A Concrete Memory Model for CompCert

Frédéric Besson, Sandrine Blazy, Pierre Wilke

A Concrete Memory Model for CompCert*

Frédéric Besson[1], Sandrine Blazy[2], and Pierre Wilke[2]

[1] Inria, Rennes, France
[2] Université Rennes 1 - IRISA, Rennes, France

Abstract. Semantics preserving compilation of low-level C programs is challenging because their semantics is implementation defined according to the C standard. This paper presents the proof of an enhanced and more concrete memory model for the CompCert C compiler which assigns a definite meaning to more C programs. In our new formally verified memory model, pointers are still abstract but are nonetheless mapped to concrete 32-bit integers. Hence, the memory is finite and it is possible to reason about the binary encoding of pointers. We prove that the existing memory model is an abstraction of our more concrete model thus validating formally the soundness of CompCert's abstract semantics of pointers. We also show how to adapt the front-end of CompCert thus demonstrating that it should be feasible to port the whole compiler to our novel memory model.

1 Introduction

Formal verification of programs is usually performed at source level. Yet, a theorem about the source code of a safety critical software is not sufficient. Eventually, what we really value is a guarantee about the run-time behaviour of the compiled program running on a physical machine. The CompCert compiler [17] fills this verification gap: its semantics preservation theorem ensures that when the source program has a defined semantics, program invariants proved at source level still hold for the compiled code. For the C language the rules governing so called *undefined behaviours* are subtle and the absence of undefined behaviours is in general undecidable. As a corollary, for a given C program, it is undecidable whether the semantic preservation applies or not.

To alleviate the problem, the semantics of CompCert C is executable and it is therefore possible to check that a given program execution has a defined semantics. Jourdan *et al.* [12] propose a more comprehensive and ambitious approach: they formalise and verify a precise C static analyser for CompCert capable of ruling out undefined behaviours for a wide range of programs. Yet, these approaches are, by essence, limited by the formal semantics of CompCert C: programs exhibiting undefined behaviours cannot benefit from any semantic preservation guarantee. This is unfortunate as real programs do have behaviours that are undefined according to the formal semantics of CompCert C[3]. This can

*This work was partially supported by the French ANR-14-CE28-0014 AnaStaSec.

[3]The official C standard is in general even stricter.

be a programming mistake but sometimes this is a design feature. In the past, serious security flaws have been introduced by optimising compilers aggressively exploiting the latitude provided by undefined behaviours [22,6]. The existing workaround is not satisfactory and consists in disabling optimisations known to be triggered by undefined behaviours.

In previous work [3], we proposed a more concrete and defined semantics for CompCert C able to give a semantics to low-level C idioms. This semantics relies on symbolic expressions stored in memory that are normalised into genuine values when needed by the semantics. It handles low-level C idioms that exploit the concrete encoding of pointers (e.g. alignment constraints) or access partially undefined data structures (e.g. bit-fields). Such properties cannot be reasoned about using the existing CompCert memory model [19,18].

The memory model of CompCert consists of two parts: standard operations on memory (e.g. alloc, store) that are used in the semantics of the languages of CompCert and their properties (that are required to prove the semantic preservation of the compiler), together with generic transformations operating over memory. Indeed, certain passes of the compiler perform non-trivial transformations on memory allocations and accesses: for instance, in the front-end, C local variables initially mapped to individually-allocated memory blocks are later on mapped to sub-blocks of a single stack-allocated activation record. Proving the semantic preservation of these transformations requires extensive reasoning over memory states, using memory invariants relating memory states during program execution, that are also defined in the memory model.

In this paper, we extend the memory model of CompCert with symbolic expressions [3] and tackle the challenge of porting memory transformations and CompCert's proofs to our memory model with symbolic expressions. The complete Coq development is available online [1]. Among others, a difficulty is that we drop the implicit assumption of an infinite memory. This has the consequence that allocation can fail. Hence, the compiler has to ensure that the compiled program is using less memory than the source program.

This paper presents a milestone towards a CompCert compiler adapted with our semantics; it makes the following contributions.

- We present a formal verification of our memory model within CompCert.
- We prove that the existing memory model of CompCert is an abstraction of our model thus validating the soundness of the existing semantics.
- We extend the notion of memory injection, the main generic notion of memory transformation.
- We adapt the proof of CompCert's front-end passes, from CompCert C until Cminor, thus demonstrating the feasibility of our endeavour.

The paper is organised as follows. Section 2 recalls the main features of the existing CompCert memory model and our proposed extension. Section 3 explains how to adapt the operations of the existing CompCert memory model to comply with the new requirements of our memory model. Section 4 shows that the existing memory model is, in a provable way, an abstraction of our new memory model. Section 5 presents our re-design of the notion of memory

injection that is the cornerstone of compiler passes modifying the memory layout. Section 6 details the modifications for the proofs for the compiler front-end passes. Related work is presented in Section 7; Section 8 concludes.

2 A More Concrete Memory Model for CompCert

In previous work [3], we propose an enhanced memory model (with symbolic expressions) for CompCert. The model is implemented and evaluated over a representative set of C programs. We empirically verify, using the reference interpreter of CompCert, that our extension is sound with respect to the existing semantics and that it captures low-level C idioms out of reach of the existing memory model. This section first recalls the main features of the current CompCert memory model and then explains our extension to this memory model.

