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

CompCertS: A Memory-Aware Verified C Compiler using a Pointer as Integer Semantics

Frédéric Besson · Sandrine Blazy · Pierre Wilke

Received: date / Accepted: date

Abstract The CompCert C compiler provides the formal guarantee that the observable behaviour of the compiled code improves on the observable behaviour of the source code. In this paper, we present a formally verified C compiler, CompCertS, which is essentially the CompCert compiler, albeit with a stronger formal guarantee: it gives a semantics to more programs and ensures that the memory consumption is preserved by the compiler. CompCertS is based on an enhanced memory model where, unlike CompCert but like Gcc, the binary representation of pointers can be manipulated much like integers and where, unlike CompCert, allocation may fail if no memory is available.

The whole proof of CompCertS is a significant proof-effort and we highlight the crux of the novel proofs of 12 passes of the back-end and a challenging proof of an essential optimising pass of the front-end.

Keywords Verified Compilation, Low-level Code, Optimisations, Pointer as Integer

1 Introduction

Over the past decade, the CompCert compiler has established a milestone in compiler verification. CompCert is a formally verified C compiler written with the Coq proof assistant, which initially targeted safety-critical embedded software.

Frédéric Besson
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
E-mail: frederic.besson@inria.fr

Sandrine Blazy
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
E-mail: sandrine.blazy@irisa.fr

Pierre Wilke
Yale University, New Haven, USA
E-mail: pierre.wilke@yale.edu

The compiler comes with a machine-checked proof that it does not introduce bugs during compilation [2]. This semantic preservation proof relies on the formal semantics of the source and target languages of the compiler, and requires that the source program has a defined semantics. Therefore, CompCert only provides formal guarantees for programs that do not exhibit undefined behaviours – a property that is in general undecidable.

CompCert's memory model is a central component of the compiler. In this paper, we show how to adapt CompCert for a more expressive memory model which lifts two main limitations. First, memory allocation in CompCert always succeeds, therefore modelling infinite memory. As a consequence, the compiler does not guarantee anything on the memory consumption of the compiled program. In particular, the compiled program may exhibit a stack overflow. Second, CompCert's memory model limits pointer arithmetic: implementation-defined operations on pointers such as arbitrary comparison or bitwise operations result in an undefined behaviour of the memory model. This may seem restrictive but this is compliant with the C standard.

In previous work [3], we proposed a more concrete memory model inspired by CompCert where memory is finite and pointers can be used as integers. On that basis, we have adapted the proof of 3 passes of CompCert's front-end [4]. In this work, we present a fully verified CompCert compiler where 12 remaining passes have been ported to our new memory model. This compiler is called CompCertS (for CompCert with Symbolic values). CompCertS gives much stronger guarantees about the behaviour of arbitrary pointer arithmetic, thus avoiding the miscompilation of programs performing bit-level manipulation of pointers.

CompCertS also provides strong guarantees about the relative memory usage of the source and target programs. This is challenging because it is unclear how to even define the memory usage at the C level. We tackle this challenge by first defining the memory usage of individual functions directly from the C level, and then proving that compiled programs use no more memory than source programs. In particular, this ensures that the absence of memory overflow is preserved by compilation.

All the results presented in this paper have been mechanically verified using the Coq proof assistant. The development is available online [1]. Additionally, we include links to the online documentation for several definitions and theorems in this paper under the form of Coq logos 🐓.

Our contribution is CompCertS, which is stronger than CompCert in the following sense: *1)* CompCertS offers guarantees for a wider class of programs; *2)* CompCertS also offers guarantees about the memory usage of the compiled program. More precisely, we make the following technical contributions:

- We present the proof of the compiler back-end (12 compiler passes) including constant propagation, common sub-expression elimination and dead-code elimination. In particular, we detail how the existing alias analyses of CompCert [19] benefit from our more defined semantics.
- We show how to instrument the C semantics with oracles specifying the memory usage of functions, so that the compiler only reduces the memory usage of the program. We thus ensure that the absence of memory overflow is preserved by compilation.

The rest of the paper is organised as follows. First, Section 2 gives background information on CompCert and the symbolic memory model of our previous work [4]. Section 3 gives an overview of the proof effort required to port the majority of the compiler, and of the proof challenges related to treating pointers as integers. Section 4 describes how we deal with an early pass of CompCert which relies on a subtle memory injection. Section 5 explains the impact of the symbolic memory model on optimisations. Section 6 shows how we ensure that the compiler reduces the memory usage of programs and proves that the absence of memory overflows is preserved. Section 7 mentions related work and finally, Section 8 concludes.

