vint}(n) \hookrightarrow \texttt{Vint}(n) \qquad F \vdash \texttt{Vfloat}(n) \hookrightarrow \texttt{Vfloat}(n)$$

$$\frac{F(b_1) = \texttt{Some}(b_2, \delta) \qquad i_2 = i_1 + \delta}{F \vdash \texttt{Vptr}(b_1, i_1) \hookrightarrow \texttt{Vptr}(b_2, i_2)}$$

Likewise, a memory injection F induces a relation $F \vdash m_1 \mapsto m_2$ between the memory states m_1 of the original program and m_2 of the transformed program. In version 1 of the CompCert memory model, this relation is defined as "every load in m_1 from a mapped block is simulated by a load in m_2 from the image of this block":

$$\begin{aligned} & F(b_1) = \texttt{Some}(b_2, \delta) \ \wedge \ \texttt{load}\ \tau\ m_1\ b_1\ i = \texttt{Some}(v_1) \\ & \quad \Rightarrow \ \exists v_2, \texttt{load}\ \tau\ m_2\ b_2\ (i + \delta) = \texttt{Some}(v_2) \ \wedge \ F \vdash v_1 \hookrightarrow v_2 \end{aligned}$$

As demonstrated by Leroy and Blazy [12, section 5], memory injections enjoy nice properties of commutation with the `store`, `alloc` and `free` operations of the memory model. These properties, in turn, support the proof of semantic preservation for the CompCert compiler passes that modify the memory behaviors of programs.

In conclusion, a crucial feature of the CompCert memory model version 1 is that block identifiers are kept relatively abstract and can be renamed, deleted or injected as sub-blocks of bigger blocks while preserving the observable behaviors of programs. Without this feature, several passes of the CompCert compiler could not be proved to preserve semantics.

3.4 Limitation: Coarse-grained access control

In version 1 of the memory model, a `load` or `store` operation succeeds as long as the accessed location is aligned and within the bounds of its enclosing block, and a `free` operation always succeeds, even if the given block was already freed. This behavior is too coarse in several situations that we now illustrate.

First, it should be the case that `free` fails if the given block has not been allocated before, or was already freed earlier. In this way, programs that perform double-free errors have undefined semantics, like they should. Attempts to free global variables should also fail, for similar reasons.

Second, memory blocks corresponding to `const` variables in C or to string literals should be marked read-only, so that a `store` into one of these blocks always fails. Besides making the semantics of CompCert C closer to the C standards, reflecting `const`-ness into the memory model in this way supports interesting optimizations such as constant propagation of `const` global variables:

```
const int cst = 4;                          const int cst = 4;
                                   →
int f(x) { return x * cst; }                int f(x) { return x << 2; }
```

Third, fine-grained access control over memory locations is also useful to extend CompCert towards data race-free concurrent programs, as proposed by Appel *et al* in the Verified Software Toolchain project [2, 3, 9].

In their approach, data races are avoided by a locking discipline enforced via a concurrent separation logic [14, 9]. Each area of shared memory is logically associated with a lock through concurrent separation logic formulas. When a thread releases an exclusive lock, the calling thread loses all access rights on the associated memory area. If, later, this lock is reacquired, the calling thread recovers these rights. Since their concurrent separation logic supports fractional permissions, it is also possible for a thread to temporarily abandon write rights, giving other threads the right to read (but not write) to this memory area, while retaining read rights for itself.

At the source language level, the access rights mentioned above are implied and enforced by the separation logic. The CompCert compiler must, then, guarantee that these access rights are respected during compilation. A typical violation would be for CompCert to move a load or store before a lock acquisition or after a lock release. One way to prove that this does not happen would be to apply the separation logic discipline to all intermediate languages and compilation passes of CompCert. A much simpler approach, which we follow in version 2 of the CompCert memory model, is to equip the memory model with a notion of fine-grained, per-byte permissions, governing for instance whether a byte can be written to. Concurrent operations such as `lock` and `unlock` are, then, modeled as changing the memory permissions as well as memory contents in an unpredictable manner, under control of an oracle external to the semantics. This suffices to prevent the compiler from moving memory accesses across `lock` and `unlock` operations.

The changes to memory permissions that occur at external function calls—in what appears to CompCert to be an "unpredictable" manner—can be reasoned about in a logic *external* to CompCert and its memory model. That logic might use fractional or token-based permission models [8] to prove race freedom in a very predictable way. These complex permission-models do not need to be completely reified into the CompCert memory model; instead, a summary of their effects can be described by the abstract permissions that we will show in section 4.2.

4 The CompCert memory model, version 2

We now describe version 2 of the CompCert memory model, which enhances version 1 to address the limitations described in sections 3.2 and 3.4, to the extent that the good properties described in sections 3.1 and 3.3 still hold. The main changes, described in more details in the following sections, are:

- Exposing the byte-level, in-memory representation of integers and floats, while keeping that of pointers abstract (section 4.3).
- Introducing fine-grained, byte-level permissions in replacement for memory bounds (section 4.2).

- Adding new operations over memory states: `loadbytes`, `storebytes`, and `drop_perm` (section 4.1).

4.1 Operations

In the new memory model, memory states are presented as a type `mem`, a constant `empty` : `mem` denoting the empty memory state, and the 7 operations

$$\begin{array}{rcl}
\texttt{alloc} & : & \texttt{mem} \to \texttt{Z} \to \texttt{Z} \to \texttt{mem} \times \texttt{block} \\
\texttt{free} & : & \texttt{mem} \to \texttt{block} \to \texttt{Z} \to \texttt{Z} \to \texttt{option mem} \\
\texttt{load} & : & \texttt{mem} \to \texttt{memory_chunk} \to \texttt{block} \to \texttt{Z} \to \texttt{option val} \\
\texttt{store} & : & \texttt{mem} \to \texttt{memory_chunk} \to \texttt{block} \to \texttt{Z} \to \texttt{val} \to \texttt{option mem} \\
\texttt{loadbytes} & : & \texttt{mem} \to \texttt{block} \to \texttt{Z} \to \texttt{Z} \to \texttt{option(list memval)} \\
\texttt{storebytes} & : & \texttt{mem} \to \texttt{block} \to \texttt{Z} \to \texttt{list memval} \to \texttt{option mem} \\
\texttt{drop_perm} & : & \texttt{mem} \to \texttt{block} \to \texttt{Z} \to \texttt{Z} \to \texttt{permission} \to \texttt{option mem}
\end{array}$$

`alloc`, `load` and `store` are as in version 1 of the model. The `free` operation, `free` $m\ b\ l\ h$, no longer frees the whole block b, but rather the range of offsets $[l, h)$ within block b. This change simplifies the definition of a separation logic on top of the memory model. It also makes it possible to reduce the size of a block after allocation, and to "punch holes" within a block, two possibilities that CompCert does not exercise currently. Another change is that `free` can now fail, typically if the locations to be freed have been freed already.