2.1 CompCert's Memory Model

Leroy *et al.* [18] give a thorough presentation of the existing memory model of CompCert, that is shared by all the languages of the compiler. We give a brief overview of its design in order to highlight the differences with our own model.

Abstract values used in the semantics of the CompCert languages (see [19]) are the disjoint union of 32-bit integers (written as `int(i)`), 32-bit floating-point numbers (written as `float(f)`), locations (written as `ptr(l)`), and the special value `undef` representing an arbitrary bit pattern, such as the value of an uninitialised variable. The abstract memory is viewed as a collection of separated blocks. A location `l` is a pair `(b, i)` where `b` is a block identifier (i.e. an abstract address) and `i` is an integer offset within this block. Pointer arithmetic modifies the offset part of a location, keeping its block identifier part unchanged. A pointer `ptr(b, i)` is valid for a memory `M` (written `valid_pointer(M, b, i)`) if the offset `i` is within the two bounds of the block `b`.

Abstract values are loaded from (resp. stored into) memory using the `load` (resp. `store`) memory operation. Memory chunks appear in these operations, to describe concisely the size, type and signedness of the value being stored. These operations return option types: we write $\emptyset$ for failure and $\lfloor x \rfloor$ for a successful return of a value x. The `free` operation may also fail (e.g. when the locations to be freed have been freed already). The memory operation `alloc` never fails, as the size of the memory is unbounded.

In the memory model, the byte-level, in-memory representation of integers and floats is exposed, while pointers are kept abstract [18]. The concrete memory is modelled as a map associating to each location a concrete value cv that is a byte-sized quantity describing the current content of a memory cell. It can be either a concrete 8-bit integer (written as `bytev`(b)) representing a part of an integer or a float, `ptrv`(l, i) to represent the i-th byte ($i \in \{1, 2, 3, 4\}$) of the location l, or `undefv` to model uninitialised memory.

```
struct {
  int a0 : 1; int a1 : 1;
} bf;

int main() {
 bf.a1 = 1; return bf.a1;}
```

(a) Bitfield in C

```
1 struct {  unsigned char __bf1;} bf;
2
3 int main(){
4  bf.__bf1 = (bf.__bf1 & ~0x2U) |
5             ((unsigned int) 1 << 1U & 0x2U);
6  return (int) (bf.__bf1 << 30) >> 31;}
```

(b) Bitfield in CompCert C

Fig. 1: Emulation of bitfields in CompCert

2.2 Motivation for an Enhanced Memory Model

Our memory model with symbolic expressions [3] gives a precise semantics to low-level C idioms which cannot be modelled by the existing memory model. The reason is that those idioms either exploit the binary representation of pointers as integers or reason about partially uninitialised data. For instance, it is common for system calls, e.g. `mmap` or `sbrk`, to return -1 (instead of a pointer) to indicate that there is no memory available. Intuitively, -1 refers to the last memory address `0xFFFFFFFF` and this cannot be a valid address because `mmap` returns pointers that are aligned – their trailing bits are necessarily 0s. Other examples are robust implementations of `malloc`: for the sake of checking the integrity of pointers, their trailing bits store a checksum. This is possible because those pointers are also aligned and therefore the trailing bits are necessarily 0s.

Another motivation is illustrated by the current handling of bitfields in CompCert: they are emulated in terms of bit-level operations by an elaboration pass preceding the formally verified front-end. Fig. 1 gives an example of such a transformation. The program defines a bitfield `bf` such that `a0` and `a1` are 1 bit long. The `main` function sets the field `a1` of `bf` to 1 and then returns this value. The expected semantics is therefore that the program returns 1. The transformed code (Fig. 1b) is not very readable but the gist of it is that field accesses are encoded using bitwise and shift operators. The transformation is correct and the target code generated by CompCert correctly returns 1. However, using the existing memory model, the semantics is undefined. Indeed, the program starts by reading the field `__fd1` of the uninitialised structure `bf`. The value is therefore `undef`. Moreover, shift and bitwise operators are strict in `undef` and therefore return `undef`. As a result, the program returns `undef`. As we show in the next section, our semantics is able to model partially undefined values and therefore gives a semantics to bitfields. Even though this case could be easily solved by modifying the pre-processing step, C programmers might themselves write such low-level code with reads of undefined memory and expect it to behave correctly.

2.3 A Memory Model with Symbolic Expressions

To give a semantics to the previous idioms, a direct approach is to have a fully concrete memory model where a pointer is a genuine integer and the memory is

```
Record compat(cm, m) : Prop := {
 addr_space: ∀ b o, [o] ∈ bound(m, b) → 0 < [cm(b)+o] < Int.max_unsigned;
 overlap : ∀  b b' o o',
    b ≠ b' → [o] ∈ bound(m,b) → [o'] ∈ bound(m, b') → [cm(b)+o] ≠ [cm(b')+o'];
 alignment: ∀ b,  cm(b) & alignt(m, b) = cm(b) }.
```

Fig. 2: Compatibility relation betw. a memory mapping and an abstract memory

an array of bytes. However, this model lacks an essential property of CompCert's semantics: determinism. For instance, with a fully concrete memory model, allocating a memory chunk returns a non-deterministic pointer – one of the many that does not overlap with an already allocated chunk. In CompCert, the allocation returns a block that is computed in a deterministic fashion. Determinism is instrumental for the simulation proofs of the compiler passes and its absence is a show stopper.