2 Background on CompCert

This section describes the architecture of the CompCert compiler [14]. It also summarises the main features and properties of our memory model [3,4]. Our work is based on version 2.4 of CompCert.

2.1 Architecture of the CompCert Compiler

CompCert compiles C programs into assembly code, through 8 other intermediate languages. The same memory model is shared by all the languages of the compiler. Each language is given a formal semantics in the form of a state transition system. The semantics observe behaviours that are either defined behaviours, with a trace of I/O events (this trace is finite for terminating programs, or infinite for diverging programs), or undefined behaviours.

Every transformation from one language to another is proved to be semantics preserving using simulation relations, relating the states of the source and target programs with some matching relation. In particular, the trace of I/O events that they emit must be the same. The proof technique most commonly used in CompCert is forward simulations, where every step in the source language is matched with a number of steps in the target language. The heart of a forward simulation proof is captured by Theorem 1.

Theorem 1 (Forward Simulation) *Given a source program and a target program represented by their state transition systems $\to_S$ and $\to_T$, there is a forward simulation between those programs through the simulation relation $\sim$ if and only if for any states S_1 and S_2 related by $\sim$, any step taken from S_1 can be simulated by a (sequence of) step(s) from S_2 such that the resulting states are still related by $\sim$. Mathematically,*

$$\forall\ S_1 \sim S_2, \forall\ S_1', S_1 \xrightarrow{e}_S S_1' \Rightarrow \exists\ S_2', S_2 \xrightarrow{e}{}^{+}_T S_2' \land S_1' \sim S_2',$$

where e is the trace of emitted I/O events.

The final compiler correctness theorem is about behaviour preservation. Behaviours are built on (possibly infinite) traces of events, in the following way, where t are finite traces and τ is an infinite trace:

$$\mathit{beh} \triangleq \texttt{Terminates}(t) \mid \texttt{Diverges}(t) \mid \texttt{Reacts}(\tau) \mid \texttt{Wrong}(t).$$

The behaviour $\texttt{Terminates}(t)$ corresponds to an execution that terminates normally after emitting the trace of events t. The behaviour $\texttt{Diverges}(t)$ is the execution of a program emitting t, and then loops silently (i.e. without emitting events)

$$\begin{array}{rl} val \ni \ \ v & := \texttt{int}(i) \mid \texttt{ptr}(b,o) \mid \texttt{undef} \\ \texttt{memval} \ni mv & := \texttt{Byte}(x) \mid \texttt{Pointer}(b,o,n) \mid \texttt{Undef} \end{array}$$

Fig. 1: Run-time and memory values

forever. $\texttt{Reacts}(\tau)$ is the behaviour of an execution that never terminates but still emits messages, resulting in the infinite trace τ. Finally, the behaviour $\texttt{Wrong}(t)$ corresponds to a program that goes wrong (i.e. triggers undefined behaviour) after having emitted the finite trace t.

Given some hypotheses about the determinism of the target language, we can transform the forward simulation proofs into a behaviour preservation theorem stating that every behaviour of the compiled program is a behaviour of the source program, i.e. the compiler has not introduced *bugs*.

The composition of the simulation lemmas for all the compiler passes forms the compiler semantic preservation theorem given below.

Theorem 2 (CompCert's semantic preservation) *Suppose that tp is the result of the successful compilation of the program p. If bh' is a behaviour of tp then there exists a behaviour bh such that bh is a behaviour of p and bh' improves on the behaviour bh.*

$$bh' \in ASem(tp) \Rightarrow \exists bh. bh \in CSem(p) \wedge bh \subseteq bh'$$

In the theorem, *CSem* gives the semantics of C programs and *ASem* gives the semantics of assembly programs. Moreover, a behaviour bh' improves on a behaviour bh (written $bh \subseteq bh'$) if either bh and bh' are the same, or undefined behaviours in bh are replaced by defined behaviours in bh'.