Three new operations were added. `loadbytes` and `storebytes` are similar to `load` and `store`, but instead of reading or writing a value, they read or write a list of byte contents (type `memval`, explained in section 4.3 below). `loadbytes` and `storebytes` are useful to give semantics to block copy operations such as `memcpy`, and also to reason over byte-level, in-memory representation of data.

Finally, `drop_perm` $m\ b\ l\ h\ p$ lowers the permissions (access rights) over locations $(b, l), \ldots, (b, h-1)$, setting them to p. Permissions are further explained in section 4.2 below. A typical use of `drop_perm` is to set to read-only a memory block corresponding to a `const` C variable, after it has been initialized.

4.2 Permissions

Memory states associate permissions, or access rights, to every byte location. The various permissions are:

`Freeable`	full permissions: can compare, read, write, and free
`Writable`	can compare, read and write but not free
`Readable`	can compare and read but not write nor free
`Nonempty`	can only compare

In the table above, "compare" refers to the ability of comparing a pointer to the given location with other pointers.[1]

Permissions are cumulative: having permission p implies having all permissions $p' < p$, where the ordering on permissions is

$$\texttt{Nonempty} < \texttt{Readable} < \texttt{Writable} < \texttt{Freeable}$$

[1] In CompCert C as in the C standards, pointer comparison involving invalid pointers (e.g. pointers to freed locations) have undefined semantics. For the purpose of giving semantics to pointer comparisons, we take that a pointer $\texttt{Vptr}(b, i)$ is valid if the location (b, i) has at least `Nonempty` permission.

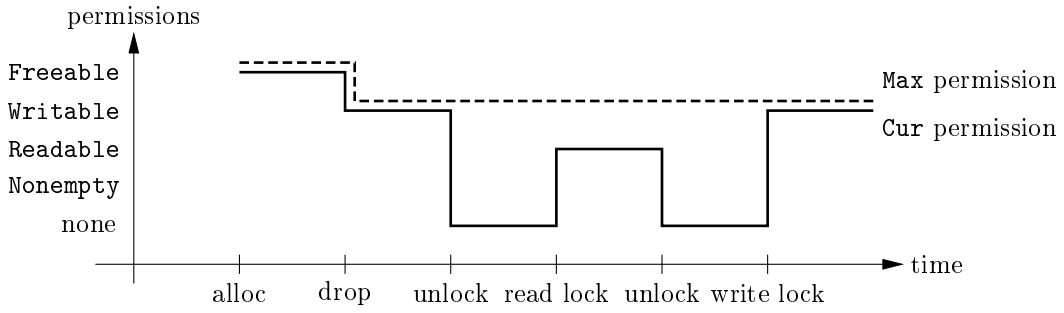


Figure 4: An example of evolutions of `Max` and `Cur` permissions for a location

It is possible for a location to have no permission at all. In this case, we say that the location is *empty*. This is typically the case for locations that have not been allocated yet, or have been freed already.

Every byte location is associated not to one, but to two permissions: the *current* permission and the *maximal* permission. At any time in the execution, the current permission is less than or equal to the maximal permission. The maximal permission evolves predictably throughout the lifetime of the location: when the location is allocated, it has maximal permission `Freeable`; this permission can later be lowered by a `drop_perm` operation; finally, freeing the location removes all its maximal permissions, making the location empty. Note that the maximal permission can only decrease once the location has been allocated. In contrast, the current permission can decrease or increase (without ever exceeding the maximal permission) during the lifetime of the location. For example, in the extension to shared memory concurrency outlined in section 3.4, an `unlock` operation temporarily drops current permissions, which can be recovered by a subsequent `lock` operation. Figure 4 illustrates this evolution of permissions throughout the lifetime of a location.

In the Coq formalization, the association of permissions to locations is exposed as a predicate:

$$\texttt{perm} : \texttt{mem} \to \texttt{block} \to \texttt{Z} \to \texttt{perm_kind} \to \texttt{permission} \to \texttt{Prop}$$

where `perm_kind` is the enumerated inductive type `Max | Perm`. The proposition `perm` $m\ b\ i\ k\ p$ holds if and only if in memory state m, location (b, i) has k-permission at least p. The cumulativity of permissions, and the fact that current permissions are never above maximal permissions, are expressed by the following implications:

$$\begin{aligned} \texttt{perm}\ m\ b\ i\ k\ p\ \wedge\ p' \leq p\ &\Rightarrow\ \texttt{perm}\ m\ b\ i\ k\ p' \\ \texttt{perm}\ m\ b\ i\ \texttt{Cur}\ p\ &\Rightarrow\ \texttt{perm}\ m\ b\ i\ \texttt{Max}\ p \end{aligned}$$

Instead of checking block offsets against block bounds, as in version 1 of the memory model, the `load` and `store` operations check that the accessed locations have current permissions at least `Readable`, resp. `Writable`. Likewise, the `free` and `drop_perm` operations check that the affected locations have current permissions at least `Freeable`. Defining

```
Definition range_perm (m: mem) (b: block) (lo hi: Z)
                      (k: perm_kind) (p: permission) : Prop :=
  forall ofs, lo <= ofs < hi -> perm m b ofs k p.
```

we have the following properties:

Operation...	succeeds if and only if...
`load` $m\ \tau\ b\ i$	`range_perm` $m\ b\ i\ (i + \lvert\tau\rvert)$ `Cur Readable`
`store` $m\ \tau\ b\ i\ v$	`range_perm` $m\ b\ i\ (i + \lvert\tau\rvert)$ `Cur Writable`
`free` $m\ b\ l\ h$	`range_perm` $m\ b\ l\ h$ `Cur Freeable`
`drop_perm` $m\ b\ l\ h\ p$	`range_perm` $m\ b\ l\ h$ `Cur Freeable`

Owing to the availability of per-byte permissions, it is no longer useful to associate low and high bounds to memory blocks. Version 2 of the memory model therefore removes the `low_bound` and `high_bound` functions of version 1. The main use of these functions in CompCert's proofs was to state that a location (b, i) is valid, i.e. already allocated but not yet freed, using the following definition:

$$(b, i) \text{ is valid in } m \stackrel{\text{def}}{=} \texttt{low_bound}\ m\ b \leq i < \texttt{high_bound}\ m\ b$$

Using version 2 of the model, the following alternate definition works just as well:

$$(b, i) \text{ is valid in } m \stackrel{\text{def}}{=} \texttt{perm}\ m\ b\ i\ \texttt{Max Nonempty}$$

Permissions are preserved by operations over memory states, with the following exceptions.

Operation	Effect on permissions
`alloc` $m\ l\ h = (m', b)$	locations $(b, l) \ldots (b, h-1)$ get `Freeable` permissions (`Max` and `Cur`)
`free` $m\ b\ l\ h$	locations $(b, l) \ldots (b, h-1)$ lose all permissions
`drop_perm` $m\ b\ l\ h\ p$	locations $(b, l) \ldots (b, h-1)$ get `Max` and `Cur` permissions p

To strengthen intuitions about permissions, it is useful to consider their meaning in an hypothetical shared-memory concurrent extension of CompCert. In such an extension, different threads have different permissions over a given memory location. Using a concurrent separation logic with fractional permissions [8], and projecting these rich logical permissions on CompCert's simple permissions, we can ensure that the permissions of two threads A and B over a given memory location always fall within one of the following cases:

A's permission	B's permission	What A can do	What B can do
`Freeable`	none	compare, load, store, free	nothing
`Writable`	$\leq$ `Nonempty`	compare, load, store	compare
`Readable`	$\leq$ `Readable`	compare, load	compare, load
`Nonempty`	$\leq$ `Writable`	compare	compare, load, store
none	any	nothing	compare, load, store, free

Combined with the permission checks performed by the various operations, this interpretation of permissions causes programs containing data races to have undefined semantics ("getting stuck"). If two threads are in danger of approaching a data race, at least one of the threads will be "stuck" in its own sequential operational semantics because it will have insufficient permission. For example, if thread A stores at a location while thread B loads from the same location, one or both of the load and store operations will fail by lack of sufficient permissions. A similar failure occurs if A compares a pointer while B is freeing the location. However, concurrent loads are possible if the location has permission `Readable` in both threads.

4.3 In-memory data representations

Memory states include a contents map that associates a value of type `memval` to each byte, that is, each pair (block identifier, byte offset).

```
Inductive memval: Type :=
  | Undef: memval
  | Byte: byte -> memval
  | Pointer: block -> int -> nat -> memval.
```

The contents of a given memory byte can be either

- `Undef`, standing for an unspecified bit pattern such as the contents of an uninitialized memory area;
- `Byte` n where n is a concrete 8-bit integer in the range $0 \ldots 255$;
- `Pointer` b i n, standing for the n-th byte of the abstract pointer (b, i).

The intent of this representation is that integer and float values, when stored in memory, are decomposed into a sequence of bytes $n_1, \ldots, n_k$, taking endianness and IEEE encoding of floats into account, then stored in the contents map as the memvals `Byte` $n_1, \ldots,$ `Byte` n_k. (This is exactly what the hardware processor does.) In contrast, when storing a pointer value `Vptr`(b, i), we hide its hardware representation by storing the memvals `Pointer` b i $0, \ldots,$ `Pointer` b i 3. Loading from memory performs the reverse operation, recovering a value from a sequence of memvals.

These conversions between values and lists of memvals are governed by a memory quantity ("chunk") and encapsulated in the following two functions:

```
encode_val: memory_chunk -> val -> list memval
decode_val: memory_chunk -> list memval -> val
```

Here is the shape of the definition of `encode_val`:

$$
\begin{aligned}
\texttt{encode_val Mint8unsigned (Vint}(n)) &= [\texttt{Byte } x_1] \text{ where } (x_1) = \texttt{encode_int } 1\ n \\
\texttt{encode_val Mint8signed (Vint}(n)) &= [\texttt{Byte } x_1] \text{ where } (x_1) = \texttt{encode_int } 1\ n \\
\texttt{encode_val Mint16unsigned (Vint}(n)) &= [\texttt{Byte } x_1; \texttt{Byte } x_2] \text{ where } (x_1, x_2) = \texttt{encode_int } 2\ n \\
\texttt{encode_val Mint16signed (Vint}(n)) &= [\texttt{Byte } x_1; \texttt{Byte } x_2] \text{ where } (x_1, x_2) = \texttt{encode_int } 2\ n \\
\texttt{encode_val Mint32 (Vint}(n)) &= [\texttt{Byte } x_1; \texttt{Byte } x_2; \texttt{Byte } x_3; \texttt{Byte } x_4] \\
&\quad \text{where } (x_1, \ldots, x_4) = \texttt{encode_int } 4\ n \\
\texttt{encode_val Mint32 (Vptr}(b, i)) &= [\texttt{Pointer } b\ i\ 0; \texttt{Pointer } b\ i\ 1; \texttt{Pointer } b\ i\ 2; \texttt{Pointer } b\ i\ 3] \\
\texttt{encode_val Mfloat32 (Vfloat}(n)) &= [\texttt{Byte } x_1; \texttt{Byte } x_2; \texttt{Byte } x_3; \texttt{Byte } x_4] \\
&\quad \text{where } (x_1, \ldots, x_4) = \texttt{encode_int } 4\ (\texttt{bits_of_single } n) \\
\texttt{encode_val Mfloat64 (Vfloat}(n)) &= [\texttt{Byte } x_1; \ldots; \texttt{Byte } x_8] \\
&\quad \text{where } (x_1, \ldots, x_8) = \texttt{encode_int } 8\ (\texttt{bits_of_double } n) \\
\texttt{encode_val } \tau\ v &= [\texttt{Undef}; \ldots; \texttt{Undef}]\ (|\tau| \text{ times } \texttt{Undef}) \text{ otherwise}
\end{aligned}
$$