Our approach to increase the semantics coverage of CompCert consists in delaying the evaluation of expressions which, for the time being, may have a non-deterministic evaluation. In memory, we therefore consider side-effect free symbolic expressions of the following form: $sv ::= v \mid \mathtt{op}_1^\tau\ sv \mid sv\ {}^\tau\mathtt{op}_2{}^\tau\ sv$, where v is a value and $\mathtt{op}_1^\tau$ (resp. ${}^\tau\mathtt{op}_2^\tau$) is a unary (resp. binary) arithmetic operator.

Our memory model is still made of a collection of blocks with the difference that 1) each block possesses an explicit alignment, and 2) memory blocks contain symbolic byte expressions. The alignment of a block `b` (written `alignt(m, b)`) states that the address of `b` has a binary encoding such that its trailing `alignt(m, b)` bits are zeros. Memory `load`s and `store`s cannot be performed on symbolic expressions which therefore need to be normalised beforehand to a genuine pointer value. Another place where normalisation is needed is before a conditional jump to ensure determinism of the jump target. The normalisation gives a semantics to expressions in terms of a concrete mapping from blocks to addresses. Formally, we define in Fig. 2 a compatibility relation stating that:[4]

- a valid location is in the range [`0x00000001`; `0xFFFFFFFE`] (see `addr_space`);
- valid locations from distinct blocks do not overlap (see `overlap`);
- a block is mapped to an address abiding to the alignment constraints.

The normalisation is such that `normalise` $m\ e\ \tau$ returns a value v of type τ if and only if the side-effect free expression e evaluates to v for every concrete mapping `cm`: $block \to \mathbb{B}_{32}$ of blocks to concrete 32 bits addresses which are compatible with the block-based memory m (written `compat(cm, m)`).

We define the evaluation of expressions as the function $[\![\cdot]\!]_{\mathtt{cm}}$, parametrised by the concrete mapping `cm`. Pointers are turned into their concrete value, as dictated by `cm`. For example, the evaluation of $\mathtt{ptr}(b, o)$ results in $\mathtt{cm}(b) + o$. Then, symbolic operations are mapped to the corresponding value operations.

Consider the code of Fig. 1b. Unlike the existing semantics, operators are not strict in `undef` but construct symbolic expressions. Hence, in line 4, we store

[4]The notation [i] denotes the machine integer i when interpreted as unsigned.

in `bf`. `__bf1` the symbolic expression e defined by (undef&~0x2U)|(1<<1U&0x2U) and therefore return the normalisation of the expression (e << 30) >> 31. The value of the expression is 1 whatever the value of `undef` and therefore the normalisation succeeds and returns, as expected, the value 1.

3 Proving the Operations of the Memory Model

CompCert's memory model exports an interface summarising all the properties of the memory operations necessary to prove the compiler passes. This section details how the properties and the proofs need to be adapted to accommodate for symbolic expressions. First, we have to refine our handling of undefined values. Second, we introduce an equivalence relation between symbolic expressions.

3.1 Precise Handling of Undefined Values

Symbolic expressions (as presented in Section 2.3) feature a unique `undef` token. This is a shortcoming that we have identified during the proof. With a single `undef`, we do not capture the fact that different occurrences of `undef` may represent the same unknown value, or different ones. For instance, consider two uninitialised `char` variables `x` and `y` . Expressions `x − x` and `x − y` both construct the symbolic expression `undef` − `undef`, which does not normalise. However we would like `x − x` to normalise to 0, since whatever the value stored in memory for `x`, say v, the result of $v - v$ should always be 0. To overcome this problem, each byte of a newly allocated memory chunk is initialised with a fresh `undef` value. After the allocation of block b, the value stored at location (b, o) is $\texttt{undef}(b, o)$. This value is fresh because block identifiers are never reused. Hence, `x − x` constructs the symbolic expression $\texttt{undef}(b, o) - \texttt{undef}(b, o)$ for some b and o which obviously normalises to 0, because $\texttt{undef}(b, o)$ now represents a unique value rather than the set of all values. Our symbolic expressions do not use the existing CompCert's values but the following type $\texttt{sval} ::= \texttt{undef}(l) \mid \texttt{int}(i) \mid \texttt{float}(f) \mid \texttt{ptr}(l)$.

3.2 Memory Allocation

CompCert's `alloc` operation always allocates a memory chunk of the requested size and returns a fresh block to the newly allocated memory (i.e. it models an infinite memory). In our model, memory consumption needs to be precisely monitored and a memory `m` is a dependent record which, in addition to a CompCert block-based memory, provides guarantees about the concrete memories compatible (see Fig. 2) with the CompCert block-based memory. Hence, our `alloc` operation is partial and returns $\emptyset$ if it fails to construct a memory object.