2.2 The Memory Model of CompCert

The memory model of CompCert is the cornerstone of the semantics of all the intermediate languages. It consists of a collection of separated *blocks*, where blocks are arrays of a given size. A value $v \in val$ (see Fig. 1) can be either a 32-bit integer $\texttt{int}(i)$, a pointer or the token `undef`. A pointer is a pair $\texttt{ptr}(b,o)$ consisting of a block identifier b and an offset o. CompCert also features 64-bit integers, single and double precision floating-point numbers, which we ignore in this paper for the sake of simplicity. To allow fine-grained access to the memory, CompCert does not store values directly in the memory. Rather, values are encoded as sequences of byte-sized *memory values* called `memval` that describe the content of a memory block. They are either concrete 8-bit integers `Byte` (x), a special `Undef` byte that represents uninitialised memory, or a byte-sized fragment of a symbolic pointer value `Pointer` (b,o,n) (read: n-th byte of pointer $\texttt{ptr}(b,o)$). Therefore, a pointer $\texttt{ptr}(b,o)$ is encoded in memory as a sequence of 4 `memval`s, from $\texttt{Pointer}(b,o,0)$ to $\texttt{Pointer}(b,o,3)$. (The version of CompCert that this works build upon, v2.4, only supports 32-bit pointers, hence 4 `memval`s. More recent versions support 64-bit pointers, made of 8 `memval`s.) The memory model exports four main operations: `load` reads values from the memory at a given address (a block and an offset), `store` writes values into the memory at a given address, `alloc` allocates a new block and `free` frees a given block.

```
struct rb_node {
  uintptr_t rb_parent_color;
  struct rb_node *rb_right;
  struct rb_node *rb_left;  };
#define rb_color(rb) (((rb)-> rb_parent_color) & 1)
#define rb_parent(r) \
    ((struct rb_node *) ((r)-> rb_parent_color & ~3))
```

Fig. 2: Red-black tree implementation in Linux

$$\begin{array}{rl} \mathit{sval} \ni \mathit{sv} & := \mathit{val} \mid \mathbf{unop}(\mathit{uop}, \mathit{sv}) \mid \mathbf{binop}(\mathit{bop}, \mathit{sv}_1, \mathit{sv}_2) \\ \texttt{smemval} \ni \mathit{smv} & := \texttt{Symbolic}(\mathit{sv}, n) \end{array}$$

Fig. 3: Symbolic run-time and memory values

2.3 A Symbolic Memory Model for CompCert

In previous work [3,4], we extended CompCert's memory model and gave semantics to pointer operations by replacing the value domain *val* by a more expressive domain *sval* of symbolic values. This low-level memory model enables reasoning about the bit-level encoding of pointers within CompCert. In this section, we first give a motivating example; then we recall the principles of symbolic values and their normalisation.

2.3.1 Motivation for Pointers as Integers.

Fig. 2 shows an example of C code that benefits from our low-level memory model. This is an implementation of red-black trees which belongs to the Linux kernel. A node in a red-black tree (type rb_node) contains an integer rb_parent_color and two pointers to its children nodes. The integer rb_parent_color encodes both the color of the node and a pointer to the parent node. The rationale for this encoding is as follows: 1) pointers to rb_nodes are at least 4-byte aligned, therefore the two trailing bits are zeros; and 2) the color of a node can be encoded with a single bit. Retrieving each piece of information from this encoding is implemented by the two macros rb_color and rb_parent shown in Fig. 2. To get the parent pointer, the macro clears the two trailing bits using a bitwise & with ∼3 (i.e. $0b1\dots100$). In CompCert, these operations are undefined because of the bitwise operations on pointers. In CompCertS, these operations are defined and therefore this kernel code can be safely compiled without fear of any miscompilation.

2.3.2 Symbolic Values.

A symbolic value $sv \in sval$ (see Fig. 3) is either a value v or an expression built from unary and binary C operators over symbolic values. Memory values `memval` are also generalised into symbolic memory values `smemval`, which have a single constructor $\texttt{Symbolic}(sv, n)$, denoting the n-th byte of a symbolic value sv. This constructor is inspired from the `Pointer` $(\cdot,\cdot,\cdot)$ constructor of CompCert (see Fig. 1) and subsumes the three existing cases.

Building symbolic values instead of the token `undef` for undefined operations delays the challenge of giving more semantics to C expressions. However, symbolic values cannot be kept symbolic indefinitely. To perform memory accesses at an address represented by the symbolic value *addr*, the address *addr* must be *normalised* into a genuine pointer $\texttt{ptr}(b, o)$. Similarly, the condition *cond* of a conditional statement must be normalised into an integer $\texttt{int}(i)$ to decide which branch to follow. The normalisation is specified as a function `normalise` which takes as input a memory state m and a symbolic value sv, and outputs a value v. Its specification relies on the notions of concrete memories valid for a memory state m, and of evaluation of expressions that we recall below.