Here, `encode_int` l n returns the list of the low l bytes of integer n, either in big-endian or little-endian order depending on the target architecture. `bits_of_single` and `bits_of_double` return as an integer the IEEE 754 representation of a single-precision or double-precision float, respectively.

The `decode_val` function is also defined by case analysis, in a manner symmetrical to that of `encode_val`.

$$
\begin{aligned}
\texttt{decode_val Mint8unsigned}\ [\texttt{Byte}\ x_1] &= \texttt{Vint}(n) \\
&\quad \text{where } n = \texttt{decode_int}\ [x_1] \\
\texttt{decode_val Mint8signed}\ [\texttt{Byte}\ x_1] &= \texttt{Vint}(\texttt{sign_extends}(n)) \\
&\quad \text{where } n = \texttt{decode_int}\ [x_1] \\
\texttt{decode_val Mint16unsigned}\ [\texttt{Byte}\ x_1; \texttt{Byte}\ x_2] &= \texttt{Vint}(n) \\
&\quad \text{where } n = \texttt{decode_int}\ [x_1; x_2] \\
\texttt{decode_val Mint16signed}\ [\texttt{Byte}\ x_1; \texttt{Byte}\ x_2] &= \texttt{Vint}(\texttt{sign_extends}(n)) \\
&\quad \text{where } n = \texttt{decode_int}\ [x_1; x_2] \\
\texttt{decode_val Mint32}\ [\texttt{Byte}\ x_1; \ldots; \texttt{Byte}\ x_4] &= \texttt{Vint}(n) \\
&\quad \text{where } n = \texttt{decode_int}\ [x_1; \ldots; x_4] \\
\texttt{decode_val Mint32}\ [\texttt{Pointer}\ b\ i\ 0; \ldots; \texttt{Pointer}\ b\ i\ 3] &= \texttt{Vptr}(b, i) \\
\texttt{decode_val Mfloat32}\ [\texttt{Byte}\ x_1; \ldots; \texttt{Byte}\ x_4] &= \texttt{Vfloat}(\texttt{single_of_bits}(n)) \\
&\quad \text{where } n = \texttt{decode_int}\ [x_1; \ldots; x_4] \\
\texttt{decode_val Mfloat64}\ [\texttt{Byte}\ x_1; \ldots; \texttt{Byte}\ x_8] &= \texttt{Vfloat}(\texttt{double_of_bits}(n)) \\
&\quad \text{where } n = \texttt{decode_int}\ [x_1; \ldots; x_8] \\
\texttt{decode_val}\ \tau\ L &= \texttt{Vundef} \text{ in all other cases}
\end{aligned}
$$

Here, `decode_int` L combines the given list of bytes L into an integer, using either big-endian or little-endian convention depending on the target architecture.

The `decode_val` function is carefully engineered to be the left inverse of `encode_val`: encoding a value, then decoding the resulting list of bytes, recovers the original value modulo normalization of the value to the memory type used:

$$\texttt{decode_val}\ \tau\ (\texttt{encode_val}\ \tau\ v) = \texttt{convert}\ \tau\ v$$

This property still holds if the bytes are decoded with an integer memory type τ' that differs only in signedness from the memory type τ used for encoding:

$$
\begin{aligned}
\texttt{decode_val}\ \tau'\ (\texttt{encode_val}\ \tau\ v) = &\ \texttt{convert}\ \tau'\ v \\
&\ \text{if } \{\tau, \tau'\} = \{\texttt{Mint8unsigned}, \texttt{Mint8signed}\} \\
&\ \text{or } \{\tau, \tau'\} = \{\texttt{Mint16unsigned}, \texttt{Mint16signed}\}
\end{aligned}
$$

Moreover, encoding as a `Mint32` and decoding as a `Mfloat32`, or conversely, gives access to the IEEE bit-level representation of single-precision floats:

$$
\begin{aligned}
\texttt{decode_val Mfloat32}\ (\texttt{encode_val Mint32}\ (\texttt{Vint}(i))) &= \texttt{single_of_bits}(i) \\
\texttt{decode_val Mint32}\ (\texttt{encode_val Mfloat32}\ (\texttt{Vfloat}(f))) &= \texttt{bits_of_single}(f)
\end{aligned}
$$

4.4 Algebraic laws

Load and loadbytes; store and storebytes A `load` operation is equivalent to a `loadbytes` operation plus a `decode_val` and an alignment check:

$$\texttt{load}\ m\ \tau\ b\ i = \texttt{Some}\ (\texttt{decode_val}\ \tau\ B) \ \Leftrightarrow\ \texttt{loadbytes}\ m\ b\ i\ |\tau| = \texttt{Some}\ B\ \wedge\ \langle\tau\rangle \text{ divides } i$$

Similarly for a `store` operation and the corresponding `storebytes` operation:

$$\texttt{store}\ m\ \tau\ b\ i\ v = \texttt{Some}\ m'\ \Leftrightarrow\ \texttt{storebytes}\ m\ b\ i\ (\texttt{encode_val}\ \tau\ v) = \texttt{Some}\ m'\ \wedge\ \langle\tau\rangle \text{ divides } i$$