The first guarantee is that for every memory `m` there exists at least a concrete memory compatible with the abstract CompCert block-based memory.

```
Lemma mem_compat: ∀ m, ∃ cm, compat(cm, m).
```

To get this property, the `alloc` function runs a greedy algorithm constructing a compatible `cm` mapping. Given a memory `m`, `size_mem(m)` returns the size of the constructed memory (i.e. the first fresh address as computed by the allocation). The algorithm makes the pessimistic assumption that the allocated blocks are maximally aligned – for CompCert, this maximum is 3 bits (addresses are divisible by 2^3). It places the allocated block at the first concrete address that is free and compliant with a 3-bit alignment. Allocation fails if no such address can be found. The rationale for the pessimistic alignment assumption is discussed in Section 5: it is essential to ensure that certain memory re-arrangements are always feasible (i.e. would not exhaust memory).

Without the `mem_compat` property, a symbolic expression e could have a normalisation v before allocation and be undefined after allocation. The existence of a concrete compatible memory ensures that the normalisation of expressions is strictly more defined after allocation.

```
Lemma normalise_alloc : ∀ m lo hi m' b e τ v,
 alloc m lo hi = ⌊(m', b)⌋ ∧ normalise m e τ = v ∧ v ≠ undef  →
 normalise m' e τ = v.
```

3.3 Good Variable Properties

In CompCert, the so-called *good variable properties* axiomatise the behaviour of the memory operations. For example, the property `load_store_same` states that, starting from memory m_1, if we store value v at location l with some chunk κ, then when we read at l with κ, we get back the value v. During a `store`, a symbolic expression is split into symbolic bytes using the function `extr(sv, i)` which extracts the i^{th} byte of a symbolic expression `sv`. The reverse operation is the concatenation of a symbolic expression $\texttt{sv}_1$ with a symbolic expression $\texttt{sv}_2$ representing a byte. These operations can be defined as `extr(sv, i)` $=$ `(sv shr (8*i)) & 0xFF` and `concat(`$\texttt{sv}_1$`,`$\texttt{sv}_2$`)` $= \texttt{sv}_1$ `shl 8 +` $\texttt{sv}_2$.

As a result, the axiom `load_store_same` needs to be generalised because the stored value v is not syntactically equal to the loaded value v_1 but is equal modulo normalisation (written $v_1 \equiv v_2$). This equivalence relation is a key insight for generalising the properties and the proofs of the memory model.

```
Axiom load_store_same: ∀ κ m1 b ofs v m2, store κ m1 b ofs v = ⌊ m2 ⌋ →
 ∃ v1, load chunk m2 b ofs = ⌊ v1 ⌋ ∧ v1 ≡ Val.load_result κ v.
```

We have generalised and proved the axioms of the memory model using the same principle. The proof effort is non-negligible as the memory model exports more than 150 lemmas. Moreover, if the structure of the proofs is similar, our proofs are complicated by the fact that we reason modulo normalisation of expressions.

4 Cross-validation of Memory Models

The semantics of the CompCert C language is part of the *trusted computing base* of the compiler. Any modelling error can be responsible for a buggy, though formally verified, compiler. To detect a glitch in the semantics, a first approach

consists in running tests and verifying that the CompCert C interpreter computes the expected value. With this respect, the CompCert C semantics successfully run hundreds of random test programs generated by CSmith [23]. Another indirect but original approach consists in relating formally different semantics for the same language. For instance, when designing the CompCert C semantics, several equivalence between alternate semantics were proved to validate this semantics [4]. Our memory model is a new and interesting opportunity to apply this methodology and perform a cross-validation of the C operators which are the building blocks of the semantics.

Our memory model aims at giving a semantics to more operations (e.g. low-level pointer operations). However, when for instance a C binary operation $v_1\ op\ v_2$ evaluates to a defined value v', using the existing memory model, the symbolic expression $v_1\ op\ v_2$ should normalise to the same value v'. For pointer values, this property is given in Fig. 3, where `sem_binary_operation_expr` is our extension to symbolic expressions of the function `sem_binary_operation`.

During the proof, we have uncovered several issues. The first issue is that our normalisation only returns a location within the bounds of the block. This is not the case for CompCert C that allows, for instance, to increment a pointer with an arbitrary offset. If the resulting offset is outside the bounds, our normalisation returns `undef`. For us, this is a natural requirement because we only apply normalisation when valid pointers are needed (i.e. before a memory read or write). To cope with this discrepancy, we add in lemma `expr_binop_ok_ptr` the precondition `valid_pointer(m, b, o)`. Another issue was a mistake in our syntactic construction of symbolic expressions: a particular cast operator was mapped to the wrong syntactic constructor. After the easy fix, we found two interesting semantics discrepancies with the current semantics of CompCert C.

One issue is related to *weakly valid* pointers [16] which model a subtlety of C stating that *pointers one past the end of an array object* can be compared. As a result, in CompCert C, if (b, o) is a valid location, then $(b, o)<(b, o+1)$ always returns true. In our model, if $(b, o+1)$ wraps around (because of an integer overflow) it may return 0 and therefore the property does not hold. To avoid this corner situation, we state that a valid address of our model excludes `Int.max_unsigned` (see Fig. 2). This is sufficient to prevent the offset from wrapping around and to be compatible with the semantics of CompCert C.

```
Lemma expr_binop_ok_ptr: ∀ op v1 t1 v2 t2 m b o,
 sem_binary_operation op v1 t1 v2 t2 m = ⌊ptr(b,o)⌋ → valid_ptr(m, b, o) →
 ∃ v', sem_binary_operation_expr op v1 t1 v2 t2 m = ⌊v'⌋ ∧
        normalise m v' Ptr = ptr(b, o).
```

Fig. 3: Example of cross-validation of binary C operators.