An intuitive way to think about symbolic values is in terms of intermediate values that do not make sense immediately, but can be soundly used to later produce regular values, just like how complex numbers were first introduced in mathematics as intermediate values to solve cubic equations.

2.3.3 Concrete Memories and Evaluation.

A concrete memory is a mapping from blocks to concrete addresses, represented as 32-bit integers. In addition to the permissions and memory contents associated to blocks in CompCert, we also associate with each memory block b a size *size* and an alignment constraint *al*. We say a pointer $\texttt{ptr}(b, o)$ is *valid* if the offset o is within the bounds $[0, size[$, written $\texttt{valid}(m, b, o)$. The size and alignment of a block b can be retrieved with the accessors $\texttt{size}(m, b)$ and $\texttt{align}(m, b)$.

Definition 1 A concrete memory cm is valid for a memory state m ($cm \vdash m$) if the following conditions hold:

1. Valid addresses lie within the address space, i.e. $\forall\ b\ o,\ \texttt{valid}(m, b, o) \Rightarrow cm(b) + o \in]0; 2^{32} - 1[$.
2. Valid pointers from distinct blocks do not overlap, i.e. $\forall\ b\ b'\ o\ o',\ b \neq b' \ \wedge\ \texttt{valid}(m, b, o) \wedge \texttt{valid}(m, b', o') \Rightarrow cm(b) + o \neq cm(b') + o'$.
3. Addresses are properly aligned, i.e. $\forall\ b,\ 2^{\texttt{align}(m,b)} \mid cm(b)$.

We exclude the address 0 from valid addresses because it represents the NULL pointer and is therefore invalid. We also exclude the address $2^{32} - 1$ so that *weakly-valid* pointers, i.e. pointers *one past the end* of an object, are also valid. (See the C standard [10], section 6.5.8.5 (Relational operators) for a discussion of pointers *one past the end.*)

The evaluation of a symbolic value sv in a concrete memory cm (written $[\![sv]\!]_{cm}$) consists in replacing pointers with their integer value (according to cm) and then evaluating the resulting expression with standard integer operations.

Example 1 Consider for example a concrete memory cm that maps a block b to the address 32. The evaluation of the symbolic value $sv = \texttt{ptr}(b, 5)\ \&\ \texttt{int}(\texttt{1})$ results in $\texttt{int}(1)$ because $[\![sv]\!]_{cm} = (cm(b) + 5)\ \&\ 1 = (32 + 5)\ \&\ 1 = 37\ \&\ 1 = 1$.

2.3.4 Specification of the Normalisation.

Rather than defining an algorithm for the normalisation, we specify its behaviour through a relation $\texttt{is\_norm}\ m\ sv\ v$, where m is a memory state, sv is a symbolic value and v is a value. This predicate is defined as follows.

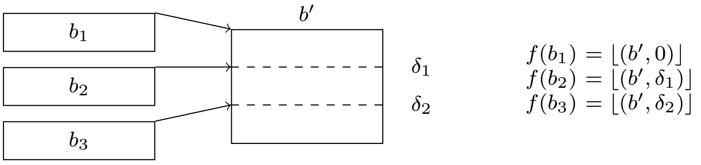

b_1
b_2
b_3
b'
δ_1
δ_2
$f(b_1) = \lfloor(b', 0)\rfloor$
$f(b_2) = \lfloor(b', \delta_1)\rfloor$
$f(b_3) = \lfloor(b', \delta_2)\rfloor$

Fig. 4: Injecting several blocks into one

Definition 2 (Sound normalisation) A value v is a sound normalisation of sv in m, if v and sv evaluate identically in every concrete memory cm valid for m.

$$\texttt{is\_norm}\ m\ sv\ v \triangleq \forall cm \vdash m \Rightarrow [\![sv]\!]_{cm} = [\![v]\!]_{cm}.$$

We prove that this relation is deterministic, i.e. any two values v_1 and v_2 that are sound normalisations of the same symbolic value sv in the same memory state m are necessarily the same. However, this is only true if at least one valid concrete memory exists for any memory state m, otherwise any value v would be a sound normalisation of any symbolic value sv. We enforce this property by restricting the allocation operation of the memory model of CompCert so that it fails if no concrete memory can be constructed. This is explained in great detail in [5]. We will discuss additional aspects related to finite memory in Section 6 in this article. For convenience, in the rest of this article, we will refer to the normalisation as the function `normalise`, which returns a value v that is a sound normalisation when such a value exists, and `undef` otherwise.