Decomposing loadbytes and storebytes operations A `loadbytes` operation on a range of locations is equivalent to two `loadbytes` operations on two adjacent ranges:

$$\texttt{loadbytes}\ m\ b\ i\ (n_1{+}n_2) = \texttt{Some}\ B \ \Leftrightarrow\ \exists B_1, \exists B_2,\ \texttt{loadbytes}\ m\ b\ i\ n_1 = \texttt{Some}\ B_1 \\ \wedge\ \texttt{loadbytes}\ m\ b\ (i+n_1)\ n_2 = \texttt{Some}\ B_2 \\ \wedge\ B = B_1.B_2$$

Likewise, a `storebyte` operation decomposes into two `storebytes`:

$$\texttt{storebytes}\ m\ b\ i\ (B_1.B_2) = \texttt{Some}\ m' \ \Leftrightarrow\ \exists m_1,\ \texttt{storebytes}\ m\ b\ i\ B_1 = \texttt{Some}\ m_1 \\ \wedge\ \texttt{storebytes}\ m_1\ b\ (i+|B_1|)\ B_2 = \texttt{Some}\ m'$$

Load after alloc Same properties as in version 1 of the model: if $\texttt{alloc}\ m\ l\ h = (m', b)$,

- $\texttt{load}\ m'\ \tau'\ b'\ i' = \texttt{load}\ m\ \tau'\ b'\ i'$ if $b' \neq b$
- If $\texttt{load}\ m\ \tau\ b\ i = \texttt{Some}\ v$, then $v = \texttt{Vundef}$

Load after free: if $\texttt{free}\ m\ b\ l\ h = m'$,

- $\texttt{load}\ m'\ \tau'\ b'\ i' = \texttt{load}\ m\ \tau'\ b'\ i'$ if $b' \neq b$ or $i' + |\tau'| \leq l$ or $h \leq i'$
- $\texttt{load}\ m\ \tau\ b\ i = \texttt{None}$ if $l \leq i$ and $i + |\tau| \leq h$.

Loadbytes after storebytes If $\texttt{storebytes}\ m\ b\ i\ B = \texttt{Some}\ m'$,

- Compatible case: $\texttt{loadbytes}\ m'\ b\ i\ |B| = \texttt{Some}\ B$
- Disjoint case: $\texttt{loadbytes}\ m'\ b'\ i'\ n' = \texttt{loadbytes}\ m\ b'\ i'\ n'$ if $b' \neq b$ or $i' + n' \leq i$ or $i + |B| \leq i'$

Cases of partial overlap can be reasoned upon by decomposing the `storebytes` or the `loadbytes` operation into multiple operations.

Load after store if $\texttt{store}\ m\ \tau\ b\ i\ v = \texttt{Some}\ m'$,

- Disjoint case: $\texttt{load}\ m'\ \tau'\ b'\ i' = \texttt{load}\ m\ \tau'\ b'\ i'$ if $b' \neq b$ or $i' + |\tau'| \leq i$ or $i + |\tau| \leq i'$
- Compatible case: $\texttt{load}\ m'\ \tau'\ b\ i = \texttt{decode_val}\ \tau'\ (\texttt{encode_val}\ \tau\ v)$ if $|\tau'| = |\tau|$.
  In particular, if τ' and τ are identical or differ only in signedness, $\texttt{load}\ m'\ \tau'\ b\ i = \texttt{convert}\ \tau'\ v$.

Compared with version 1 of the memory model, it is no longer the case that the `load` must return an `Vundef` value in the "incompatible" and "overlapping" cases. For example, if we write a 64-bit float f at location (b, i), then read a 32-bit integer from this location, the result is not `Vundef` but an integer corresponding to one half of the IEEE bit-level representation of f. This is a strength of the new memory model, as it addresses the limitations described in section 3.2. However, we can still say something in the "incompatible" and "overlapping" cases if the value v just stored is a pointer value.

- Incompatible case for pointer values: if v is a pointer value and $\texttt{load}\ m'\ \tau'\ b\ i = \texttt{Some}\ v'$ and $\tau \neq \texttt{Mint32}\ \vee\ \tau' \neq \texttt{Mint32}$, then $v' = \texttt{Vundef}$.
- Overlapping case for pointer values: if v is a pointer value and $\texttt{load}\ m'\ \tau'\ b\ i' = \texttt{Some}\ v'$ and $i' \neq i$ and $i' + |\tau'| > i$ and $i + |\tau| > i'$, then $v' = \texttt{Vundef}$.

These special cases are related to a more general integrity property for stored pointer values, described next.

Integrity of pointer values As discussed above, it is possible to load integer or float values that were never stored in memory, but arise from combinations of byte-level representations of other integer or float values previously stored in memory. However, the memory model guarantees that this cannot happen for pointer values:

> If $\texttt{store}\ m\ \tau\ b\ i\ v = \texttt{Some}\ m'$ and $\texttt{load}\ m'\ \tau'\ b'\ i' = \texttt{Some}(\texttt{Vptr}(b'', i''))$, then either
>
> - the pointer value that is loaded is the value just stored: $\tau = \tau' = \texttt{Mint32}$ and $b' = b$ and $i' = i$ and $v = \texttt{Vptr}(b'', i'')$;
> - or the load is disjoint from the store: $b' \neq b$ or $i' + |\tau'| \leq i$ or $i + |\tau| \leq i'$; therefore, the pointer value that is loaded was already present in the original memory state: $\texttt{load}\ m\ \tau'\ b'\ i' = \texttt{Some}(\texttt{Vptr}(b'', i''))$

This integrity property is important: if pointer values could arise "out of thin air" by loading byte-level representations of other values, the properties of invariance by memory transformations (section 3.3) would be invalidated, and several passes of the CompCert compiler could no longer be proved semantics-preserving.

