Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Formal verification of a C-like memory model and its uses for verifying program transformations

Xavier Leroy · Sandrine Blazy

the date of receipt and acceptance should be inserted later

Abstract This article presents the formal verification, using the Coq proof assistant, of a memory model for low-level imperative languages such as C and compiler intermediate languages. Beyond giving semantics to pointer-based programs, this model supports reasoning over transformations of such programs. We show how the properties of the memory model are used to prove semantic preservation for three passes of the Compcert verified compiler.

1 Introduction

A prerequisite to the formal verification of computer programs—by model checking, program proof, static analysis, or any other means—is to formalize the semantics of the programming language in which the program is written, in a way that is exploitable by the verification tools used. In the case of program proofs, these formal semantics are often presented in operational or axiomatic styles, e.g. 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. However, many programs that require formal verification are written in imperative languages featuring pointers (or references) 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 store

X. Leroy
INRIA Paris-Rocquencourt, B.P. 105, 78153 Le Chesnay, France
E-mail: Xavier.Leroy@inria.fr

S. Blazy
ENSIIE, 1 square de la Résistance, 91025 Evry cedex, France
E-mail: Sandrine.Blazy@ensiie.fr

and operations over it. The memory model is often a delicate part of a formal semantics for an imperative programming language. A very 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, making 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 article reports on the formalization and verification, using the Coq proof assistant, of a memory model for C-like imperative languages. 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 areas, 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 the Compcert compiler [15, 3], a moderately-optimizing compiler that translates the Clight subset of the C programming language down to PowerPC assembly code. The memory model is used by the formal semantics of all languages manipulated by the compiler: the source language, the target language, and 7 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, local variables of a Clight 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 requires extensive reasoning over memory states, using the properties of the memory model given further in the paper.

The remainder of this article is organized as follows. Section 2 axiomatizes the values that are stored in memory states and the associated memory data types. Section 3 specifies an abstract memory model and illustrates its use for reasoning over programs. Section 4 defines the concrete implementation of the memory model used in Compcert and shows that it satisfies both the abstract specification and additional useful properties. Section 5 describes the transformations over memory states performed by three passes of the Compcert compiler. It then defines the memory invariants and proves the simulation results between memory operations that play a crucial role in proving semantics preservation for these three passes. Section 6 briefly comments on the Coq mechanization of these results. Related work is discussed in section 7, followed by conclusions and perspectives in section 8.

All results presented in this article have been mechanically verified using the Coq proof assistant [8, 2]. The complete Coq development is available online at `http://gallium.inria.fr/~xleroy/memory-model/`. Consequently, the paper only sketches the proofs of some of its results; the reader is referred to the Coq development for the full proofs.

2 Values and data types

We assume given a set `val` of values, ranged over by v, used in the dynamic semantics of the languages to represent the results of calculations. In the Compcert development, `val` is defined as the discriminated union of 32-bit integers $\texttt{int}(n)$, 64-bit double-precision floating-point numbers $\texttt{float}(f)$, memory locations $\texttt{ptr}(b, i)$ where b is a memory block reference b and i a byte offset within this block, and the constant `undef` representing an undefined value such as the value of an uninitialized variable.

We also assume given a set `memtype` of memory data types, ranged over by τ. Every memory access (`load` or `store` operation) takes as argument a memory data type, serving two purposes: (1) to indicate the size and natural alignment of the data being accessed, and (2) to enforce compatibility guarantees between the type with which a data was stored and the type with which it is read back. For a semantics for C, we can use C type expressions from the source language as memory data types. For the Compcert intermediate languages, we use the following set of memory data types, corresponding to the data that the target processor can access in one `load` or `store` instruction:

$\tau ::=$ `int8signed` | `int8unsigned`	8-bit integers
| `int16signed` | `int16unsigned`	16-bit integers
| `int32`	32-bit integers or pointers
| `float32`	32-bit, single-precision floats
| `float64`	64-bit, double-precision floats

The first role of a memory data type τ is to determine the size $|\tau|$ in bytes that a data of type τ occupies in memory, as well as the natural alignment $\langle\tau\rangle$ for data of this type. The alignment $\langle\tau\rangle$ models the address alignment constraints that many processors impose, *e.g.*, the address of a 32-bit integer must be a multiple of 4. Both size and alignment are positive integers.[1] [2]

(A1) $|\tau| > 0$ and $\langle\tau\rangle > 0$

To reason about some memory transformations, it is useful to assume that there exists a maximal alignment `max_alignment` that is a multiple of all possible alignment values:

(A2) $\langle\tau\rangle$ divides `max_alignment`

For the semantics of C, $|\tau|$ is the size of the type τ as returned by the `sizeof` operator of C. A possible choice for $\langle\tau\rangle$ is the size of the largest scalar type occurring in τ. For the Compcert intermediate languages, we take:

$$|\texttt{int8signed}| = |\texttt{int8unsigned}| = 1$$
$$|\texttt{int16signed}| = |\texttt{int16unsigned}| = 2$$
$$|\texttt{int32}| = |\texttt{float32}| = 4$$
$$|\texttt{float64}| = 8$$

[1] In this article, we write A for axioms, that is, assertions that we will not prove; S for specifications, that is, expected properties of the abstract memory model which the concrete model, as well as any other implementation, satisfies; D for derived properties, provable from the specifications; and P for properties of the concrete memory model.

[2] Throughout this article, variables occurring free in mathematical statements are implicitly universally quantified at the beginning of the statement.

Concerning alignments, Compcert takes $\langle \tau \rangle = 1$ and `max_alignment` $= 1$, since the target architecture (PowerPC) has no alignment constraints. To model a target architecture with alignment constraints such as the Sparc, we would take $\langle \tau \rangle = |\tau|$ and `max_alignment` $= 8$.

We now turn to the second role of memory data types, namely a form of dynamic type-checking. For a strongly-typed language, a memory state is simply a partial mapping from memory locations to values: either the language is statically typed, guaranteeing at compile-time that a value written with type τ is always read back with type τ; or the language is dynamically typed, in which case the generated machine code contains enough run-time type tests to enforce this property. However, the C language and most compiler intermediate languages are weakly typed. Consider a C "union" variable:

```
union { int i; float f; } u;
```

It is possible to assign an integer to `u.i`, then read it back as a float via `u.f`. This will not be detected at compile-time, and the C compiler will not generate code to prevent this. Yet, the C standard [13] specifies that this code has undefined behavior. More generally, after writing a data of type τ to a memory location, this location can only be read back with the same type τ or a compatible type; the behavior is undefined otherwise [13, section 6.5, items 6 and 7]. To capture this behavior in a formal semantics for C, the memory state associates type-value pairs (τ, v), and not just values, to locations. Every `load` with type τ' at this location will check compatibility between the actual type τ of the location and the expected type τ', and fail if they are not compatible.

We abstract this notion of compatibility as a relation $\tau \sim \tau'$ between types. We assume that a type is always compatible with itself, and that compatible types have the same size and the same alignment:

(A3) $\tau \sim \tau$
(A4) If $\tau_1 \sim \tau_2$, then $|\tau_1| = |\tau_2|$ and $\langle \tau_1 \rangle = \langle \tau_2 \rangle$

Several definitions of the $\sim$ relation are possible, leading to different instantiations of our memory model. In the strictest instantiation,, $\tau \sim \tau'$ holds only if $\tau = \tau'$; that is, no implicit casts are allowed during a `store-load` sequence. The C standard actually permits some such casts [13, section 6.5, item 7]. For example, an integer n can be stored as an `unsigned char`, then reliably read back as a `signed char`, with result `(signed char)` n. This can be captured in our framework by stating that `unsigned char` $\sim$ `signed char`. For the Compcert intermediate languages, we go one step further and define $\tau_1 \sim \tau_2$ as $|\tau_1| = |\tau_2|$.

To interpret implicit casts in a `store-load` sequence, we need a function `convert` : `val` $\times$ `memtype` $\to$ `val` that performs these casts. More precisely, writing a value v with type τ, then reading it back with a compatible type τ' results in value `convert`(v, τ'). For the strict instantiation of the model, we take `convert`$(v, \tau') = v$. For the interpretation closest to the C standard, we need `convert`$(v, \tau') = (\tau')\ v$, where the right-hand side denotes a C type cast. Finally, for the Compcert intermediate languages, `convert` is defined as:

$$\begin{aligned}
\texttt{convert}(\texttt{int}(n), \texttt{int8unsigned}) &= \texttt{int}(\text{8-bit zero extension of } n) \\
\texttt{convert}(\texttt{int}(n), \texttt{int8signed}) &= \texttt{int}(\text{8-bit sign extension of } n) \\
\texttt{convert}(\texttt{int}(n), \texttt{int16unsigned}) &= \texttt{int}(\text{16-bit zero extension of } n) \\
\texttt{convert}(\texttt{int}(n), \texttt{int16signed}) &= \texttt{int}(\text{16-bit sign extension of } n)
\end{aligned}$$

$$\begin{aligned}
\texttt{convert}(\texttt{int}(n), \texttt{int32}) &= \texttt{int}(n) \\
\texttt{convert}(\texttt{ptr}(b, i), \texttt{int32}) &= \texttt{ptr}(b, i) \\
\texttt{convert}(\texttt{float}(f), \texttt{float32}) &= \texttt{float}(f \text{ normalized to single precision}) \\
\texttt{convert}(\texttt{float}(f), \texttt{float64}) &= \texttt{float}(f) \\
\texttt{convert}(v, \tau) &= \texttt{undef} \quad \text{in all other cases}
\end{aligned}$$

Note that this definition of `convert`, along with the fact that $\tau_1 \not\sim \tau_2$ if $|\tau_1| \neq |\tau_2|$, ensures that low-level implementation details such as the memory endianness or the bit-level representation of floats cannot be observed by Compcert intermediate programs. For instance, writing a float f with type `float32` and reading it back with compatible type `int32` results in the undefined value `undef` and not in the integer corresponding to the bit-pattern for f.

3 Abstract memory model

We now give an abstract, incomplete specification of a memory model that attempts to formalize the memory-related aspects of C and related languages. We have an abstract type `block` of references to memory blocks, and an abstract type `mem` of memory states. Intuitively, we view the memory state as a collection of separated blocks, identified by block references b. Each block behaves like an array of bytes, and is addressed using byte offsets $i \in \mathbb{Z}$. A memory location is therefore a pair (b, i) of a block reference b and an offset i within this block. The constant $\texttt{empty} : \texttt{mem}$ represents the initial memory state. Four operations over memory states are provided as total functions:

$$\begin{aligned}
\texttt{alloc} &: \texttt{mem} \times \mathbb{Z} \times \mathbb{Z} \to \texttt{option}(\texttt{block} \times \texttt{mem}) \\
\texttt{free} &: \texttt{mem} \times \texttt{block} \to \texttt{option mem} \\
\texttt{load} &: \texttt{memtype} \times \texttt{mem} \times \texttt{block} \times \mathbb{Z} \to \texttt{option val} \\
\texttt{store} &: \texttt{memtype} \times \texttt{mem} \times \texttt{block} \times \mathbb{Z} \times \texttt{val} \to \texttt{option mem}
\end{aligned}$$

Option types are used to represent potential failures. A value of type $\texttt{option}\ t$ is either ε (pronounced "none"), denoting failure, or $\lfloor x \rfloor$ (pronounced "some x"), denoting success with result $x : t$.

Allocation of a fresh memory block is written $\texttt{alloc}(m, l, h)$, where m is the initial memory state, and $l \in \mathbb{Z}$ and $h \in \mathbb{Z}$ are the low and high bounds for the fresh block. The allocated block has size $h - l$ bytes and can be accessed at byte offsets $l, l+1, \ldots, h-2, h-1$. In other terms, the low bound l is inclusive but the high bound h is exclusive. Allocation can fail and return ε if not enough memory is available. Otherwise, $\lfloor b, m' \rfloor$ is returned, where b is the reference to the new block and m' the updated memory state.

Conversely, $\texttt{free}(m, b)$ deallocates block b in memory m. It can fail if *e.g.*, b was already deallocated. In case of success, an updated memory state is returned.