Example 2 Consider a program which stores information in the 2 least significant bits of a 4-byte aligned pointer (cf. Fig. 2). The symbolic value after setting the last 2 bits of a pointer $\texttt{ptr}(b, 0)$ is $sv = \texttt{ptr}(b, 0) \mid 3$. To recover the original pointer, the last two bits can be cleared by the following bitwise manipulation: $sv' = sv\ \&\ \sim 3$. We have that sv' normalises into pointer $\texttt{ptr}(b, 0)$ because for any valid concrete memory cm:

$$[\![sv']\!]_{cm} = [\![(\texttt{ptr}(b, 0) \mid 3)\ \&\ \sim 3]\!]_{cm} = (cm(b) \mid 3)\ \&\ \sim 3 = cm(b)$$

The last rewriting step is justified by the alignment constraints of block b. Since $[\![\texttt{ptr}(b, 0)]\!]_{cm} = cm(b)$ for any cm, then sv' normalises into $\texttt{ptr}(b, 0)$.

2.4 Memory Injections

Memory injections are CompCert's central notion to formalise the effect of merging blocks together; they are used to specify the passes that transform the memory layout. The stereotypical example is the construction of stack frames, which happens during the transformation from C♯minor to Cminor. At the C♯minor level, each local variable is allocated in its own block. In Cminor, a single block contains all the local variables, stored at different offsets. This mapping from local variable blocks in C♯minor to offsets in the stack block in Cminor is captured by a memory injection. A memory injection is characterised by an injection function $f : block \to \lfloor block \times \mathbb{Z} \rfloor$ that optionally associates with each block a new block and an offset within that block. For example, in Fig. 4, the blocks b_1, b_2 and b_3 are injected by f into the single block b', at different offsets.

In addition to reflecting the structural relation between memory states, injections also relate the contents of the memory states. Values that are stored at corresponding locations are required to be *in injection.* Two values v_1 and v_2 are in injection if 1) v_1 is `undef`, or 2) v_1 and v_2 are the same non-pointer value, or 3) v_1 is $\texttt{ptr}(b, o)$, v_2 is $\texttt{ptr}(b', o + \delta)$ and $f(b) = \lfloor(b', \delta)\rfloor$[1]. For example, in Fig. 4, the pointer $\texttt{ptr}(b_2, o)$ is in injection with the pointer $\texttt{ptr}(b', o + \delta_1)$.

Two symbolic values are in injection (see [4]) if they have the same structure (the same operators are applied) and the values at the leaves of each symbolic value are in injection. In [4], we proved a central result that relates injections and normalisations, recalled in Theorem 3.

Theorem 3 *For any total injection f, for any memory states m_1 and m_2 in injection by f, for any symbolic values sv_1 and sv_2 in injection by f, the normalisations of sv_1 in m_1 and of sv_2 in m_2 are in injection by f.*

This theorem has the precondition that f must be a *total* injection, i.e. all non-empty blocks must be injected (i.e. $f(b) \neq \emptyset$). In this paper, one of our contributions is a generalisation of Theorem 3, which covers the case of more general injections. As we shall see in Section 4, it is required to prove the SimplLocals pass of CompCert.

3 Overview of the Compiler Proof

This paper addresses the challenge of porting the CompCert compiler to our semantics with symbolic values, where pointer operations behave as integer operations, e.g. bitwise operators are defined on pointers and memory is bounded. Fig. 5 gives an overview of the 19 compiler passes of CompCert, together with the kind of simulation relations that are used to prove them. Three such relations between memory states are defined: memory equalities, memory extensions and memory injections. They share a common basis, the notions of memory embeddings, defined in [15]. Memory equalities are used by passes that do not modify the memory at all, neither its structure nor its contents. Memory extensions are used by passes that do not modify the structure of the memory, but are allowed to specialise the values stored in the memory (e.g. , transform an `undef` value into any other value). Finally, as explained in the previous section, memory injections are used for passes that modify the structure of the memory.