4.5 Implementation

The CompCert Coq sources provide an implementation of the memory model in module `Memory`, which is proved to satisfy the algebraic laws listed above and all the other properties specified in module `Memtype`. We briefly outline this implementation, referring the reader to the commented Coq development for more details. Memory states are represented by the following record type:

```
Definition block : Type := Z.

Record mem : Type := mkmem {
  mem_contents: ZMap.t (ZMap.t memval);
  mem_access: ZMap.t (Z -> perm_kind -> option permission);
  nextblock: block;
  nextblock_pos: nextblock > 0;
  access_max:
    forall b ofs, perm_order'' (mem_access#b ofs Max) (mem_access#b ofs Cur);
  nextblock_noaccess:
    forall b ofs k, b <= 0 \/ b >= nextblock -> mem_access#b ofs k = None
```

A memory state is composed of three pieces of data:

- A block identifier `nextblock`, which tracks the first non-allocated block. This is the block identifier that the next `alloc` operation will return as result.
- A map `mem_contents` from (block, offset) pairs to memvals. This map is implemented using the `ZMap` data structure from CompCert's `Maps` library. `ZMap` provides an efficient implementation of `Z`-indexed finite maps with a default value.
- A map `mem_access` from (block, offset, permission-kind) triples to the type `option permission`. This maps records, for every location, the greatest `Cur` permission and the greatest `Max` permission. `None` means no permissions at all.

Additionally, three invariants are packaged with this data:

- The next block is always positive. (Negative block identifiers are reserved by CompCert to represent function pointers.)

- For every location, the `Cur` permission never exceeds the `Max` permission.
- Blocks that have not been allocated yet have empty permissions.

To give a flavor of the implementation, here is the definition of the `store` operation:

```
Definition store (chunk: memory_chunk) (m: mem)
                 (b: block) (ofs: Z) (v: val) : option mem :=
  if valid_access_dec m chunk b ofs Writable then
    Some (mkmem (ZMap.set b
                          (setN (encode_val chunk v) ofs (m.(mem_contents)#b))
                          m.(mem_contents))
                m.(mem_access)
                m.(nextblock)
                m.(nextblock_pos)
                m.(access_max)
                m.(nextblock_noaccess))
  else
    None.
```

`valid_access_dec` decides whether the offset is aligned with respect to the memory chunk and whether all addressed bytes have `Cur`,`Writable` permissions. If not, `store` fails, returning `None`. If so, an updated memory state is returned, where the locations $(\mathtt{b}, \mathtt{ofs}), \ldots, (\mathtt{b}, \mathtt{ofs}+|\mathtt{chunk}|-1)$ are set (by the auxiliary function `setN`) to the list of memvals returned by `encode_val chunk v`, and all other locations are unchanged.

Some clients of the CompCert memory model, such as libraries of lemmas about program transformations and program logics, often find it convenient to reason about the constructive definitions (such as the one shown here for `store`) instead of their axiomatizations. Other clients, such as the correctness proofs for CompCert's compilation passes, only need the axiomatization in terms of algebraic laws outlined in section 4.4.

5 Assessment of the memory model, version 2

Version 2 of the memory model preserves the main features of version 1, discussed in section 3, namely:

- It gives the expected semantics to C programs that conform to ISO C99.
- It gives the expected semantics to those nonconformant C programs that perform wild casts between pointers and make assumptions about the memory layout of composite types (structs, unions and arrays).
- It enjoys useful properties of invariance by memory transformations. The notion of memory injections initially developed for version 1 extends easily to version 2 of the model and was shown to commute with memory operations. The proofs of semantic preservation for CompCert's compiler passes were relatively easy to adapt to version 2 of the memory model.

We now discuss the improvements brought by version 2 on the limitations of version 1.

5.1 Capability: Accounting for low-level programming idioms on integers and floats

Version 2 of the memory model is able to give precise semantics to programs that make assumptions about the memory representations of base types (integers and floats). We now revisit the

examples of section 3.2 to illustrate this new capability.

Endianness change

```
unsigned int bswap(unsigned int x)
{
    union { unsigned int i; char c[4]; } src, dst;
    int n;
    src.i = x;
    /* point 1 */
    dst.c[3] = src.c[0]; dst.c[2] = src.c[1];
    dst.c[1] = src.c[2]; dst.c[0] = src.c[3];
    /* point 2 */
    return dst.i;
}
```

Using the `store-storebytes` equivalence, we see that at program point 1, the memory block associated with `src` contains the encoding [`Byte` x_1; `Byte` x_2; `Byte` x_3; `Byte` x_4] of the integer `x`, where $(x_1, \ldots, x_4) =$ `encode_int` 4 `x`.

Using the `loadbytes-storebytes` laws, we obtain that at program point 2, the memory block associated with `dst` contains the memvals [`Byte` x_4; `Byte` x_3; `Byte` x_2; `Byte` x_1].

The `load-loadbytes` equivalence, then, shows that the integer returned by the function is `decode_int` $[x_4, x_3, x_2, x_1]$, which is indeed the byte-swapping of integer `x`.

Single-precision absolute value

```
float fabs_single(float x)
{
    union { float f; unsigned int i; } u;
    u.f = x;
    /* point 1 */
    u.i = u.i & 0x7FFFFFFF;
    /* point 2 */
    return u.f;
}
```

According to the `load-store`-compatible law, the value of `u.i` at program point 1 is

$$\texttt{decode_val Mint32 (encode_val Mfloat32 x)} = \texttt{bits_of_single(x)}$$

that is, the 32-bit integer corresponding to the IEEE 754 representation of the single-precision float `x`.

Using the same law again, the return value of the function is `single_of_bits`(n), where n is the value of `u.i` at point 2. Since $n =$ `bits_of_single(x) & 0x7FFFFFFF`, it follows that `fabs_single` computes the function

$$x \mapsto \texttt{single_of_bits(bits_of_single}(x)\texttt{ \& 0x7FFFFFFF)}$$

Using a formalization of IEEE 754 such as Flocq [5], it can be proved that this function is floating-point absolute value.