Reading from memory is written $\texttt{load}(\tau, m, b, i)$. A data of type τ is read from block b of memory state m at byte offset i. If successful, the value thus read is returned. The memory state is unchanged.

Symmetrically, $\texttt{store}(\tau, m, b, i, v)$ writes value v at offset i in block b of m. If successful, the updated memory state is returned.

We now axiomatize the expected properties of these operations. The properties are labeled S to emphasize that they are specifications that any implementation of the model must satisfy. The first hypotheses are "good variable" properties defining the behavior of a `load` following an `alloc`, `free` or `store` operation.

(S5) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $b' \neq b$, then $\texttt{load}(\tau, m', b', i) = \texttt{load}(\tau, m, b', i)$
(S6) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $\texttt{load}(\tau, m', b', i) = \texttt{load}(\tau, m, b', i)$
(S7) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $\tau \sim \tau'$, then $\texttt{load}(\tau', m', b, i) = \texttt{convert}(v, \tau')$
(S8) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $b' \neq b \ \vee \ i' + |\tau'| \leq i \ \vee \ i + |\tau| \leq i'$, then $\texttt{load}(\tau', m', b', i') = \texttt{load}(\tau', m, b', i')$

Hypotheses S5 and S6 state that allocating a block b or freeing a block b preserves loads performed in any other block $b' \neq b$. Hypothesis S7 states that after writing value v with type τ at offset i in block b, reading from the same location with a compatible type τ' succeeds and returns the value $\texttt{convert}(v, \tau')$. Hypothesis S8 states that storing a value of type τ in block b at offset i commutes with loading a value of type τ' in block b' at offset i', provided the memory areas corresponding to the `store` and the `load` are separate: either $b' \neq b$, or the range $[i, i + |\tau|)$ of byte offsets modified by the `store` is disjoint from the range $[i', i' + |\tau'|)$ read by the `load`.

Note that the properties above do not fully specify the `load` operation: nothing can be proved about the result of loading from a freshly allocated block, or freshly deallocated block, or just after a store with a type and location that do not fall in the S7 and S8 case. This under-specification is intentional and follows the C standard. The concrete memory model of section 4 will fully specify these behaviors.

The "good variable" properties use hypotheses $b' \neq b$, that is, separation properties between blocks. To establish such properties, we axiomatize the relation $m \models b$, meaning that the block reference b is valid in memory m. Intuitively, a block reference is valid if it was previously allocated but not yet deallocated; this is how the $m \models b$ relation will be defined in section 4.

(S9) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $\neg(m \models b)$.
(S10) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $m' \models b' \ \Leftrightarrow \ b' = b \ \vee \ m \models b'$
(S11) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $m' \models b' \ \Leftrightarrow \ m \models b'$
(S12) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $m' \models b' \ \Leftrightarrow \ m \models b'$
(S13) If $m \models b$, then there exists m' such that $\texttt{free}(m, b) = \lfloor m' \rfloor$.

Hypothesis S9 says that the block returned by `alloc` is fresh, *i.e.*, distinct from any other block that was valid in the initial memory state. Hypothesis S10 says that the newly allocated block is valid in the final memory state, as well as all blocks that were valid in the initial state. Block validity is preserved by `store` operations (S11). After a $\texttt{free}(m, b)$ operation, all initially valid blocks other than b remain valid, but it is unspecified whether the deallocated block b is valid or not (S12). Finally, the `free` operation is guaranteed to succeed when applied to a valid block (S13).

The next group of hypotheses axiomatizes the function $\mathcal{B}(m, b)$ that associates low and high bounds l, h to a block b in memory state m. We write $\mathcal{B}(m, b) = [l, h)$ to emphasize the meaning of bounds as semi-open intervals of allowed byte offsets within block b.

(S14) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $\mathcal{B}(m', b) = [l, h)$.
(S15) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $b' \neq b$, then $\mathcal{B}(m', b') = \mathcal{B}(m, b')$.
(S16) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $\mathcal{B}(m', b') = \mathcal{B}(m, b')$.

(S17) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $\mathcal{B}(m', b') = \mathcal{B}(m, b')$.

A freshly allocated block has the bounds that were given as argument to the `alloc` function (S14). The bounds of a block b' are preserved by an `alloc`, `store` or `free` operation, provided b' is not the block being allocated or deallocated.

For convenience, we write $\mathcal{L}(m, b)$ and $\mathcal{H}(m, b)$ for the low and high bounds attached to b, respectively, so that $\mathcal{B}(m, b) = [\mathcal{L}(m, b), \mathcal{H}(m, b))$.

Combining block validity with bound information, we define the "valid access" relation $m \models \tau \mathbin{@} b, i$, meaning that in state m, it is valid to write with type τ in block b at offset i.

$$m \models \tau \mathbin{@} b, i \quad \stackrel{\text{def}}{=} \quad m \models b \ \wedge\ \langle \tau \rangle \text{ divides } i \ \wedge\ \mathcal{L}(m, b) \leq i \ \wedge\ i + |\tau| \leq \mathcal{H}(m, b)$$

In other words, b is a valid block, the range $[i, i + |\tau|)$ of byte offsets being accessed is included in the bounds of b, and the offset i is an integer multiple of the alignment $\langle \tau \rangle$. If these conditions hold, we impose that the corresponding `store` operation succeeds.

(S18) If $m \models \tau \mathbin{@} b, i$ then there exists m' such that $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$.

Here are some derived properties of the valid access relation, easily provable from the hypotheses above.

(D19) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $\langle \tau \rangle$ divides i and $l \leq i$ and $i + |\tau| \leq h$, then $m' \models \tau \mathbin{@} b, i$.
(D20) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $m \models \tau \mathbin{@} b', i$, then $m' \models \tau \mathbin{@} b', i$.
(D21) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $m' \models \tau \mathbin{@} b', i \ \Leftrightarrow\ m \models \tau \mathbin{@} b', i$.
(D22) If $\texttt{free}(m, b) = \lfloor m' \rfloor$ and $b' \neq b$, then $m' \models \tau \mathbin{@} b', i \ \Leftrightarrow\ m \models \tau \mathbin{@} b', i$.

Proof D19 follows from S10 and S14. D20 follows from S10 and S15, noticing that $b' \neq b$ by S9. D21 follows from S11 and S16, and D22 from S12 and S17.

To finish this section, we show by way of an example that the properties axiomatized above are sufficient to reason over the behavior of a C pointer program using axiomatic semantics. Consider the following C code fragment:

```
int * x = malloc(2 * sizeof(int));
int * y = malloc(sizeof(int));
x[0] = 0;
x[1] = 1;
*y = x[0];
x[0] = x[1];
x[1] = *y;
```

We would like to show that in the final state, `x[0]` is 1 and `x[1]` is 0. Assuming that errors are automatically propagated using a monadic interpretation, we can represent the code fragment above as follows, using the operations of the memory model to make explicit memory operations. The variable `m` holds the current memory state. We also annotate the code with logical assertions expressed in terms of the memory model. The notation Γ stands for the three conditions $\texttt{x} \neq \texttt{y}$, $\texttt{m} \models \texttt{x}$, $\texttt{m} \models \texttt{y}$.

```
(x, m) = alloc(m, 0, 8);
         /* m ⊨ x */
(y, m) = alloc(m, 0, 4);
         /* Γ */
m = store(int, x, 0, 0);
         /* Γ, load(m,x,0) = ⌊0⌋ */
```

```
m = store(int, x, 4, 1);
        /* Γ, load(m,x,0) = ⌊0⌋, load(m,x,4) = ⌊1⌋ */
t = load(int, x, 0);
        /* Γ, load(m,x,0) = ⌊0⌋, load(m,x,4) = ⌊1⌋, t = 0 */
m = store(int, y, 0, t);
        /* Γ, load(m,x,0) = ⌊0⌋, load(m,x,4) = ⌊1⌋, load(m,y,0) = ⌊0⌋ */
t = load(int, x, 4);
        /* Γ, load(m,x,0) = ⌊0⌋, load(m,x,4) = ⌊1⌋, load(m,y,0) = ⌊0⌋, t = 1 */
m = store(int, x, 0, t);
        /* Γ, load(m,x,0) = ⌊1⌋, load(m,x,4) = ⌊1⌋, load(m,y,0) = ⌊0⌋ */
t = load(int, y, 0);
        /* Γ, load(m,x,0) = ⌊1⌋, load(m,x,4) = ⌊1⌋, load(m,y,0) = ⌊0⌋, t = 0 */
m = store(int, x, 4, t);
        /* Γ, load(m,x,0) = ⌊1⌋, load(m,x,4) = ⌊0⌋, load(m,y,0) = ⌊0⌋ */
```

Every postcondition can be proved from its precondition using the hypotheses listed in this section. The validity of blocks `x` and `y`, as well as the inequality $\texttt{x} \neq \texttt{y}$, follow from S9, S10 and S11. The assertions over the results of `load` operations and over the value of the temporary `t` follow from the good variable properties S7 and S8. Additionally, we can show that the `store` operations do not fail using S18 and the additional invariants $\texttt{m} \models \texttt{int} \,@\, \texttt{x}, 0$ and $\texttt{m} \models \texttt{int} \,@\, \texttt{x}, 4$ and $\texttt{m} \models \texttt{int} \,@\, \texttt{y}, 0$, which follow from D19, D20 and D21.

4 Concrete memory model

We now develop a concrete implementation of a memory model that satisfies the axiomatization in section 3. The type `block` of memory block references is implemented by the type $\mathbb{N}$ of nonnegative integers. Memory states (type `mem`) are quadruples (N, B, F, C), where

- N : `block` is the first block not yet allocated;
- B : `block` $\to \mathbb{Z} \times \mathbb{Z}$ associates bounds to each block reference;
- F : `block` $\to$ `boolean` says, for each block, whether it has been deallocated (`true`) or not (`false`);
- C : `block` $\to \mathbb{Z} \to$ `option` (`memtype` $\times$ `val`) associates a content to each block b and each byte offset i. A content is either ε, meaning "invalid", or $\lfloor \tau, v \rfloor$, meaning that a value v was stored at this location with type τ.

We define block validity $m \models b$, where $m = (N, B, F, C)$, as $b < N \ \wedge \ F(b) =$ `false`, that is, b was previously allocated ($b < N$) but not previously deallocated ($F(b) =$ `false`). Similarly, the bounds $\mathcal{B}(m, b)$ are defined as $B(b)$.

The definitions of the constant `empty` and the operations `alloc`, `free`, `load` and `store` follow. We write $m = (N, B, F, C)$ for the initial memory state.

`empty` =
  $(0,\ \lambda b.\ [0, 0),\ \lambda b.\ \texttt{false},\ \lambda b.\lambda i.\ \varepsilon)$
`alloc`(m, l, h) =
  if `can_allocate`$(m, h - l)$ then $\lfloor b, m' \rfloor$ else ε
  where $b = N$
  and $m' = (N + 1, B\{b \leftarrow [l, h)\}, F\{b \leftarrow \texttt{false}\}, C\{b \leftarrow \lambda i.\ \varepsilon\})$
`free`(m, b) =
  if not $m \models b$ then ε

else $\lfloor N, B\{b \leftarrow [0,0)\}, F\{b \leftarrow \texttt{true}\}, C\rfloor$
$\texttt{store}(\tau, m, b, i, v) =$
if not $m \models \tau \mathbin{@} b, i$ then ε
else $\lfloor N, B, F, C\{b \leftarrow c'\}\rfloor$
where $c' = C(b)\{i \leftarrow \lfloor \tau, v\rfloor, i+1 \leftarrow \varepsilon, \ldots, i+|\tau|-1 \leftarrow \varepsilon\}$
$\texttt{load}(\tau, m, b, i) =$
if not $m \models \tau \mathbin{@} b, i$ then ε
else if $C(b)(i) = \lfloor \tau', v\rfloor$ and $\tau' \sim \tau$
and $C(b)(i+j) = \varepsilon$ for $j = 1, \ldots, |\tau| - 1$
then $\lfloor \texttt{convert}(v, \tau)\rfloor$
else $\lfloor \texttt{undef}\rfloor$

Allocation is performed by incrementing linearly the N component of the memory state. Block identifiers are never reused, which greatly facilitates reasoning over "dangling pointers" (references to blocks previously deallocated). The new block is given bounds $[l, h)$, deallocated status `false`, and invalid contents $\lambda i.\ \varepsilon$.[3] An unspecified, boolean-valued `can_allocate` function is used to model the possibility of failure if the request ($h - l$ bytes) exceeds the available memory. In the Compcert development, `can_allocate` always returns `true`, therefore modeling an infinite memory.