The last issue is related to the comparison with the `null` pointer. In CompCert, this is the only pointer which is not represented by a location (b, i) but by the integer 0. The semantics therefore assumes that a genuine location can never

```
int main(){ int i=0, *p = &i;
            for(i=0; i < INT_MAX; i++)  if (p++ == 0) return 1;
            return 0; }
```

Fig. 4: A `null` pointer comparison glitch

be equal to the `null` pointer. In our semantics, a location (b, i) can evaluate to 0 in case of wrap around. This is a glitch in the CompCert semantics that is illustrated by the code snippet of Fig. 4. This program initialises a pointer p to the address of the variable `i`. In the loop, `p` is incremented until it equals 0 in which case the loop exits and the program returns 1. With this program, the executable semantics of CompCert C returns 0 because p==0 is always false whatever the value of `p`. However, when running the compiled program, the pointer is a mere integer, the integer eventually overflows; wraps around and becomes 0. Hence, the test holds and the program returns 1. We might wonder how the CompCert semantic preservation can hold in the presence of such a contradiction. Actually, the pointers are kept logical all the way through to the assembly level, and the comparison with the `null` pointer is treated the same during all the compilation process, thus even the assembly program in CompCert returns 0. The inconsistency only appears when the assembly program is compiled into binary and run on a physical machine.

The fix consists in defining the semantics of the comparison with the `null` pointer only if the pointer is *weakly valid*. This causes the program to have undefined semantics at the C level as soon as we increment the pointer beyond its bounds. The issue was reported and the fix was incorporated in the trunk release of CompCert. After adjusting both memory models, we are able to prove that both semantics agree when the existing CompCert C semantics is defined thus cross-validating the semantics of operators.

5 Redesign of Memory Injections

Memory injections are instrumental for proving the correctness of several compiler passes of CompCert. A memory injection defines a mapping between memories; it is a versatile tool to explain how passes reorganise the memory (e.g. construct an activation record from local variables). This section explains how to generalise this concept for symbolic expressions. It requires a careful handling of undefined values `undef`(l) which are absent from the existing memory model.

5.1 Memory Injections in CompCert

In CompCert, a memory injection is a relation between two memories m_1 and m_2 parameterised by an injection function `f: block` $\rightarrow$ `option location` mapping blocks in m_1 to locations in m_2. The injection relation is defined over values (and called `val_inject`) and then lifted to memories (and called `inject`). The `val_inject` relation distinguishes three cases:

1. For concrete values (i.e. integers or floating-point numbers), the relation is reflexive: e.g. `int(i)` is in relation with `int(i)`;
2. `ptr`(b, i) is in relation with `ptr`$(b', i + \delta)$ when `f`$(b) = \lfloor (b', \delta) \rfloor$;
3. `undef` is in relation with any value (including `undef`).

The purpose of the injection is twofold: it establishes a relation between pointers using the function `f` but it can also specialise `undef` by a defined value.

In CompCert, so-called generic memory injections state that every valid location in memory m_1 is mapped by function `f` into a valid location in memory m_2; the corresponding location in m_2 must be properly aligned with respect to the size of the block; and the values stored at corresponding locations must be in injection. Among other conditions, we have that if several blocks in m_1 are mapped to the same block in m_2, the mapping ensures the absence of overlapping.

5.2 Memory Injection with Symbolic Expressions

The injection of symbolic expressions demands a generalisation because `undef` is now parameterised by a location `l`. The function `f` is still present and serves the same purpose. However, the injection must also be applied to undefined values. Moreover, our generalised injection requires an explicit specialisation function `spe`: `location` → `option byte`. Our injection `expr_inject` is therefore defined as the composition of the function `apply_spe spe` which specialises `undef(l)` into concrete bytes, and the function `apply_inj f` which injects locations. Both `spe` and `f` are partial functions. If `spe(l)`=$\emptyset$, the undefined location is not specialised. If `f(b)`=$\emptyset$ and `b` appears in the expression, it cannot be injected.

```
Definition expr_inject spe f e1 e2 := apply_inj f (apply_spe spe e1) = ⌊e2 ⌋.
```

Example 1. Consider the injection `f` and the specialisation functions `spe` and `spe'` defined by: `f`$(b_1) = \lfloor (b_0, 1) \rfloor$, `spe`$(b_1, 0) = \lfloor 0 \rfloor$ and `spe'`$(b_1, 0) = \lfloor 1 \rfloor$. The `val_inject` relation (left column) between values becomes in our memory model the following `expr_inject` relation (right column).

```
val_inject f undef undef          | expr_inject spe  f undef(b1,1) undef(b0,2)
val_inject f undef int(0)         | expr_inject spe  f undef(b1,0) int(0)
val_inject f undef int(1)         | expr_inject spe' f undef(b1,0) int(1)
val_inject f ptr(b1,1) ptr(b0,2)  | expr_inject spe  f ptr(b1,1)   ptr(b0,2)
```

Injections of Memories. Like in the existing CompCert, the injection of values is then lifted to memories. With our memory model, the properties of injections need to be adapted to accommodate for symbolic expressions.