Converting integers to double-precision floats, PowerPC-style

```
double double_of_signed_int(int x)
{
    union { double d; unsigned int i[2]; } a, b;

    a.i[0] = 0x43300000;  a.i[1] = 0x80000000;
    b.i[0] = 0x43300000;  b.i[1] = 0x80000000 + x;
    /* point 1 */
    return b.d - a.d;
}
```

Using many of the memory model laws (`store-storebytes` and `load-loadbytes` equivalence; `loadbytes-storebytes` laws; and `loadbytes` and `storebytes` decomposition properties) and assuming a big-endian architecture, we obtain that at point 1,

$$\mathtt{a.d} = \mathtt{double_of_bits}(\mathtt{0x4330000080000000})$$
$$\mathtt{b.d} = \mathtt{double_of_bits}(\mathtt{0x4330000080000000} + \mathtt{x})$$

The return value of the function is, therefore, the double-precision floating-point difference between these two floats. Using Flocq, it remains to prove that this difference is indeed the float `(double) x`, taking advantage of the fact $-2^{31} \leq \mathtt{x} < 2^{31}$. The point is that the correctness of this function was reduced to a pure floating-point arithmetic problem; the byte-level manipulations over floats are now precisely defined thanks to the new memory model.

5.2 Limitation: No access to bit-level representations of pointers

Consider again the block copy example from section 3.2:

```
void * memcpy(void * dest, const void * src, size_t n)
{
    for (i = 0; i < n; i++)
        ((char *) dest)[i] = ((const char *) src)[i];
    return dest;
}
```

Version 2 of the memory model is able to show that this function executes as expected, but only if the source array `src` contains no pointer values; more precisely, if the memvals at `src`...`src+n−1` are all of the `Byte` or `Undef` kind.

Indeed, when loaded with C type `char`, the memval `Byte` x reads as the integer x or x's sign extension (depending on whether the `char` type is signed). Storing this integer with C type `char` amounts to storing the memval `Byte` x. If the source memval is `Undef`, it reads as the value `Vundef`, and writes back as the memval `Undef`. Finally, if the source memval is a pointer fragment `Pointer` b i n, it reads as the value `Vundef` and writes as `Undef`.

Assuming no overlap between the `src` and `dest` memory areas, the net effect of `memcpy`'s loop, is, therefore, to copy the memvals contained in `src` to the area pointed by `dest`, turning pointer fragments into `Undef` memvals and preserving `Byte` and `Undef` memvals. In other words, the expected behavior of `memcpy`, namely, making an exact copy of `src` into `dest`, is guaranteed by CompCert's semantics only if `src` contains integers and floats, but no pointers.

Similar limitations arise in examples other than `memcpy`. One is the `memcmp` function from the C standard library, which compares the contents of two memory areas as if they were arrays

of characters. (Unlike `memcpy`, the informal semantics of `memcmp` is very unclear to begin with, in particular because it observes the values of padding bytes introduced by compilers in compound data structures.) Another example is the occasional need to reverse the endianness of a pointer, for instance when a big-endian processor exchanges linked data structures with a little-endian USB controller.

Is this a serious limitation? More practical experience with embedded critical codes is needed to answer this question, but here are a few thoughts about this issue.

First, the C standards guarantee the existence of a correct `memcpy` function in the C standard library, but never say that it can be written in conformant C, as the simple byte-per-byte copy loop above or in any other ways. Our memory model version 2 is perfectly able to axiomatize the behavior of such a correct `memcpy` function, as a `loadbytes` operation over the whole range $\texttt{src} \ldots \texttt{src} + \texttt{n} - 1$ followed by a `storebytes` at `dest`.

Second, like many C compilers, CompCert provides a predefined block copy operation, `__builtin_memcpy`, whose semantics is precisely definedd as a `loadbytes` operation followed by a `storebytes` (plus checks for absence of overlap). A current limitation of this built-in operation is that the number `n` of bytes to copy must be a compile-time constant. In exchange, CompCert is able to produce very efficient assembly code for `__builtin_memcpy`, using multi-byte memory accesses and unrolling the copy loop when appropriate. The point is that CompCert-compiled systems code should never define its own `memcpy` function and call it as $\texttt{memcpy}(\texttt{dest}, \texttt{src}, \texttt{sizeof}(\texttt{src}))$: using `__builtin_memcpy` is not only better defined semantically speaking, but also much more efficient.

Third, if the need arises to copy arrays of pointers, we can define a version of `memcpy` specialized for this case:

```
void * memcpy_ptr(void ** dest, const void ** src, size_t n)
{
    for (i = 0; i < n / sizeof(void *); i++)
        dest[i] = src[i];
    return dest;
}
```

Both version 1 and version 2 of the CompCert memory model show that this function correctly copies arrays of pointers. With version 2, it might be possible to show that structs containing a mixture of pointer fields and numerical fields, or arrays of such structs, are copied unchanged as well. This conjecture relies on the fact that pointer fields in structs are always 4-aligned, and force the alignment of the enclosing struct to be at least 4.

5.3 Capability: fine-grained access control

The permission mechanism introduced in version 2 of the model is effective to better control the memory operations that are allowed on global variables, and also to support more aggressive optimizations in the CompCert compiler.

More precise semantics From the standpoint of the CompCert operational semantics, the initial memory state in which a program starts execution is built as follows. For every global variable of the C program, a memory block is allocated, then filled with the initial value provided for this variable, if any, or by a default value of "all zeroes" otherwise; then, permissions over the whole block are dropped to

- `Nonempty` if the type of the global variable carries the `volatile` modifier;

- `Readable` if this type carries the `const` modifier but not the `volatile` modifier;
- `Writable`, otherwise.

(CompCert treats string literals as global, initialized arrays of characters with type `const char []`, hence string literals, too, get `Readable` permissions.)

Dropping permissions over global variables has several benefits. First, attempting to deallocate a global variable or string literal by calling `free` on its address now has undefined semantics, as it should. (This follows from the fact that memory blocks corresponding to global variables always lack the `Freeable` permission.) Second, it is now a semantic error for a program to try to assign into a `const` global variable, or to access a `volatile` global variable through normal `load` and `store` operations.