Freeing a block simply sets its deallocated status to `true`, rendering this block invalid, and its bounds to $[0, 0)$, reflecting the fact that this block no longer occupies any memory space.

A memory store first checks block and bounds validity using the $m \models \tau \mathbin{@} b, i$ predicate, which is decidable. The contents $C(b)$ of block b are set to $\lfloor \tau, v\rfloor$ at offset i, recording the store done at this offset, and to ε at offsets $i+1, \ldots, i+|\tau|-1$, invalidating whatever data was previously stored at these addresses.

A memory load checks several conditions: first, that the block and offset being addressed are valid and within bounds; second, that block b at offset i contains a valid data $\lfloor v, \tau'\rfloor$; third, that the type τ' of this data is compatible with the requested type τ; fourth, that the contents of offsets $i+1$ to $i+|\tau|-1$ in block b are invalid, ensuring that the data previously stored at i in b was not partially overwritten by a store at an overlapping offset.

It is easy to show that this implementation satisfies the specifications given in section 3.

Lemma 23 *Properties S5 to S18 are satisfied.*

Proof Most properties follow immediately from the definitions of `alloc`, `free`, `load` and `store` given above. For the "good variable" property S7, the `store` assigned contents $\lfloor \tau, v\rfloor, \varepsilon, \ldots, \varepsilon$ to offsets $i, \ldots, i + |\tau| - 1$, respectively. Since $|\tau'| = |\tau|$ by A1, the checks performed by `load` succeed. For the other "good variable" property, S8, the assignments performed by the `store` over $C(b)$ at offsets $i, \ldots, i+|\tau|-1$ have no effect over the values of offsets $i', \ldots, i' + |\tau'| - 1$ in $C(b')$, given the separation hypothesis $(b' \neq b \ \vee \ i' + |\tau'| \leq i \ \vee \ i + |\tau| \leq i')$.

[3] Since blocks are never reused, the freshly-allocated block b already has deallocated status `false` and contents $\lambda i.\ \varepsilon$ in the initial memory state (N, B, F, C). Therefore, in the definition of `alloc`, the updates $F\{b \leftarrow \texttt{false}\}$ and $C\{b \leftarrow \lambda i.\ \varepsilon\}$ are not strictly necessary. However, they allow for simpler reasoning over the `alloc` function, making it unnecessary to prove the invariants $F(b) = \texttt{false}$ and $C(b) = \lambda i.\ \varepsilon$ for all $b \geq N$.

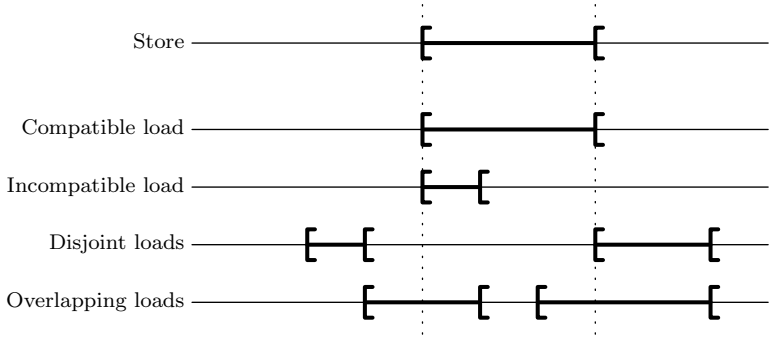


Fig. 1 A `store` followed by a `load` in the same block: the four cases of property D29.

Moreover, the implementation also enjoys a number of properties that we now state. In the following sections, we will only use these properties along with those of section 3, but not the precise definitions of the memory operations. The first two properties state that a `store` or a `load` succeeds if and only if the corresponding memory reference is valid.

(P24) $m \models \tau \mathbin{@} b, i \;\Leftrightarrow\; \exists m', \texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$
(P25) $m \models \tau \mathbin{@} b, i \;\Leftrightarrow\; \exists v, \texttt{load}(\tau, m, b, i) = \lfloor v \rfloor$

Then come additional properties capturing the behavior of a `load` following an `alloc` or a `store`. In circumstances where the "good variable" properties of the abstract memory model leave unspecified the result of the `load`, these "not-so-good variable" properties guarantee that the `load` predictably returns $\lfloor \texttt{undef} \rfloor$.

(P26) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$ and $\texttt{load}(\tau, m', b, i) = \lfloor v \rfloor$, then $v = \texttt{undef}$.
(P27) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $\tau \not\sim \tau'$ and $\texttt{load}(\tau', m', b, i) = \lfloor v' \rfloor$, then $v' = \texttt{undef}$.
(P28) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $i' \neq i$ and $i' + |\tau'| > i$ and $i + |\tau| > i'$ and $\texttt{load}(\tau', m', b, i') = \lfloor v' \rfloor$, then $v' = \texttt{undef}$.

Proof For P26, the contents of m' at b, i are ε and therefore not of the form $\lfloor \tau, v \rfloor$. For P27, the test $\tau \sim \tau'$ in the definition of `load` fails. For P28, consider the contents c of block b in m'. If $i < i'$, the `store` set $c(i') = \varepsilon$. If $i > i'$, the `store` set $c(i'+j) = \lfloor \tau, v \rfloor$ for some $j \in [1, |\tau'|)$. In both cases, one of the checks in the definition of `load` fails.

Combining properties S7, S8, P25, P27 and P28, we obtain a complete characterization of the behavior of a `load` that follows a `store`. (See figure 1 for a graphical illustration of the cases.)

(D29) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$ and $m \models \tau' \mathbin{@} b', i'$, then one and only one of the following four cases holds:
- Compatible: $b' = b$ and $i' = i$ and $\tau \sim \tau'$, in which case $\texttt{load}(\tau', m', b', i') = \lfloor \texttt{convert}(v, \tau') \rfloor$.
- Incompatible: $b' = b$ and $i' = i$ and $\tau \not\sim \tau'$, in which case $\texttt{load}(\tau', m', b', i') = \lfloor \texttt{undef} \rfloor$.
- Disjoint: $b' \neq b$ or $i' + |\tau'| \leq i$ or $i + |\tau| \leq i'$, in which case $\texttt{load}(\tau', m', b', i') = \texttt{load}(\tau', m, b', i')$.

– Overlapping: $b' = b$ and $i' \neq i$ and $i' + |\tau'| > i$ and $i + |\tau| > i'$, in which case $\texttt{load}(\tau', m', b', i') = \lfloor \texttt{undef} \rfloor$.

As previously mentioned, an interesting property of the concrete memory model is that `alloc` never reuses block identifiers, even if some blocks have been deallocated before. To account for this feature, we define the relation $m \# b$, pronounced "block b is fresh in memory m", and defined as $b \geq N$ if $m = (N, B, F, C)$. This relation enjoys the following properties:

(P30) $m \# b$ and $m \models b$ are mutually exclusive.
(P31) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $m \# b$.
(P32) If $\texttt{alloc}(m, l, h) = \lfloor b, m' \rfloor$, then $m' \# b' \;\Leftrightarrow\; b' \neq b \;\wedge\; m \# b'$.
(P33) If $\texttt{store}(\tau, m, b, i, v) = \lfloor m' \rfloor$, then $m' \# b' \;\Leftrightarrow\; m \# b'$.
(P34) If $\texttt{free}(m, b) = \lfloor m' \rfloor$, then $m' \# b' \;\Leftrightarrow\; m \# b'$.

Using the freshness relation, we say that two memory states m_1 and m_2 have the same domain, and write $\mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$, if $\forall b, (m_1 \# b \;\Leftrightarrow\; m_2 \# b)$. In our concrete implementation, two memory states have the same domain if and only if their N components are equal. Therefore, `alloc` is deterministic with respect to the domain of the current memory state: `alloc` chooses the same free block when applied twice to memory states that have the same domain, but may differ in block contents.

(P35) If $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$ and $\texttt{alloc}(m_2, l, h) = \lfloor b_2, m_2' \rfloor$ and $\mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$, then $b_1 = b_2$ and $\mathrm{Dom}(m_1') = \mathrm{Dom}(m_2')$.

The last property of the concrete implementation used in the remainder of this paper is the following: a block b that has been deallocated is both invalid and empty, in the sense that its low and high bounds are equal.

(P36) If $\texttt{free}(m, b) = \lfloor m' \rfloor$, then $\neg(m' \models b)$.
(P37) If $\texttt{free}(m, b) = \lfloor m' \rfloor$, then $\mathcal{L}(m', b) = \mathcal{H}(m', b)$.

5 Memory transformations

We now study the use of the concrete memory model to prove the correctness of program transformations as performed by compiler passes. Most passes of the Compcert compiler preserve the memory behavior of the program: some modify the flow of control, others modify the flow of data not stored in memory, but the memory states before and after program transformation match at every step of the program execution. The correctness proofs for these passes exploit none of the properties of the memory model. However, three passes of the Compcert compiler change the memory behavior of the program, and necessitate extensive reasoning over memory states to be proved correct. We now outline the transformations performed by these three passes.

The first pass that modifies the memory behavior is the translation from the source language Clight to the intermediate language Cminor. In Clight, all variables are allocated in memory: the evaluation environment maps variables to references of memory blocks that contain the current values of the variables. This is consistent with the C specification and the fact that the address of any variable can be taken and used as a memory pointer using the `&` operator. However, this feature renders register allocation and most other optimizations very difficult, because aliasing between a pointer and a variable is always possible.

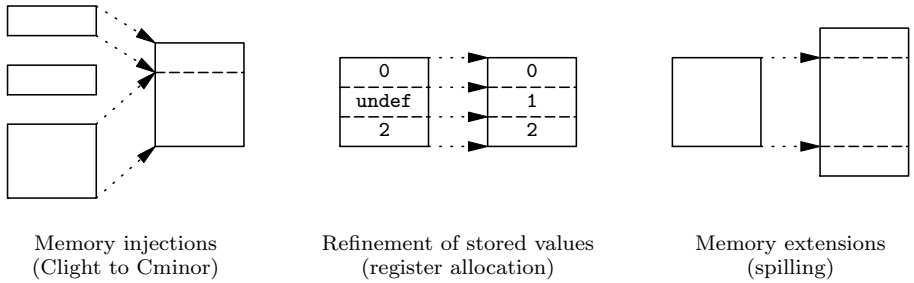


Fig. 2 Transformations over memory states in the Compcert compiler

Therefore, the Clight to Cminor translation detects scalar local variables whose address is never taken with the `&` operator, and "pulls them out of memory": they become Cminor local variables, whose current values are recorded in an environment separate from the memory state, and whose address cannot be taken. Other Clight local variables remain memory-allocated, but are grouped as sub-areas of a single memory block, the Cminor stack block, which is automatically allocated at function entry and deallocated at function exit. (See figure 2, left.)

Consequently, the memory behavior of the source Clight program and the transformed Cminor program differ greatly: when the Clight program allocates N fresh blocks at function entry for its N local variables, the Cminor program allocates only one; the `load` and `store` operations performed by the Clight semantics every time a local variable is accessed either disappear in Cminor or becomes `load` and `store` in sub-areas of the Cminor stack block.