Alignment constraints are modelled in the existing CompCert as a property of offsets. Roughly speaking, a value of size s bytes can be stored at a location (b, o) such that the offset o is a multiple of s. For instance, an integer `int(i)` could be stored at offsets 0, 4, 8, and so on. This model makes the implicit assumption

```
Record inject spe f m₁ m₂ : Prop :=  { ...
 mi_align: ∀ b b' z,  f b = ⌊(b',  z)⌋ →
         alignt(m₁, b) ≤ alignt(m₂, b')  ∧ 2^[alignt(m₁,b)] | z;
 mi_size_mem:   size_mem m₂ ≤ size_mem m₁; }
```

Fig. 5: Memory injection: extra constraints

that memory blocks are always sufficiently aligned. In our model, blocks are given an explicit alignment. As a result, we can precisely state that an injection preserves alignement and is given by the `mi_align` property of Fig. 5. Note that the weaker formulation $2^{\texttt{alignt}(\texttt{m}_1,\texttt{b})} | 2^{\texttt{alignt}(\texttt{m}_2,\texttt{b}')} + z$ is sound. However, the chosen formulation has the advantage of being backward compatible with the existing properties of offsets in CompCert.

The size constraint is evaluated using the `size_mem` function that is the algorithm used by the allocation function (see Section 3.2). This constraint ensures that an injection is compatible with allocation as stated by the following lemma. The hypothesis $\texttt{size\_mem m}_2 \leq \texttt{size\_mem m}_1$ (called `mi_size_mem` in Fig. 5) ensures that the block $\texttt{b}_2$ can be allocated in memory $\texttt{m}_2$.

```
Theorem alloc_parallel_inject: ∀ spe f m₁ m₂ lo hi m₁' b₁,
 0 ≤ lo ≤ hi → inject spe f m₁ m₂ → alloc m₁ lo hi = ⌊(m₁', b₁)⌋ →
 ∃ m₂', ∃ b₂, alloc m₂ lo hi=⌊(m₂',b₂)⌋ ∧ inject spe f[b₁ ↦ ⌊(b₂,0)⌋] m₁' m₂'.
```

Absence of offset overflows. The existing formalisation of `inject` has a property `mi_representable` which states that the offset $o+\delta$ obtained after injection does not overflow. With our concrete memory model, this property is not necessary anymore as it can be proved for any injection.

5.3 Memory Injection and Normalisation

Our normalisation is defined w.r.t. all the concrete memories compatible with the CompCert block-based memory (see Section 2.3). Theorem `norm_inject` shows that under the condition that all blocks are injected, if e and e' are in injection, then their normalisations are in injection too. Thus, the normalisation can only get more defined after injection. This is expected as the injection can merge blocks and therefore makes pointer arithmetic more defined. The condition that all blocks need to be injected is necessary. Without it, there could exist a concrete memory `cm'` in `m'` without counterpart in `m`. The normalisation could therefore fail when the expression would evaluate differently in `cm'`. A consequence of this theorem is that the compiler is not allowed to reduce the memory usage.

```
Theorem norm_inject:∀ spe f m m' e e' τ,
 all_blocks_injected f m → inject spe f m m' → expr_inject spe f e e' →
 val_inject f (normalise m e τ) (normalise m' e' τ).
```

6 Proving the Front-end of the CompCert Compiler

The architecture of the front-end of CompCert is given in Fig. 6. The front-end compiles CompCert C programs down to Cminor programs. Later compiler passes are architecture dependent and are therefore part of the back-end. This section explains how to adapt the semantics preservation proofs of the front-end to our memory model with symbolic expressions.

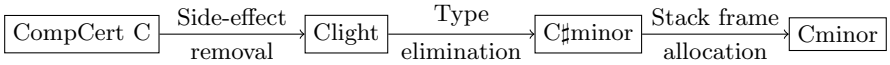


Fig. 6: Architecture of CompCert's front-end

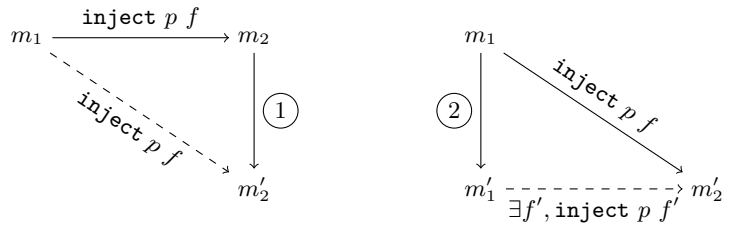
6.1 CompCert Front-end with Symbolic Expressions

The semantics of all intermediate languages need to be modified in order to account for symbolic expressions. In principle, the transformation consists in replacing values by symbolic expressions *everywhere* and introducing the normalisation function when accessing memory. In reality, the transformation is more subtle because, for instance, certain intermediate semantic functions explicitly require locations represented as pairs (b, o). In such situations, a naive solution consists in introducing a normalisation. This solution proves wrong and breaks semantics preservation proofs because introduced normalisations may be absent in subsequent intermediate languages. The right approach consists in delaying normalisation as much as possible. Normalisations are therefore introduced before memory accesses. They are also introduced when evaluating the condition of `if` statements and to model the lazy evaluation of && and || operators. Using this strategy we have adapted the semantics (with built-in functions as only external functions) of the 4 languages of the front-end.