The latter point deserves more explanations on how CompCert handles the `volatile` modifier. Accesses to l-values having `volatile` static type are compiled and given semantics not via normal `load` and `store` operations, but via special built-in functions, `__builtin_volatile_read` and `__builtin_volatile_write` that check whether the location actually accessed is an object declared `volatile` or not. In the former case, the `volatile` access is treated as an input/output operation, communicating with the outside world through an event in the trace of observables for the program, and bypassing the memory model entirely. In the latter case, the `volatile` access is treated as a regular `load`/`store` operation. To summarize:

Static type	Semantics & compilation	Actual location accessed	
		not `volatile`	`volatile`
not `volatile`	regular `load`/`store` operation	`load`/`store`	error
`volatile`	`__builtin_volatile` operation	`load`/`store`	I/O event

The semantic error in the top right case is a consequence of the accessed location lacking `Readable` and `Writable` permissions. It agrees with the prescriptions of the C standards, which state that undefined behavior arises if a `volatile` object is accessed through a non-`volatile` l-value, as can arise if a pointer cast is used to remove the `volatile` modifier from the type of the pointed object.

The discussion above is framed in terms of global variables. For function-local variables, CompCert essentially ignores the `const` and `volatile` modifiers: `volatile` local variables make little sense, as they cannot correspond to a hardware memory device; `const` local variables cannot have their permissions dropped, because there would be no way to raise these permissions back to `Freeable` before deallocating them at function return time.

More aggressive optimizations We improved the constant propagation pass of CompCert 1.11 to take advantage of the "`const`-ness" of global variables. Consider:

```
const int n = 1;
const double tbl[3] = { 1.11, 2.22, 3.33 };
double f(void) { return tbl[n]; }
```

Owing to the `const` modifiers, the value of `n` is always 1 throughout execution, and the value of `tbl[1]` is always `2.22`. It is therefore legitimate to optimize function `f` into

```
double f(void) { return 2.22; }
```

This is what the improved constant propagation pass now does. Its correctness proof (semantic preservation) nicely exploits the new features of the CompCert v2 memory model. Namely, the proof shows that the contents of `const` global variables are identical in the initial memory state and in the memory state at any point of the program execution. Indeed, a successful `store`

operation performed by the program cannot change the contents of a memory block attached to a `const` global variables, because 1- such a memory block has maximal permission `Readable` (at most) in the initial memory state; 2- maximal permissions over already-allocated blocks can only decrease during execution; and 3- a successful `store` requires `Writable` current permissions over the locations it modifies.

Besides sporting a nice proof of semantic preservation, constant propagation of `const` global variables noticeably improves the quality of the assembly code produced by CompCert in some cases. One example that we observed is C code automatically generated from Scade, where the C code generator puts many numerical constants in `const` global variables rather than naming them with macros.

6 Conclusions and perspectives

Version 2 of the CompCert memory model enables the semantics of the source and intermediate languages of CompCert to describe the memory behavior of programs with increased precision, while preserving all the properties of memory operations that CompCert's correctness proof relies on. This increase in precision translates into two improvements:

- More of the popular, low-level, non-standard-conformant C programming idioms, such as bit-level manipulations of in-memory representations of integers and floats, can be given well-defined semantics and, therefore, be proved correct with respect to their high-level specifications, but also guaranteed to be compiled by CompCert in a semantics-preserving manner.
- More of the serious C undefined behaviors, such as modifying a string literal, can be captured as errors by the CompCert formal semantics, enabling the CompCert compiler to perform more aggressive optimizations and an hypothetical verification tool based on CompCert's semantics to guarantee the absence of such errors.

The main limitation that remains CompCert's memory model is its inability to model byte-level access to pointer values, as can happen in block copy operations, for instance. We argued that this limitation seems to be of low practical importance. Nonetheless, we see two ways to lift this limitation:

- The first approach is fairly *ad-hoc* and consists in extending CompCert's type of values with a fifth case, `Vptr_fragment`(b, i, n), abstractly denoting the n-th byte of the in-memory representation of pointer `Vptr`(b, i). Byte-sized `load` and `store` operations would translate between the `Vptr_fragment`(b, i, n) value and the `Pointer`(b, i, n) memval without loss of information. Most if not all arithmetic operations would be undefined over values of the `Vptr_fragment` kind. This is the minimal extension that would give semantics to the `memcpy` example.
- The second approach is much more radical: replace CompCert's current "value" type (a discriminated union of integer, float, pointer and undefined values) by the type `list memval` of lists of byte-level, in-memory representations of values. This is the approach followed by Norrish in his Cholera formal semantics for the C language [13]: in Cholera, r-value expressions evaluate (conceptually) to their byte-level, in-memory representations. In this approach, encoding and decoding integer/float/pointer values to and from lists of bytes is no longer performed at load/store time, but rather when an arithmetic or logical operation is performed. This is a major departure from the approach followed in CompCert so far.

Finally, the introduction of fine-grained permissions in the memory model is a first step towards extending CompCert to shared-memory, data-race-free concurrency. Further steps in this di-

rection include re-engineering the operational semantics of CompCert's languages around the "oracle" model of Dockins and Appel [2].

Acknowledgments

We thank Pascal Cuoq, Rob Dockins and Alexandre Pilkiewicz for their suggestions and feedback on the design of the v2 memory model and its uses in the CompCert semantics and correctness proofs.

References

[1] Sarita V. Adve and Hans-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2010.

[2] Andrew W. Appel. Verified software toolchain - (invited talk). In *Programming Languages and Systems – 20th European Symposium on Programming, ESOP 2011*, volume 6602 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2011.

[3] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 5–21. Springer, 2007.

[4] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[5] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pages 243–252. IEEE Computer Society Press, 2011.

[6] Richard Bornat. Proving pointer programs in Hoare logic. In *MPC '00: Proc. Int. Conf. on Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 102–126. Springer, 2000.

[7] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[8] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *Programming Languages and Systems, 7th Asian Symposium*, volume 5904 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2009.

[9] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2008.

[10] ISO. International standard ISO/IEC 9899:1999, Programming languages – C, 1999.

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

[12] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[13] Michael Norrish. *C formalized in HOL*. PhD thesis, University of Cambridge, 1998. Technical report UCAM-CL-TR-453.

[14] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[15] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In *34th symposium Principles of Programming Languages*, pages 97–108. ACM Press, 2007.

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399