The second pass that affects memory behavior is register allocation. In RTL, the source language for this translation, local variables and temporaries are initialized to the `undef` value on function entry. (This initialization agrees with the semantics of Clight, where reading an uninitialized local variable amounts to loading from a freshly allocated block.) After register allocation, some of these RTL variables and temporaries are mapped to global hardware registers, which are not initialized to the `undef` value on function entry, but instead keep whatever value they had in the caller function at point of call. This does not change the semantics of well-defined RTL programs, since the RTL semantics goes wrong whenever an `undef` value is involved in an arithmetic operation or conditional test. Therefore, values of uninitialized RTL variables do not participate in the computations performed by the program, and can be changed from `undef` to any other value without changing the semantics of the program. However, the original RTL program could have stored these values of uninitialized variables in memory locations. Therefore, the memory states before and after register allocation have the same shapes, but the contents of some memory locations can change from `undef` to any value, as pictured in figure 2, center.

The third and last pass where memory states differ between the original and transformed codes is the spilling pass performed after register allocation. Variables and temporaries that could not be allocated to hardware registers must be "spilled" to memory, that is, stored in locations within the stack frame of the current function. Additional stack frame space is also needed to save the values of callee-save registers on function entry. Therefore, the spilling pass needs to enlarge the stack frame that

was laid out at the time of Cminor generation, to make room for spilled variables and saved registers. The memory state after spilling therefore differs from the state before spilling: stack frame blocks are larger, and the transformed program performs additional `load` and `store` operations to access spilled variables. (See figure 2, right.)

In the three examples of program transformations outlined above, we need to formalize an invariant that relates the memory states at every point of the executions of the original and transformed programs, and prove appropriate simulation results between the memory operations performed by the two programs. Three such relations between memory states are studied in the remainder of this section: memory extensions in section 5.2, corresponding to the spilling pass; refinement of stored values in section 5.3, corresponding to the register allocation pass; and memory injections in section 5.4, corresponding to the Cminor generation pass. These three relations share a common basis, the notion of memory embeddings, defined and studied first in section 5.1.

5.1 Generic memory embeddings

An *embedding* E is a function of type $\texttt{block} \to \texttt{option}(\texttt{block} \times \mathbb{Z})$ that establishes a correspondence between blocks of a memory state m_1 of the original program and blocks of a memory state m_2 of the transformed program. Let b_1 be a block reference in m_1. If $E(b_1) = \varepsilon$, this block corresponds to no block in m_2: it has been eliminated by the transformation. If $E(b_1) = \lfloor b_2, \delta \rfloor$, the block b_1 in m_1 corresponds to the block b_2 in m_2, or a sub-block thereof, with offsets being shifted by δ. That is, the memory location (b_1, i) in m_1 is associated to the location $(b_2, i+\delta)$ in m_2. We say that a block b of m_1 is *unmapped* in m_2 if $E(b) = \varepsilon$, and *mapped* otherwise.

We assume we are given a relation $E \vdash v_1 \hookrightarrow v_2$ between values v_1 of the original program and values v_2 of the transformed program, possibly parametrized by E. (Sections 5.2, 5.3 and 5.4 will particularize this relation).

We say that E embeds memory state m_1 in memory state m_2, and write $E \vdash m_1 \hookrightarrow m_2$, if every successful load from a mapped block of m_1 is simulated by a successful load from the corresponding sub-block in m_2, in the following sense:

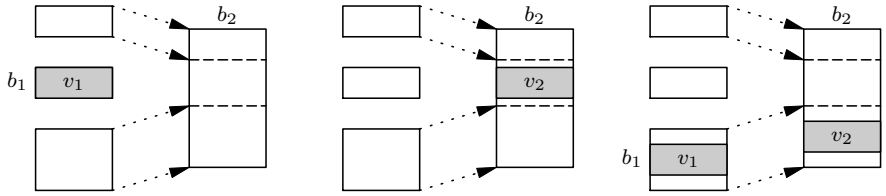
$$\begin{aligned} E(b_1) = \lfloor b_2, \delta \rfloor \ \wedge\ \texttt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor \\ \Rightarrow\ \exists v_2, \texttt{load}(\tau, m_2, b_2, i+\delta) = \lfloor v_2 \rfloor \ \wedge\ E \vdash v_1 \hookrightarrow v_2 \end{aligned}$$

We now state and prove commutation and simulation properties between the memory embedding relation and the operations of the concrete memory model. First, validity of accesses is preserved, in the following sense.

Lemma 38 *If $E(b_1) = \lfloor b_2, \delta \rfloor$ and $E \vdash m_1 \hookrightarrow m_2$, then $m_1 \models \tau \mathbin{@} b_1, i$ implies $m_2 \models \tau \mathbin{@} b_2, i+\delta$.*

Proof By property P25, there exists v_1 such that $\texttt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor$. By hypothesis $E \vdash m_1 \hookrightarrow m_2$, there exists v_2 such that $\texttt{load}(\tau, m_2, b_2, i+\delta) = \lfloor v_2 \rfloor$. The result follows from property P25.

When is the memory embedding relation preserved by memory stores? There are three cases to consider, depicted in figure 3. In the leftmost case, the original program performs a store in memory m_1 within a block that is not mapped, while the transformed program performs no store, keeping its memory m_2 unchanged.



Lemma 39 Lemma 40 Lemma 41

Fig. 3 The three simulation lemmas for memory stores. The grayed areas represent the locations of the stores. v_1 is a value stored by the original program and v_2 a value stored by the transformed program.

Lemma 39 *If* $E(b_1) = \varepsilon$ *and* $E \vdash m_1 \hookrightarrow m_2$ *and* $\mathtt{store}(\tau, m_1, b_1, i, v) = \lfloor m_1' \rfloor$*, then* $E \vdash m_1' \hookrightarrow m_2$.

Proof Consider a load in m_1' from a mapped block: $E(b_1') = \lfloor b_2', \delta \rfloor$ and $\mathtt{load}(\tau', m_1', b_1', i') = \lfloor v_1 \rfloor$. By hypothesis $E(b_1) = \varepsilon$, we have $b_1' \neq b_1$. By S8, it follows that $\mathtt{load}(\tau', m_1, b_1', i') = \mathtt{load}(\tau', m_1', b_1', i') = \lfloor v_1 \rfloor$. The result follows from hypothesis $E \vdash m_1 \hookrightarrow m_2$.

In the second case (figure 3, center), the original program performs no store in its memory m_1, but the transformed program stores some data in an area of its memory m_2 that is disjoint from the images of the blocks of m_1.

Lemma 40 *Let* b_2, i, τ *be a memory reference in* m_2 *such that*

$$\forall b_1, \delta, \quad E(b_1) = \lfloor b_2, \delta \rfloor \;\Rightarrow\; \mathcal{H}(m_1, b_1) + \delta \leq i \;\vee\; i + |\tau| \leq \mathcal{L}(m_1, b_1) + \delta$$

If $E \vdash m_1 \hookrightarrow m_2$ *and* $\mathtt{store}(\tau, m_2, b_2, i, v) = \lfloor m_2' \rfloor$*, then* $E \vdash m_1 \hookrightarrow m_2'$.

Proof Consider a load in m_1 from a mapped block: $E(b_1) = \lfloor b_2', \delta \rfloor$ and $\mathtt{load}(\tau', m_1, b_1, i') = \lfloor v_1 \rfloor$. By P25, this load is within bounds: $\mathcal{L}(m_1, b_1) \leq i'$ and $i' + |\tau'| \leq \mathcal{H}(m_1, b_1)$. By hypothesis $E \vdash m_1 \hookrightarrow m_2$, there exists v_2 such that $\mathtt{load}(\tau, m_2, b_2', i' + \delta) = \lfloor v_2 \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$. We check that the separation condition of S8 holds. This is obvious if $b_2' \neq b_2$. Otherwise, by hypothesis on b_2, either $\mathcal{H}(m_1, b_1) + \delta \leq i$ or $i + |\tau| \leq \mathcal{L}(m_1, b_1) + \delta$. In the first case, $i' + \delta + |\tau'| \leq \mathcal{H}(m_1, b_1) + \delta \leq i$, and in the second case, $i + |\tau| \leq \mathcal{L}(m_1, b_1) + \delta \leq i' + \delta$. Therefore, $\mathtt{load}(\tau', m_2', b_2', i' + \delta) = \mathtt{load}(\tau', m_2, b_2', i' + \delta) = \lfloor v_2 \rfloor$, and the desired result follows.

In the third case (figure 3, right), the original program stores a value v_1 in a mapped block of m_1, while in parallel the transformed program stores a matching value v_2 at the corresponding location in m_2. For this operation to preserve the memory embedding relation, it is necessary that the embedding E is *nonaliasing*. We say that an embedding E is nonaliasing in a memory state m if distinct blocks are mapped to disjoint subblocks:

$$\begin{aligned} & b_1 \neq b_2 \;\wedge\; E(b_1) = \lfloor b_1', \delta_1 \rfloor \;\wedge\; E(b_2) = \lfloor b_2', \delta_2 \rfloor \\ & \quad \Rightarrow\; b_1' \neq b_2' \\ & \quad \vee\; [\mathcal{L}(m, b_1) + \delta_1, \mathcal{H}(m, b_1) + \delta_1) \cap [\mathcal{L}(m, b_2) + \delta_2, \mathcal{H}(m, b_2) + \delta_2) = \emptyset \end{aligned}$$

The disjointness condition between the two intervals can be decomposed as follows: either $\mathcal{L}(m, b_1) \geq \mathcal{H}(m, b_1)$ (block b_1 is empty), or $\mathcal{L}(m, b_2) \geq \mathcal{H}(m, b_2)$ (block b_2 is empty), or $\mathcal{H}(m, b_1) + \delta_1 \leq \mathcal{L}(m, b_2) + \delta_2$, or $\mathcal{H}(m, b_2) + \delta_2 \leq \mathcal{L}(m, b_1) + \delta_1$.

Lemma 41 *Assume* $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$. *Let* v_1, v_2 *be two values and* τ *a type such that* v_1 *embeds in* v_2 *after conversion to any type* τ' *compatible with* τ*:*

$$\forall \tau', \quad \tau \sim \tau' \;\Rightarrow\; E \vdash \texttt{convert}(v_1, \tau') \hookrightarrow \texttt{convert}(v_2, \tau')$$

If $E \vdash m_1 \hookrightarrow m_2$ *and* E *is nonaliasing in the memory state* m_1 *and* $E(b_1) = \lfloor b_2, \delta \rfloor$ *and* $\texttt{store}(\tau, m_1, b_1, i, v_1) = \lfloor m_1' \rfloor$*, then there exists a memory state* m_2' *such that* $\texttt{store}(\tau, m_2, b_2, i + \delta, v_2) = \lfloor m_2' \rfloor$ *and moreover* $E \vdash m_1' \hookrightarrow m_2'$.

Proof The existence of m_2' follows from lemma 38 and property P24. Consider a load in m_1' from a mapped block: $E(b_1') = \lfloor b_2', \delta' \rfloor$ and $\texttt{load}(\tau', m_1', b_1', i') = \lfloor v_1' \rfloor$. By property D29, there are four cases to consider.

- Compatible: $b_1' = b_1$ and $i' = i$ and $\tau \sim \tau'$. In this case, $v_1' = \texttt{convert}(v_1, \tau')$. By S7, we have $\texttt{load}(\tau', m_2', b_2', i' + \delta') = \texttt{load}(\tau', m_2', b_2, i + \delta) = \lfloor \texttt{convert}(v_2, \tau') \rfloor$. The result $E \vdash v_1' \hookrightarrow \texttt{convert}(v_2, \tau')$ follows from the hypothesis over v_1 and v_2.
- Incompatible: $b_1' = b_1$ and $i' = i$ and $\tau \not\sim \tau'$. In this case, $v_1' = \texttt{undef}$. By P27 and P25, we have $\texttt{load}(\tau', m_2', b_2', i' + \delta') = \texttt{load}(\tau', m_2', b_2, i + \delta) = \lfloor \texttt{undef} \rfloor$. The result follows from the hypothesis $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$.
- Disjoint: $b_1' \neq b_1$ or $i' + |\tau'| \leq i$ or $i + |\tau| \leq i'$. In this case, $\texttt{load}(\tau', m_1, b_1', i') = \lfloor v_1' \rfloor$. By hypothesis $E \vdash m_1 \hookrightarrow m_2$, there exists v_2' such that $\texttt{load}(\tau', m_2, b_2', i' + \delta') = \lfloor v_2' \rfloor$ and $E \vdash v_1' \hookrightarrow v_2'$. Exploiting the nonaliasing hypothesis over E and m_1, we show that the separation hypotheses of property S8 hold, which entails $\texttt{load}(\tau', m_2', b_2', i' + \delta') = \lfloor v_2' \rfloor$ and the expected result.
- Overlapping: $b_1' = b_1$ and $i' \neq i$ and $i' + |\tau'| > i$ and $i + |\tau| > i'$. In this case $v_1' = \texttt{undef}$. We show $\texttt{load}(\tau', m_2', b_2', i' + \delta') = \lfloor \texttt{undef} \rfloor$ using P28, and conclude using the hypothesis $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$.