In our experience, the difficulty of the original semantics preservation proofs is not correlated with the difficulty of adapting the proofs to our memory with symbolic expressions. For instance, the compilation pass from CompCert C to Clight is arguably the most complex pass to prove; the proof is almost identical with symbolic expressions. In the following, we focus on the two other passes which stress different features of our memory model.

6.2 From Clight to C♯minor

The compilation from Clight to C♯minor translates loops and `switch` statements into simpler control structures. This pass does not transform the memory and therefore the existing proof can be reused. The pass also performs type-directed transformations and removes redundant casts. For example, it translates the expression `p + 1` with `p` of type `int *` into the expression `p + sizeof(int)`. For the existing memory model, both expressions compute exactly the same value. However, with symbolic expressions, syntactic equality is a too strong requirement that needs to be relaxed to a weaker equivalence relation. A natural candidate



1. allocation of stack frame
2. allocation of local variables

Fig. 7: Structure of `match_callstack_alloc_variables` 's proof in CompCert

is the equality of the normalisation. However, this relation is too weak and fails to pass the induction step. Indeed, when expressions e_1 and e_2 have the same normalisation ($v_1 \equiv v_2$), it is not the case that $op_1^\tau(v) \equiv op_1^\tau(v')$ when the normalisations are `undef`. A stronger relation is the equality of the evaluation of symbolic expressions in any concrete memory (compatible or not).

We lift the equivalence relation to memories in the obvious way. To carry out the proof, we also extend the interface of the memory model and prove that the memory operations are morphisms for the equivalence relation. With these modifications, the compiler pass can be proved semantics preserving using the existing proof structure.

6.3 From C♯minor to Cminor

The compilation from C♯minor to Cminor allocates the stack frame, thus transforming significantly the memory. The stack frame is a single block and local variables are accessed via offsets in this block. The proof introduces a memory injection stating how the blocks representing local variables in C♯minor are mapped into the single block representing the stack frame in Cminor.

The existing proof can be adapted with our generalised notion of injection (see Section 5) with the notable exceptions of two intermediate lemmas whose proofs need to be completely re-engineered. The problem is related with the preservation of the memory injection when allocating and de-allocating the variables in C♯minor and the stack frame in Cminor. The structure of the original proof is depicted in Fig. 7 where plain arrows represent hypotheses and the dotted arrow the conclusion. The existing proof first allocates the stack frame in memory m_2 to obtain the memory m'_2. It then establishes that the existing injection between the intial memories m_1 and m_2 still holds with the memory m'_2. In a second step, the memory m'_1 is obtained by allocating variables in memory m_1 and the proof constructs an injection thus concluding the proof.

With our memory model, memory injections need to reduce the memory usage – this is needed to ensure that allocations cannot fail. Here, this is obviously not the case because the memory m'_2 contains a stack frame whereas the corre-

Fig. 8: Variables on the left ; stack frame on the right.

sponding variables are not yet allocated in m_1. Our modified proof is directly by induction over the number of allocated variables. In this case, we prove that if the variables do fit into memory, then so does the stack frame. Note that to accommodate for alignment and padding the stack frame might allocate more bytes than the variables. However, our allocation algorithm makes a worst-case assumption about alignment and padding and therefore ensures that there is enough room for allocating the stack frame. We therefore conclude that the memories m_1' and m_2' are in injection.

At function exit, the variables and the stack frame are freed from memory. As before, the arguments of the original proof do not hold with our memory model. Once again, we adapt the two-step proof with a direct induction over the number of variables. To carry out this proof and establish an injection we have to reason about the relative sizes of the memories. We already discussed how the allocation algorithm rules out the possibility for the stack frame not to fit into the memory. Here, we have to deal with the opposite situation where the stack frame could use less memory than the variables.

To avoid this situation, and facilitate the proof, our stack frame currently allocates all the padding introduced when allocating the variables. This pessimistic construction is depicted in Fig. 8 where thick lines identify block boundaries and padding is identified by grey rectangles. It shows our injection of two 8 bits character variables and a 32 bit integer variable into a stack frame.

We are currently investigating on how to give a finer account of the necessary padding to avoid allocating too much memory for the stack frame. Yet, using the strategy described above, we are able to complete the proof of the front-end while reusing as much as possible the architecture of the existing proof.

7 Related Work

Examples of low-level memory models include Norrish's HOL semantics for C [20] and the work of Tuch *et al.* [21]. There, memory is essentially a mapping from addresses to bytes, and memory operations are axiomatised in these terms. Reasoning about program transformations is more difficult than with a block-based model; Tuch *et al.* use separation logic to alleviate these difficulties.