We now turn to relating allocations with memory embeddings, starting with the case where two allocations are performed in parallel, one in the original program, the other in the transformed program.

Lemma 42 *Assume* $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$. *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_1, l_1, h_1) = \lfloor b_1, m_1' \rfloor$ *and* $\texttt{alloc}(m_2, l_2, h_2) = \lfloor b_2, m_2' \rfloor$ *and* $E(b_1) = \lfloor b_2, \delta \rfloor$ *and* $l_2 \leq l_1 + \delta$ *and* $h_1 + \delta \leq h_2$ *and* $\texttt{max\_alignment}$ *divides* δ*, then* $E \vdash m_1' \hookrightarrow m_2'$.

Proof Consider a load in m_1' from a mapped block: $E(b_1') = \lfloor b_2', \delta \rfloor$ and $\texttt{load}(\tau, m_1', b_1', i) = \lfloor v_1 \rfloor$. If $b_1' \neq b_1$, we have $\texttt{load}(\tau, m_1, b_1', i) = \lfloor v_1 \rfloor$ by S5, and there exists v_2 such that $\texttt{load}(\tau, m_2, b_2', i + \delta) = \lfloor v_2 \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$. It must be the case that $b_2' \neq b_2$, otherwise the latter $\texttt{load}$ would have failed (by S9 and P25). The expected result $\texttt{load}(\tau, m_2', b_2', i + \delta) = \lfloor v_2 \rfloor$ follows from S5.

If $b_1' = b_1$, we have $\texttt{load}(\tau, m_1', b_1, i) = \lfloor \texttt{undef} \rfloor$ by P26, and $l_1 \leq i$, $i + |\tau| \leq h_1$ and $|\tau|$ divides i by P25 and S14. It follows that $m_2' \models \tau \mathbin{@} b_2, i + \delta$, and therefore $\texttt{load}(\tau, m_2', b_2, i + \delta) = \lfloor \texttt{undef} \rfloor$ by P25 and P26. This is the expected result since $E \vdash \texttt{undef} \hookrightarrow \texttt{undef}$.

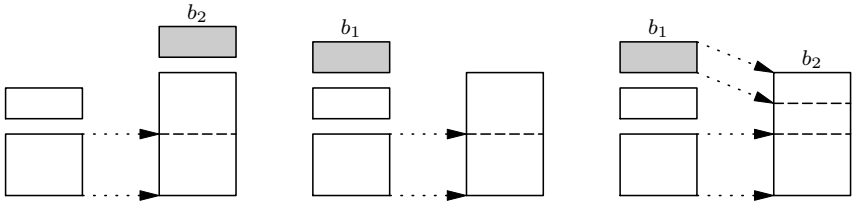


Fig. 4 The three simulation lemmas for memory allocations. The grayed areas represent the freshly allocated blocks.

To complement lemma 42, we also consider the cases where allocations are performed either in the original program or in the transformed program, but not necessarily in both. (See figure 4.) We omit the proof sketches, as they are similar to that of lemma 42.

Lemma 43 *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_2, l, h) = \lfloor b_2, m_2' \rfloor$*, then* $E \vdash m_1 \hookrightarrow m_2'$*.*

Lemma 44 *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$ *and* $E(b_1) = \varepsilon$*, then* $E \vdash m_1' \hookrightarrow m_2$*.*

Lemma 45 *Assume* $E \vdash \texttt{undef} \hookrightarrow \texttt{v}$ *for all values* v*. If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$ *and* $E(b_1) = \lfloor b_2, \delta \rfloor$ *and* $m_2 \models b_2$ *and* $\mathcal{L}(m_2, b_2) \leq l + \delta$ *and* $h + \delta \leq \mathcal{H}(m_2, b_2)$ *and* $\texttt{max\_alignment}$ *divides* δ*, then* $E \vdash m_1' \hookrightarrow m_2$*.*

Finally, we consider the interaction between `free` operations and memory embeddings. Deallocating a block in the original program always preserves embedding.

Lemma 46 *If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{free}(m_1, b_1) = \lfloor m_1' \rfloor$*, then* $E \vdash m_1' \hookrightarrow m_2$*.*

Proof If $\texttt{load}(\tau, m_1', b_1', i) = \lfloor v_1 \rfloor$, it must be that $b_1' \neq b_1$ by P25 and P36. We then have $\texttt{load}(\tau, m_1, b_1', i) = \lfloor v_1 \rfloor$ by S6 and conclude by hypothesis $E \vdash m_1 \hookrightarrow m_2$.

Deallocating a block in the transformed program preserves embedding if no valid block of the original program is mapped to the deallocated block.

Lemma 47 *Assume* $\forall b_1, \delta, \quad E(b_1) = \lfloor b_2, \delta \rfloor \;\Rightarrow\; \neg(m_1 \models b_1)$*. If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{free}(m_2, b_2) = \lfloor m_2' \rfloor$*, then* $E \vdash m_1 \hookrightarrow m_2'$*.*

Proof Assume $E(b_1) = \lfloor b_2', \delta \rfloor$ and $\texttt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor$. It must be the case that $b_2' \neq b_2$, otherwise $m_1 \models b_1$ would not hold, contradicting P25. The result follows from the hypothesis $E \vdash m_1 \hookrightarrow m_2$ and property S6.

Combining lemmas 46 and 47, we see that embedding is preserved by freeing a block b_1 in the original program and in parallel freeing a block b_2 in the transformed program, provided that no block other than b_1 maps to b_2.

Lemma 48 *Assume* $\forall b, \delta, \quad E(b) = \lfloor b_2, \delta \rfloor \;\Rightarrow\; b = b_1$*. If* $E \vdash m_1 \hookrightarrow m_2$ *and* $\texttt{free}(m_1, b_1) = \lfloor m_1' \rfloor$ *and* $\texttt{free}(m_2, b_2) = \lfloor m_2' \rfloor$*, then* $E \vdash m_1' \hookrightarrow m_2'$*.*

```
sp := alloc(0,8)                    sp := alloc(-4,8)
store(int, sp, 0, 42)               store(int, sp, 0, 42)
x := load(int, sp, 0)               x := load(int, sp, 0)
...                                 store(int, sp, -4, x)   // spill
...                                 ...
...                                 x := load(int, sp, -4)  // reload
y := x + x                          y := x + x
free(sp)                            free(sp)
```

Fig. 5 Example of insertion of spill code. Left: original program, right: transformed program. The variable `x` was spilled to the stack location at offset -4. The variable `y` was not spilled.

Finally, it is useful to notice that the nonaliasing property of embeddings is preserved by `free` operations.

Lemma 49 *If E is nonaliasing in m_1, and* $\texttt{free}(m_1, b) = \lfloor m_1' \rfloor$*, then E is nonaliasing in m_1'.*

Proof The block b becomes empty in m_1': by P37, $\mathcal{L}(m_1', b) = \mathcal{H}(m_1', b)$. The result follows from the definition of nonaliasing embeddings.

5.2 Memory extensions

We now instantiate the generic framework of section 5.1 to account for the memory transformations performed by the spilling pass: the transformed program allocates larger blocks than the original program, and uses the extra space to store data of its own (right part of figure 2).

Figure 5 illustrates the effect of this transformation on the memory operations performed by the original and transformed programs. Each `alloc` in the original program becomes an `alloc` operation with possibly larger bounds. Each `store`, `load` and `free` in the original program corresponds to an identical operation in the transformed program, with the same arguments and results. The transformed program contains additional `store` and `load` operations, corresponding to spills and reloads of variables, operating on memory areas that were not accessible in the original program (here, the word at offset -4 in the block `sp`).

To prove that this transformation preserves semantics, we need a relation between the memory states of the original and transformed programs that (1) guarantees that matching pairs of `load` operations return the same value, and (2) is preserved by `alloc`, `store` and `free` operations.

In this section, we consider a fixed embedding E_{id} that is the identity function: $E_{id}(b) = \lfloor b, 0 \rfloor$ for all blocks b. Likewise, we define embedding between values as equality between these values: $E_{id} \vdash v_1 \hookrightarrow v_2$ if and only if $v_1 = v_2$.

We say that a transformed memory state m_2 extends an original memory state m_1, and write $m_1 \subseteq m_2$, if E_{id} embeds m_1 in m_2, and both memory states have the same domain:

$$m_1 \subseteq m_2 \quad \stackrel{\text{def}}{=} \quad E_{id} \vdash m_1 \hookrightarrow m_2 \ \wedge \ \mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$$

The $\subseteq$ relation over memory states is reflexive and transitive. It implies the desired equality between the results of a `load` performed by the initial program and the corresponding `load` after transformation.

Lemma 50 *If* $m_1 \subseteq m_2$ *and* $\texttt{load}(\tau, m_1, b, i) = \lfloor v \rfloor$, *then* $\texttt{load}(\tau, m_2, b, i) = \lfloor v \rfloor$.

Proof Since $E_{id} \vdash m_1 \hookrightarrow m_2$ holds, and $E_{id}(b) = \lfloor b, 0 \rfloor$, there exists a value v' such that $\texttt{load}(\tau, m_2, b, i) = \lfloor v' \rfloor$ and $E_{id} \vdash v \hookrightarrow v'$. The latter entails $v' = v$ and the expected result.

We now show that any `alloc`, `store` or `free` operation over m_1 is simulated by a similar memory operation over m_2, preserving the memory extension relation.

Lemma 51 *Assume* $\texttt{alloc}(m_1, l_1, h_1) = \lfloor b_1, m_1' \rfloor$ *and* $\texttt{alloc}(m_2, l_2, h_2) = \lfloor b_2, m_2' \rfloor$. *If* $m_1 \subseteq m_2$ *and* $l_2 \leq l_1$ *and* $h_1 \leq h_2$, *then* $b_1 = b_2$ *and* $m_1' \subseteq m_2'$.

Proof The equality $b_1 = b_2$ follows from P35. The embedding $E_{id} \vdash m_1' \hookrightarrow m_2'$ follows from lemma 42. The domain equality $\mathrm{Dom}(m_1') = \mathrm{Dom}(m_2')$ follows from P32.

Lemma 52 *Assume* $\texttt{free}(m_1, b) = \lfloor m_1' \rfloor$ *and* $\texttt{free}(m_2, b) = \lfloor m_2' \rfloor$. *If* $m_1 \subseteq m_2$, *then* $m_1' \subseteq m_2'$.

Proof Follows from lemma 48 and property P34.

Lemma 53 *If* $m_1 \subseteq m_2$ *and* $\texttt{store}(\tau, m_1, b, i, v) = \lfloor m_1' \rfloor$, *then there exists* m_2' *such that* $\texttt{store}(\tau, m_2, b, i, v) = \lfloor m_2' \rfloor$ *and* $m_1' \subseteq m_2'$.

Proof Follows from lemma 41 and property P33. By construction, the embedding E_{id} is nonaliasing for any memory state.

Finally, the transformed program can also perform additional stores, provided they fall outside the memory bounds of the original program. (These stores take place when a variable is spilled to memory.) Such stores preserve the extension relation.

Lemma 54 *Assume* $m_1 \subseteq m_2$ *and* $\texttt{store}(\tau, m_2, b, i, v) = \lfloor m_2' \rfloor$. *If* $i + |\tau| \leq \mathcal{L}(m_1, b)$ *or* $\mathcal{H}(m_1, b) \leq i$, *then* $m_1 \subseteq m_2'$.

Proof Follows from lemma 40 and property P33.

5.3 Refinement of stored values

In this section, we consider the case where the original and transformed programs allocate identically-sized blocks in lockstep, but some of the `undef` values produced and stored by the original program can be replaced by more defined values in the transformed program. This situation, depicted in the center of figure 2, occurs when verifying the register allocation pass of Compcert. Figure 6 outlines an example of this transformation.

We say that a value v_1 is refined by a value v_2, and we write $v_1 \leq v_2$, if either $v_1 = \texttt{undef}$ or $v_1 = v_2$. We assume that the `convert` function is compatible with refinements: $v_1 \leq v_2 \Rightarrow \texttt{convert}(v_1, \tau) \leq \texttt{convert}(v_2, \tau)$. (This is clearly the case for the examples of `convert` functions given at the end of section 2.)

We instantiate again the generic framework of section 5.1, using the identity embedding $E_{id} = \lambda b.\ \lfloor b, 0 \rfloor$ and the value embedding

$$E_{id} \vdash v_1 \hookrightarrow v_2 \quad \stackrel{\text{def}}{=} \quad v_1 \leq v_2.$$

```
// x implicitly initialized to undef
sp := alloc(0,8)
store(int, sp, 0, x)
...
y := load(int, sp, 0)
...
free(sp)
```

```
// R1 not initialized
sp := alloc(0,8)
store(int, sp, 0, R1)
...
R2 := load(int, sp, 0)
...
free(sp)
```

Fig. 6 Example of register allocation. Left: original code, right: transformed code. Variables `x` and `y` have been allocated to registers `R1` and `R2`, respectively.

We say that a transformed memory state m_2 refines an original memory state m_1, and write $m_1 \leq m_2$, if E_{id} embeds m_1 in m_2, and both memory states have the same domain:

$$m_1 \leq m_2 \quad \stackrel{\text{def}}{=} \quad E_{id} \vdash m_1 \hookrightarrow m_2 \ \wedge \ \mathrm{Dom}(m_1) = \mathrm{Dom}(m_2)$$

The $\leq$ relation over memory states is reflexive and transitive.

The following simulation properties are immediate consequences of the results from section 5.1.

Lemma 55 *Assume* $\mathtt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$ *and* $\mathtt{alloc}(m_2, l, h) = \lfloor b_2, m_2' \rfloor$. *If* $m_1 \leq m_2$, *then* $b_1 = b_2$ *and* $m_1' \leq m_2'$.

Lemma 56 *Assume* $\mathtt{free}(m_1, b) = \lfloor m_1' \rfloor$ *and* $\mathtt{free}(m_2, b) = \lfloor m_2' \rfloor$. *If* $m_1 \leq m_2$, *then* $m_1' \leq m_2'$.

Proof Follows from lemma 48 and property P34.

Lemma 57 *If* $m_1 \leq m_2$ *and* $\mathtt{load}(\tau, m_1, b, i) = \lfloor v_1 \rfloor$, *then there exists a value* v_2 *such that* $\mathtt{load}(\tau, m_2, b, i) = \lfloor v_2 \rfloor$ *and* $v_1 \leq v_2$.

Lemma 58 *If* $m_1 \leq m_2$ *and* $\mathtt{store}(\tau, m_1, b, i, v_1) = \lfloor m_1' \rfloor$ *and* $v_1 \leq v_2$, *then there exists* m_2' *such that* $\mathtt{store}(\tau, m_2, b, i, v_2) = \lfloor m_2' \rfloor$ *and* $m_1' \leq m_2'$.

5.4 Memory injections

We now consider the most difficult memory transformation encountered in the Compcert development, during the translation from Clight to Cminor: the removal of some memory allocations performed by the Clight semantics and the coalescing of other memory allocations into sub-areas of a single block (see figure 2, left).

The pseudocode in figure 7 illustrates the effect of this transformation on the memory behavior of the program. Here, the transformation elected to "pull `x` out of memory", using a local variable `x` in the transformed program to hold the contents of the block pointed by `x` in the original program. It also merged the blocks pointed by `y` and `z` into a single block pointed by `sp`, with `y` corresponding to the sub-block at offsets $[0, 8)$ and `z` to the sub-block at offsets $[8, 10)$.

To relate the memory states in the original and transformed programs at any given point, we will again reuse the results on generic memory embeddings established in section 5.1. However, unlike in sections 5.2 and 5.3, we cannot work with a fixed embedding E, but need to build it incrementally during the proof of semantic preservation.

```
x := alloc(0, 4)                    sp := alloc(0, 10)
y := alloc(0, 8)
z := alloc(0, 2)
store(int, x, 0, 42)                x := 42
... load(int, x, 0) ...             ... x ...
store(double, y, 0, 3.14)           store(double, sp, 0, 3.14)
... load(short, z, 2) ...           ... load(short, sp, 8) ...
free(x)
free(y)
free(z)                             free(sp)
```

Fig. 7 Example of the Clight to Cminor translation. Left: original program, right: transformed program. Block `x` in the original program is pulled out of memory; its contents are stored in the local variable `x` in the transformed program. Blocks `y` and `z` become sub-blocks of `sp`, at offsets 0 and 8 respectively.

In the Compcert development, we use the following relation between values v_1 of the original Clight program and v_2 of the generated Cminor program, parametrized by an embedding E:

$$E \vdash \mathtt{undef} \hookrightarrow v_2 \qquad E \vdash \mathtt{int}(n) \hookrightarrow \mathtt{int}(n) \qquad E \vdash \mathtt{float}(n) \hookrightarrow \mathtt{float}(n)$$

$$\frac{E(b_1) = \lfloor b_2, \delta \rfloor \qquad i_2 = i_1 + \delta}{E \vdash \mathtt{ptr}(b_1, i_1) \hookrightarrow \mathtt{ptr}(b_2, i_2)}$$

In other words, `undef` Clight values can be refined by any Cminor value; integers and floating-point numbers must not change; and pointers are relocated as prescribed by the embedding E. Notice in particular that if $E(b) = \varepsilon$, there is no Cminor value v such that $E \vdash \mathtt{ptr}(b, i) \hookrightarrow v$. This means that the source Clight program is not allowed to manipulate a pointer value pointing to a memory block that we have decided to remove during the translation.

We assume that the $E \vdash v_1 \hookrightarrow v_2$ relation is compatible with the `convert` function: $E \vdash v_1 \hookrightarrow v_2$ implies $E \vdash \mathtt{convert}(v_1, \tau) \hookrightarrow \mathtt{convert}(v_2, \tau)$. (This clearly holds for the examples of `convert` functions given at the end of section 2.)

We say that an embedding E injects a Clight memory state m_1 in a Cminor memory state m_2, and write $E \vdash m_1 \mapsto m_2$, if the following four conditions hold:

$$\begin{aligned} E \vdash m_1 \mapsto m_2 \quad \stackrel{\text{def}}{=} \quad & E \vdash m_1 \hookrightarrow m_2 && (1)\\ \wedge\ & \forall b_1,\ \ m_1 \# b_1 \ \Rightarrow\ E(b_1) = \varepsilon && (2)\\ \wedge\ & \forall b_1, b_2, \delta,\ \ E(b_1) = \lfloor b_2, \delta \rfloor \ \Rightarrow\ \neg(m_2 \# b_2) && (3)\\ \wedge\ & E \text{ is nonaliasing for } m_1 && (4) \end{aligned}$$

Condition (1) is the embedding of m_1 into m_2 in the sense of section 5.1. Conditions (2) and (3) ensure that fresh blocks are not mapped, and that images of mapped blocks are not fresh. Condition (4) ensures that the embedding does not cause sub-blocks to overlap.

Using this definition, it is easy to show simulation results for the `load` and `store` operations performed by the original program.

Lemma 59 *If $E \vdash m_1 \mapsto m_2$ and $\mathtt{load}(\tau, m_1, b_1, i) = \lfloor v_1 \rfloor$ and $E(b_1) = \lfloor b_2, \delta \rfloor$, then there exists a value v_2 such that $\mathtt{load}(\tau, m_2, b_2, i + \delta) = \lfloor v_2 \rfloor$ and $E \vdash v_1 \hookrightarrow v_2$.*

Proof By (1) and definition of $E \vdash m_1 \hookrightarrow m_2$.

Lemma 60 *If* $E \vdash m_1 \mapsto m_2$ *and* $\texttt{store}(\tau, m_1, b_1, i, v_1) = \lfloor m_1' \rfloor$ *and* $E(b_1) = \lfloor b_2, \delta \rfloor$ *and* $E \vdash v_1 \hookrightarrow v_2$*, then there exists* m_2' *such that* $\texttt{store}(\tau, m_2, b_2, i + \delta, v_2) = \lfloor m_2' \rfloor$ *and* $E \vdash m_1' \mapsto m_2'$.

Proof Follows from lemma 41. Conditions (2), (3) and (4) are preserved because of properties S16 and P33.

Lemma 61 *If* $E \vdash m_1 \mapsto m_2$ *and* $\texttt{store}(\tau, m_1, b_1, i, v_1) = \lfloor m_1' \rfloor$ *and* $E(b_1) = \varepsilon$*, then* $E \vdash m_1' \mapsto m_2'$.

Proof Follows from lemma 39 and properties S16 and P33.

In the Compcert development, given the algebra of values used (see section 2), we can define the following variants `loadptr` and `storeptr` of `load` and `store` where the memory location being accessed is given as a pointer value:

$\texttt{loadptr}(\tau, m, a) =$
  $\texttt{match}\ a\ \texttt{with}\ \texttt{ptr}(b, i) \Rightarrow \texttt{load}(\tau, m, b, i) \mid \_ \Rightarrow \varepsilon$
$\texttt{storeptr}(\tau, m, a, v) =$
  $\texttt{match}\ a\ \texttt{with}\ \texttt{ptr}(b, i) \Rightarrow \texttt{store}(\tau, m, b, i, v) \mid \_ \Rightarrow \varepsilon$

Lemmas 59 and 60 can then be restated in a more "punchy" way, taking advantage of the way $E \vdash v_1 \hookrightarrow v_2$ is defined over pointer values:

Lemma 62 *If* $E \vdash m_1 \mapsto m_2$ *and* $\texttt{loadptr}(\tau, m_1, a_1) = \lfloor v_1 \rfloor$ *and* $E \vdash a_1 \hookrightarrow a_2$*, then there exists a value* v_2 *such that* $\texttt{loadptr}(\tau, m_2, a_2) = \lfloor v_2 \rfloor$ *and* $E \vdash v_1 \hookrightarrow v_2$.

Lemma 63 *If* $E \vdash m_1 \mapsto m_2$ *and* $\texttt{storeptr}(\tau, m_1, b_1, a_1, v_1) = \lfloor m_1' \rfloor$ *and* $E \vdash a_1 \hookrightarrow a_2$ *and* $E \vdash v_1 \hookrightarrow v_2$*, then there exists* m_2' *such that* $\texttt{storeptr}(\tau, m_2, b_2, a_2, v_2) = \lfloor m_2' \rfloor$ *and* $E \vdash m_1' \mapsto m_2'$.

We now relate a sequence of deallocations performed in the original program with a single deallocation performed in the transformed program. (In the example of figure 7, this corresponds to the deallocations of `x`, `y` and `z` on one side and the deallocation of `sp` on the other side.) If l is a list of block references, we define the effect of deallocating these blocks as follows:

$\texttt{freelist}(m, l) =$
  $\texttt{match}\ l\ \texttt{with}$
    $\texttt{nil} \Rightarrow \lfloor m \rfloor$
  $\mid b :: l' \Rightarrow \texttt{match}\ \texttt{free}(m, b)\ \texttt{with}\ \varepsilon \Rightarrow \varepsilon \mid \lfloor m' \rfloor \Rightarrow \texttt{freelist}(m', l')$

Lemma 64 *Assume* $\texttt{freelist}(m_1, l) = \lfloor m_1' \rfloor$ *and* $\texttt{free}(m_2, b_2) = \lfloor m_2' \rfloor$*. Further assume that* $E(b_1) = \lfloor b_2, \delta \rfloor \Rightarrow b_1 \in l$ *for all* b_1, δ*; in other words, all blocks mapped to* b_2 *are in* l *and therefore are being deallocated from* m_1*. If* $E \vdash m_1 \mapsto m_2$*, then* $E \vdash m_1' \mapsto m_2'$.