Memory models have been proposed to ease the reasoning about low-level code. VCC [7] generates verification conditions using an abstract typed memory model [8] where the memory is a mapping from typed pointers to structured C values. This memory model is not formally verified. Using Isabelle/HOL, Autocorres [10,11] constructs provably correct abstractions of C programs. The memory models of VCC [8] and Autocorres [11] ensure separation properties of

pointers for high-level code and are complete w.r.t. the concrete memory model. For the CompCert model [18], separation properties of pointers are for free because pointers are modelled as abstract locations. For our symbolic extension, the completeness (and correctness) of the normalisation is defined w.r.t. a concrete memory model and therefore allows for reasoning about low-level idioms.

Several formal semantics of C are defined over a block-based memory model (e.g. [9,15,17]). The different models differ upon their precise interpretation of the ISO C standard. The CompCert C semantics [5] provides the specification for the correctness of the CompCert compiler [17]. CompCert is used to compile safety critical embedded systems [2] and the semantics departs from the ISO C standard to capture existing practices. Our semantics extends the existing CompCert semantics and benefits from its infrastructure.

Krebbers *et al.* also extend the CompCert semantics but aim at being as close as possible to the C standard [16]; he formalises sequence points in non-deterministic programs [15] and strict aliasing restrictions in `union` types of C11 [14]. This is orthogonal to the focus of our semantics which gives a meaning to implementation defined low-level pointer arithmetic and models bit-fields. Most recently, Kang *et al.* [13] propose a formal memory model for a C-like language which allows optimisations in the presence of integer-pointer casts. Pointers are kept logical until they are cast to integers, then a concrete address is non-deterministically assigned to the block of the pointer. Their semantics of C features non-determinism while determinism is a crucial feature of our model.

8 Conclusion

This work is a milestone towards a CompCert compiler proved correct with respect to a more concrete memory model. Our formal development adds about 10000 lines of Coq to the existing CompCert memory model. A side-product of our work is that we have uncovered and fixed a problem in the existing semantics of the comparison with the `null` pointer. We are very confident that this is the very last remaining bug that can be found at this semantics level. We also prove that the front-end of CompCert can be adapted to our refined memory model. The proof effort is non-negligible: the proof script for our new memory model is twice as big as the existing proof script. The modifications of the front-end are less invasive because the proof of compiler passes heavily rely on the interface of the memory model.

As future work, we shall study how to adapt the back-end of CompCert. We are confident that program optimisations based on static analyses will not be problematic. We expect the transformations to still be sound with the caveat that static analyses might require minor adjustments to accommodate for our more defined semantics. A remaining challenge is register allocation which may allocate additional memory during the spilling phase. An approach to solve this issue is to use the extra-memory that is available due to our pessimistic construction of stack frames. Withstanding the remaining difficulties, we believe that the

full CompCert compiler can be ported to our novel memory model. This would improve further the confidence in the generated code.

References

1. Companion website. URL: http://www.irisa.fr/celtique/ext/new-mem.
2. R. Bedin França, S. Blazy, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS2*, 2012.
3. F. Besson, S. Blazy, and P. Wilke. A precise and abstract memory model for C using symbolic values. In *APLAS*, volume 8858 of *LNCS*, 2014.
4. S. Blazy. Experiments in validating formal semantics for C. In *C/C++ Verification Workshop*. Raboud University Nijmegen report ICIS-R07015, 2007.
5. S. Blazy and X. Leroy. Mechanized Semantics for the Clight Subset of the C Language. *J. Automated Reasoning*, 43(3), 2009.
6. A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: designing scalable software for multicore processors. In *SOSP*. ACM, 2013.
7. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, and al. VCC: A Practical System for Verifying Concurrent C. In *TPHOLs*, volume 5674 of *LNCS*. Springer, 2009.
8. E. Cohen, M. Moskal, S. Tobies, and W. Schulte. A Precise Yet Efficient Memory Model For C. *ENTCS*, 254, 2009.
9. C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *POPL*. ACM, 2012.
10. D. Greenaway, J. Andronick, and G. Klein. Bridging the Gap: Automatic Verified Abstraction of C. In *ITP*, volume 7406 of *LNCS*. Springer, 2012.
11. D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: formal verification of C code without the pain. In *PLDI*. ACM, 2014.
12. J. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified C static analyzer. In *POPL*, 2015.
13. J. Kang, C.-K. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis. A formal C memory model supporting integer-pointer casts. In *PLDI*. ACM, 2015.
14. R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *CPP*, volume 8307 of *LNCS*. Springer, 2013.
15. R. Krebbers. An operational and axiomatic semantics for non-determinism and sequence points in C. In *POPL*. ACM, 2014.
16. R. Krebbers, X. Leroy, and F. Wiedijk. Formal C semantics: Compcert and the C standard. In *ITP 2014*, volume 8558 of *LNCS*. Springer, 2014.
17. X. Leroy. Formal verification of a realistic compiler. *C. ACM*, 52(7), 2009.
18. X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert memory model. In *Program Logics for Certified Compilers*. Cambridge University Press, 2014.
19. X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Automated Reasoning*, 41(1), 2008.
20. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.
21. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *POPL*. ACM, 2007.
22. X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. Kaashoek. Undefined behavior: What happened to my code? In *APSYS '12*, 2012.
23. X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*. ACM, 2011.