Proof First, notice that for all $b_1 \in l$, $\neg(m_1' \models b_1)$, by S13 and P36. Part (1) of the expected result then follows from lemmas 46 and 47. Parts (2) and (3) follow from P34. Part (4) follows by repeated application of lemma 49.

Symmetrically, we now consider a sequence of allocations performed by the original program and relate them with a single allocation performed by the transformed program. (In the example of figure 7, this corresponds to the allocations of `x`, `y` and `z` on one side and the allocation of `sp` on the other side.) A difficulty is that the current embedding E needs to be changed to map the blocks allocated by the original program; however, changing E should not invalidate the mappings for pre-existing blocks.

We say that an embedding E' is compatible with an earlier embedding E, and write $E \leq E'$, if, for all blocks b, either $E(b) = \varepsilon$ or $E'(b) = E(b)$. In other words, all blocks that are mapped by E remain mapped to the same target sub-block in E'. This relation is clearly reflexive and transitive. Moreover, it preserves injections between values:

Lemma 65 *If $E \vdash v_1 \hookrightarrow v_2$ and $E \leq E'$, then $E' \vdash v_1 \hookrightarrow v_2$.*

We first state and prove simulation lemmas for one allocation, performed either by the original program or by the transformed program. The latter case is straightforward:

Lemma 66 *If $E \vdash m_1 \mapsto m_2$ and $\mathtt{alloc}(m_2, l, h) = \lfloor b_2, m_2' \rfloor$, then $E \vdash m_1 \mapsto m_2'$.*

Proof Follows from lemma 43 and property P32.

For an allocation performed by the original program, we distinguish two cases: either the new block is unmapped (lemma 67), or it is mapped to a sub-block of the transformed program (lemma 68).

Lemma 67 *Assume $E \vdash m_1 \mapsto m_2$ and $\mathtt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$. Write $E' = E\{b_1 \leftarrow \varepsilon\}$. Then, $E' \vdash m_1' \mapsto m_2$ and $E \leq E'$.*

Proof By part (2) of hypothesis $E \vdash m_1 \hookrightarrow m_2$ and property P31, it must be the case that $E(b_1) = \varepsilon$. It follows that $E' = E$, and therefore we have $E \leq E'$ and $E' \vdash m_1 \hookrightarrow m_2$. Applying lemma 44, we obtain part (1) of the expected result $E' \vdash m_1' \mapsto m_2$. Part (2) follows from P32. Parts (3) and (4) are straightforward.

Lemma 68 *Assume $\mathtt{alloc}(m_1, l, h) = \lfloor b_1, m_1' \rfloor$ and $m_2 \models b_2$ and $\mathcal{L}(m_2, b_2) \leq l + \delta$ and $h + \delta \leq \mathcal{H}(m_2, b_2)$ and `max_alignment` divides δ. Further assume that for all blocks b' and offsets δ',*

$$E(b') = \lfloor b_2, \delta' \rfloor \;\Rightarrow\; \mathcal{H}(m_1, b') + \delta' \leq l + \delta \;\vee\; h + \delta \leq \mathcal{L}(m_1, b') + \delta' \qquad (*)$$

Write $E' = E\{b_1 \leftarrow \lfloor b_2, \delta \rfloor\}$. If $E \vdash m_1 \mapsto m_2$, then $E' \vdash m_1' \mapsto m_2$ and $E \leq E'$.

Proof By part (2) of hypothesis $E \vdash m_1 \hookrightarrow m_2$ and property P31, it must be the case that $E(b_1) = \varepsilon$. It follows that $E \leq E'$.

We first show that $E' \vdash m_1 \hookrightarrow m_2$. Assume $E'(b) = \lfloor b', \delta' \rfloor$ and $\mathtt{load}(\tau, m_1, b, i) = \lfloor v \rfloor$. It must be the case that $b \neq b_1$, since b_1 is not valid in m_1. Therefore, $E(b) = \lfloor b', \delta' \rfloor$ and the result follows from part (1) of hypothesis $E \vdash m_1 \hookrightarrow m_2$ and from lemma 65.

Using lemma 45, we obtain part (1) of the expected result $E' \vdash m_1' \mapsto m_2$. Part (2) follows from P32. Part (3) follows from the fact that b_2 is not fresh (property P30). Finally, part (4) follows from hypothesis (*) and property S15.

We now define the `alloclist` function, which, given a list L of (low, high) bounds, allocates the corresponding blocks and returns both the list B of their references and the final memory state. In the example of figure 7, the allocation of `x`, `y` and `z` corresponds to an invocation of `alloclist` with the list $L = (0, 4); (0, 8); (0, 2)$.

```
alloclist(m, L) =
  match L with
    nil ⇒ ⌊nil, m⌋
  | (l, h) :: L' ⇒
      match alloc(m, l, h) with
        ε ⇒ ε
      | ⌊b, m'⌋ ⇒
          match alloclist(m', L') with
            ε ⇒ ε
          | ⌊B, m''⌋ ⇒ ⌊b :: B, m''⌋
```

Along with the list $L = (l_1, h_1) \ldots (l_n, h_n)$ of allocation requests to be performed in the original program, we assume given the bounds (l, h) of a block to be allocated in the transformed program, and a list $P = p_1, \ldots, p_n$ of elements of type `option` $\mathbb{Z}$, indicating how these allocated blocks should be mapped in the transformed program. If $p_i = \varepsilon$, the i-th block is unmapped, but if $p_i = \lfloor \delta_i \rfloor$, it should be mapped at offset δ_i. In the example of figure 7, we have $l = 0$, $h = 10$, $p_1 = \varepsilon$, $p_2 = \lfloor 0 \rfloor$, and $p_3 = \lfloor 8 \rfloor$.

We say that the quadruple (L, P, l, h) is well-formed if the following conditions hold:

1. L and P have the same length.
2. If $p_i = \lfloor \delta_i \rfloor$, then $l \leq l_i + \delta_i$ and $h_i + \delta_i \leq h$ and `max_alignment` divides δ_i (the image of the i-th block is within bounds and aligned).
3. If $p_i = \lfloor \delta_i \rfloor$ and $p_j = \lfloor \delta_j \rfloor$ and $i \neq j$, then $h_i + \delta_i \leq l_j + \delta_j$ or $h_j + \delta_j \leq l_i + \delta_i$ (blocks are mapped to disjoint sub-blocks).

Lemma 69 *Assume that (L, P, l, h) is well-formed. Assume* `alloclist`$(m_1, L) = \lfloor B, m_1' \rfloor$ *and* `alloc`$(m_2, l, h) = \lfloor b, m_2' \rfloor$. *If $E \vdash m_1 \mapsto m_2$, there exists an embedding E' such that $E' \vdash m_1' \mapsto m_2'$ and $E \leq E'$. Moreover, writing b_i for the i-th element of B and p_i for the i-th element of P, we have $E'(b_i) = \varepsilon$ if $p_i = \varepsilon$, and $E'(b_i) = \lfloor b, \delta_i \rfloor$ if $p_i = \lfloor \delta_i \rfloor$.*

Proof By lemma 66, we have $E \vdash m_1 \mapsto m_2'$. We then show the expected result by induction over the length of the lists L and P, using an additional induction hypothesis: for all b', δ, i, if $E(b') = \lfloor b, \delta \rfloor$ and $p_i = \lfloor \delta' \rfloor$, then $h_i + \delta_i \leq \mathcal{L}(m_1, b') + \delta$ or $\mathcal{H}(m_1, b') + \delta \leq l_i + \delta_i$. In other words, the images of mapped blocks that remain to be allocated are disjoint from the images of the mapped blocks that have already been allocated. The proof uses lemmas 67 and 68. We finish by proving the additional induction hypothesis for the initial state, which is easy since the initial embedding E does not map any block to a sub-block of the fresh block b.

6 Mechanical verification

We now briefly comment on the Coq mechanization of the results presented in this article, which can be consulted on-line at `http://gallium.inria.fr/~xleroy/memory-model/`. The Coq development is very close to what is presented here. Indeed, almost all specifications and statements of theorems given in this article were transcribed directly from the Coq development. The only exception is the definition of well-formed multiple allocation requests at the end of section 5.4, which is presented as inductive predicates in the Coq development, such predicates being easier to

reason upon inductively than definitions involving i-th elements of lists. Also, the Coq development proves additional lemmas not shown in this paper: 8 derived properties in the style of properties D19-D22, used to shorten the proofs of section 5, and 16 properties of auxiliary functions occurring in the concrete implementation of the model, used in the proofs of section 4, especially that of lemma 23.

The mechanized development uses the Coq module system [6] to clearly separate specifications from implementations. The specification of the abstract memory model from section 3, as well as the properties of the concrete memory model from section 4, are given as module signatures. The concrete implementation of the model is a structure that satisfies these two signatures. The derived properties of sections 3 and 4, as well as the memory transformations of section 5, are presented as functors, *i.e.*, modules parametrized by any implementation of the abstract signature or concrete signature, respectively. This use of the Coq module system ensures that the results we proved, especially those of section 5, do not depend on accidental features of our concrete implementation, but only on the properties stated earlier.

The Coq module system was effective at enforcing this kind of abstraction. However, we hit one of its limitations: no constructs are provided to extend *a posteriori* a module signature (interface) with additional declarations and logical properties. The Standard ML and Objective Caml module systems support such extensions through the `open` and `include` constructs, respectively. By lack of such constructs in Coq, the signature of the abstract memory model must be manually duplicated in the signature of the concrete memory model, and later changes to the abstract signature must be manually propagated to the concrete signature. For our development, this limitation was a minor annoyance, but it is likely to cause serious problems for developments that involve a large number of refinement steps.

The Coq development represents approximately 1070 non-blank lines of specifications and statements of theorems, and 970 non-blank lines of proof scripts. Most of the proofs are conducted manually, since Coq does not provide much support for proof automation. However, our proofs intensively use the `omega` tactic, a decision procedure for Presburger arithmetic that automates reasoning about linear equalities and inequalities. The `eauto` (Prolog-style resolution) and `congruence` (equational reasoning via the congruence closure algorithm) tactics were also occasionally useful, but the `tauto` and `firstorder` tactics (propositional and first-order automatic reasoning, respectively) were either too weak (`tauto`) or too inefficient (`firstorder`) to be useful.

As pointed out by one of the reviewers, our formalization is conducted mostly in first-order logic: functions are used as data in sections 4 and 5, but only to implement finite maps, which admit a simple, first-order axiomatization. A legitimate question to ask, therefore, is whether our proofs could be entirely automated using a modern theorem prover for first-order logic. We experimented with this approach using three automatic theorem provers: Ergo [7], Simplify [10] and Z3 [9]. The Why platform for program proof [12] was used to administer the tests and to translate automatically between the input syntaxes of the provers. Most of the Coq development was translated to Why's input syntax. (The only part we did not translate is the concrete implementation of the memory model, because it uses recursive functions that are difficult

	Ergo	Simplify	Z3	At least one
Derived properties from sections 3 and 4	15/15	15/15	15/15	15/15
Generic memory embeddings (section 5.1)	7/12	1/12	6/12	9/12
Memory extensions (section 5.2)	0/7	1/7	5/7	5/7
Refinement of stored values (section 5.3)	2/8	3/8	6/8	6/8
Memory injections (section 5.4)	4/8	4/8	7/8	7/8
Total	28/50	24/50	39/50	42/50

Table 1 Experiments with three automated theorem provers

to express in this syntax.) Each derived property and lemma was given to the three provers as a goal, with a time limit of 5 minutes of CPU time.[4]

Table 1 summarizes the results of this experiment. A total of 50 goals were given to the three provers: the derived properties D19-D22 and D29 from sections 3 and 4, all lemmas from section 5, and some auxiliary lemmas present in the Coq development. Of these 50 goals, 42 were proved by at least one of the three provers. The 8 goals that all three provers fail to establish within the 5-minute time limit correspond to lemmas 41, 42, 48, 51, 53, 55, 58, and 67 from section 5. These preliminary results are encouraging: while interactive proof remains necessary for some of the most difficult theorems, integration of first-order theorem proving within a proof assistant has great potential to significantly shorten our proofs.

7 Related work

Giving semantics to imperative languages and reasoning over pointer programs has been the subject of much work since the late 1960's. Reynolds [23] and Tennent and Ghica [26] review some of the early work in this area. In the following, we mostly focus on semantics and verifications that have been mechanized.

For the purpose of this discussion, memory models can be roughly classified as either "high level", where the model itself provides some guarantees of separation, enforcement of memory bounds, etc., or "low-level", where the memory is modeled essentially as an array of bytes and such guarantees must be enforced through additional logical assertions.

A paradigmatic example of high-level modeling is the Burstall-Bornat encoding of records (`struct`), where each field is viewed as a distinct memory store mapping addresses to contents [5, 4]. Such a representation captures the fact that distinct fields of a `struct` value are separated: it becomes obvious that assigning to one field though a pointer (`p->f = x` in C) leaves unchanged the values of any other field. In turn, this separation guarantee greatly facilitates reasoning over programs that manipulate linked data structures, as demonstrated by Mehta and Nipkow [19] and the Caduceus program prover of Filliâtre and Marché [11]. However, this representation makes it difficult to account for other features of the C language, such as `union` types and some casts between pointer types.

Examples of low-level modeling of memory include Norrish's HOL semantics for C [21] and the work of Tuch, Klein and Norrish [27]. There, a memory state is essentially

[4] The test was run on a 2.4 GHz Intel Core 2 Duo processor, with 2 Gb of RAM, running the MacOS 10.4 operating system. The versions of the provers used are: Ergo 0.7.2, compiled with OCaml version 3.10.1; Simplify 1.5.5; Z3 1.1, running under CrossOver Mac.

a mapping from addresses to bytes, and allocation, loads and stores are axiomatized in these terms. The axioms can either leave unspecified all behaviors undefined in the C standard, or specify additional behaviors arising from popular violations of the C standard such as casts between incompatible pointer types. Reasoning about programs and program transformations is more difficult than with a high-level memory model; Tuch, Klein and Norrish [27] use separation logic to alleviate these difficulties.

The memory model presented in this article falls half-way between high-level models and low-level models. It guarantees several useful properties: separation between blocks obtained by distinct calls to `alloc`, enforcement of bounds during memory accesses, and the fact that loads overlapping a prior store operation predictably return the `undef` value. These properties play a crucial role in verifying the program transformations presented in section 5. In particular, a lower-level memory model where a load overlapping a previous store could return an unspecified value would invalidate the simulation lemmas for memory injections (section 5.4). On the other hand, the model offers no separation guarantees over accesses performed within the same memory block. The natural encoding of a `struct` value as a single memory block does not, by itself, validate the Burstall-Bornat separation properties; additional reasoning over field offsets is required.

Separation logic, introduced by O'Hearn, Reynolds and Yang [22,24], and the related spatial logic of Jia and Walker [14], provide an elegant way to reason over memory separation properties of pointer programs. Central to these approaches is the separating conjunction $P*Q$, which guarantees that the logical assertions P and Q talk about disjoint areas of the memory state. Examples of use of separation logic include correctness proofs for memory allocators and garbage collectors [17,18]. It is possible, but not very useful, to define a separation logic on top of our memory model, where in a separating conjunction $P * Q$, every memory block is wholly owned by either P or Q but not both. Appel and Blazy [1] develop a finer-grained separation logic for the Cminor intermediate language of Compcert where disjoint parts of a given block can be separated.

While intended for sequential programs, the memory model described in this paper can also be used to describe concurrent executions in a strongly-consistent shared memory context, where the memory effect of a concurrent program is equivalent to an interleaving of the load and store operations performed by each of its threads. Modern multiprocessor systems implement weakly-consistent forms of shared memory, where the execution of a concurrent program cannot be described as such interleavings of atomic load and store operations. The computer architecture community has developed sophisticated hardware memory models to reason about weakly-consistent memory. For instance, Shen, Arvind and Rudolph [25] use a term rewriting system to define a memory model that decomposes load and store operations into finer-grained operations. This model formalizes the notions of data replication and instruction reordering. It aims at defining the legal behaviors of a distributed shared-memory system that relies on execution trace of memory accesses. Another example of architecture-centric memory model is that of Yang, Gopalakrishnan and Lindstrom [28].

Going back to memory models for programming languages, features of architectural models for weakly-consistent shared memory also appear in specifications of programming languages that support shared-memory concurrency. A famous example is Java, whose specification of its memory model has gone through several iterations. Manson, Pugh and Adve [16] describe and formalize the latest version of the Java memory model. Reasoning over concurrent, lock-free programs in Java or any other shared-memory,

weakly-consistent concurrency model remains challenging. Reasoning over transformations of such programs is an open problem.

Software written in C, especially systems code, often makes assumptions about the layout of data in memory and the semantics of memory accesses that are left unspecified by the C standard. Recent work by Nita, Grossman and Chambers [20] develops a formal framework to characterize these violations of the C standard and to automatically detect the portability issues they raise. Central to their approach is the notion of a *platform*, which is an abstract description of the assumptions that non-portable code makes about concrete data representations. Some aspects of their notion of platform are captured by our memory model, via the size and alignment functions and the type compatibility relation from section 2. However, our model does not account for many other aspects of platforms, such as the layout and padding algorithm for `struct` types.

8 Conclusions

We have presented and formalized a software memory model at a level of abstraction that is intermediate between the high-level view of memory that underlies the C standard and the low-level view of memory that is implemented in hardware. This memory model is adequate for giving semantics and reasoning over intermediate languages typically found in compilers. In particular, the main features of our model (separation between blocks, bounds checking, and the `undef` value resulting from ill-defined loads) played a crucial role in proving semantics preservation for the Compcert verified compiler.

This model can also be used to give a concrete semantics for the C language that specifies the behavior of a few popular violations of the C standard, such as arbitrary casts between pointers, as well as pointer arithmetic within `struct` types. However, many other violations commonly used in systems code or run-time systems for programming languages (for instance, copying arrays of characters 4 or 8 elements as a time using integer or floating-point loads and stores) cannot be accounted for in our model. We have considered several variants of our model that could give meaning to these idioms, but have not yet found one that would still validate all simulation properties of section 5. This remains an important direction for future work, since such a model would be useful not only to reason over systems C code, but also to prove that the semantics of such code is preserved during compilation by the Compcert compiler.

Another direction for future work is to construct and prove correct refinements from a high-level model such as the Burstall-Bornat model used in Caduceus [11] to our model, and from our model to a low-level, hardware-oriented memory model. Such refinements would strengthen the proof of semantic preservation of the Compcert compiler, which currently uses a single memory model for the source and target languages.

Acknowledgements Our early explorations of memory models for the Compcert project were conducted in collaboration with Benjamin Grégoire and François Armand, and benefited from discussions with Catherine Dubois and Pierre Letouzey. Sylvain Conchon, Jean-Christophe Filliâtre and Benjamin Monate helped us experiment with automatic theorem provers. We thank the anonymous reviewers as well as Nikolay Kosmatov for their careful reading of this article and their helpful suggestions for improvements.

References

1. Appel, A.W., Blazy, S.: Separation logic for small-step Cminor. In: Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007, *Lecture Notes in Computer Science*, vol. 4732, pp. 5–21. Springer (2007)
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. EATCS Texts in Theoretical Computer Science. Springer (2004)
3. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: FM 2006: Int. Symp. on Formal Methods, *Lecture Notes in Computer Science*, vol. 4085, pp. 460–475. Springer (2006)
4. Bornat, R.: Proving pointer programs in Hoare logic. In: MPC '00: Proc. Int. Conf. on Mathematics of Program Construction, *Lecture Notes in Computer Science*, vol. 1837, pp. 102–126. Springer (2000)
5. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. Machine Intelligence **7**, 23–50 (1972)
6. Chrząszcz, J.: Modules in type theory with generative definitions. Ph.D. thesis, Warsaw University and University of Paris-Sud (2004)
7. Conchon, S., Contejean, E., Kanig, J.: The Ergo automatic theorem prover. Software and documentation available at `http://ergo.lri.fr/` (2005–2008)
8. Coq development team: The Coq proof assistant. Software and documentation available at `http://coq.inria.fr/` (1989–2008)
9. De Moura, L., Bjørner, N., et al.: Z3: an efficient SMT solver. Software and documentation available at `http://research.microsoft.com/projects/z3` (2006–2008)
10. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
11. Filliâtre, J.C., Marché, C.: Multi-Prover Verification of C Programs. In: 6th Int. Conf. on Formal Engineering Methods, ICFEM 2004, *Lecture Notes in Computer Science*, vol. 3308, pp. 15–29. Springer (2004)
12. Filliâtre, J.C., Marché, C., Moy, Y., Hubert, T.: The Why software verification platform. Software and documentation available at `http://why.lri.fr/` (2004-2008)
13. ISO: International standard ISO/IEC 9899:1999, Programming languages – C (1999)
14. Jia, L., Walker, D.: ILC: A foundation for automated reasoning about pointer programs. In: Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, *Lecture Notes in Computer Science*, vol. 3924, pp. 131–145. Springer (2006)
15. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd symposium Principles of Programming Languages, pp. 42–54. ACM Press (2006)
16. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: 32nd symposium Principles of Programming Languages, pp. 378–391. ACM Press (2005)
17. Marti, N., Affeldt, R., Yonezawa, A.: Formal verification of the heap manager of an operating system using separation logic. In: Formal Methods and Software Engineering, 8th Int. Conf. ICFEM 2006, *Lecture Notes in Computer Science*, vol. 4260, pp. 400–419. Springer (2006)
18. McCreight, A., Shao, Z., Lin, C., Li, L.: A general framework for certifying garbage collectors and their mutators. In: Programming Language Design and Implementation 2007, pp. 468–479. ACM Press (2007)
19. Mehta, F., Nipkow, T.: Proving pointer programs in higher-order logic. Information and Computation **199**(1–2), 200–227 (2005)
20. Nita, M., Grossman, D., Chambers, C.: A theory of platform-dependent low-level software. In: 35th symposium Principles of Programming Languages. ACM Press (2008)
21. Norrish, M.: C formalized in HOL. Ph.D. thesis, University of Cambridge (1998). Technical report UCAM-CL-TR-453
22. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Computer Science Logic, 15th Int. Workshop, CSL 2001, *Lecture Notes in Computer Science*, vol. 2142, pp. 1–19. Springer (2001)
23. Reynolds, J.C.: Intuitionistic reasoning about shared data structures. In: J. Davies, B. Roscoe, J. Woodcock (eds.) Millenial Perspectives in Computer Science, pp. 303–321. Palgrave (2000)
24. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th symposium on Logic in Computer Science (LICS 2002), pp. 55–74. IEEE Computer Society Press (2002)

25. Shen, X., Arvind, Rudolph, L.: Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers. In: ISCA '99: Proc. int. symp. on Computer Architecture, pp. 150–161. IEEE Computer Society Press (1999)
26. Tennent, R.D., Ghica, D.R.: Abstract models of storage. Higher-Order and Symbolic Computation **13**(1–2), 119–129 (2000)
27. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: 34th symposium Principles of Programming Languages, pp. 97–108. ACM Press (2007)
28. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: UMM: an operational memory model specification framework with integrated model checking capability. Concurrency and Computation: Practice and Experience **17**(5–6), 465–487 (2005)