Our changes to the semantics of the individual languages consist mainly in inserting normalisations before memory accesses and conditionals. These changes are reflected in the semantic preservation proofs, where we now have to account for the preservation of normalisations.

The compiler passes that are proved based on the equality simulation relation are the simplest to port. The passes based on memory extensions and memory injections require additional lemmas about the preservation of normalisations with respect to these memory relations, and the passes based on memory injections operate the most difficult memory transformations of the compiler.

In the rest of this paper, we will focus on three particular aspects of our proof effort. First, in Section 4 we address the problems raised by the SimplLocals pass

[1] $\lfloor\cdot\rfloor$ denotes the option type. We write $\lfloor v\rfloor$ for Some(v) and $\emptyset$ for None.

Language / Pass	Simulation relation
Frontend	
C	
Cstrategy	equality
SimplExpr	equality
Clight	
SimplLocals	injection
C♯minorgen	extension
C♯minor	
Cminorgen	injection
Backend	
Cminor	
Selection	extension
CminorSel	
RTLgen	extension

Language / Pass	Simulation relation
RTL	
Tailcall	extension
Inlining	injection
Renumber	equality
Constprop	extension
CSE	extension
Deadcode	extension
Allocation	extension
LTL	
Tunneling	equality
Linearize	equality
Linear	
CleanupLabels	equality
Stacking	injection
Mach	
Asmgen	extension
Assembly	

Fig. 5: Overview of the compiler passes and the simulation relations used

of CompCert, which modifies the structure of the memory, and uses a kind of memory injection that is not covered by our previous work [5]. Then, in Section 5 we explain the challenges related to optimisations, and in particular the notion of pointer provenance. The existing pointer analysis in CompCert needs to be refined, so that it is correct in our symbolic setting. Finally, in Section 6 we describe the implications of having a bounded memory model in CompCert. In particular, we need that every compiler pass reduces the memory usage of programs, and we show how we ensure this is in fact the case in CompCertS.

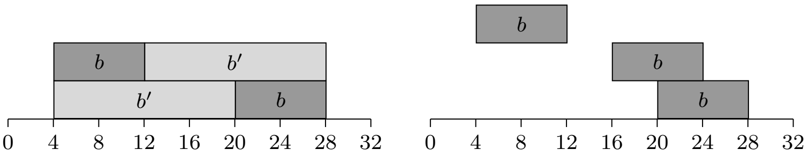
4 Proving the Correctness of SimplLocals

The SimplLocals compiler pass is one of the earliest in CompCert. Its source language is Clight, a stripped-down dialect of C where expressions are side-effect-free. The purpose of this pass is to pull out of memory the local scalar variables that do not need to reside in memory: those whose address is never taken. Those variables are transformed into *temporaries*, i.e. pseudo-registers, upon which most subsequent optimisations operate.

4.1 Arguments for the correctness of SimplLocals .

In CompCert, the correctness of this compiler pass relies on memory injections. The blocks corresponding to variables that are not transformed into temporaries are injected into themselves (i.e. $f(b) = \lfloor b', 0 \rfloor$), while the blocks corresponding to variables that are transformed into temporaries are not injected (i.e. $f(b) = \emptyset$).

The core difficulty of porting the proof of SimplLocals to the symbolic setting resides in proving that normalisations are preserved by injections. In previous work, we have established Theorem 3 which proves this preservation for total injections. Here, the injection is partial (i.e. some blocks are not injected) and



(a) Before injection

(b) After injection

Fig. 6: Concrete memories and partial injections

therefore Theorem 3 does not apply. The following example illustrates the challenge of dealing with partial injections.

Example 3 For the sake of simplicity, consider a memory size of 32 bytes and a memory state m_1 with two blocks b and b' which are both 4-byte aligned: b of size 8 and b' of size 16. We show in Fig. 6a the only two possible concrete memories, where b is the darker block and b' is the lighter one. Note that no block can be assigned the address 0 nor the address 28, as per Definition 1.

Consider the symbolic value $sv = \texttt{ptr}(b, 0)! = 16$. It normalises into 1 in m_1, because b is never allocated at address 16 in any concrete memory valid for m_1. Indeed, this address is always occupied by block b'. Now consider a memory state m_2 where the block b' has been pulled out of memory. Fig. 6b shows that in m_2 it is, of course, still possible to allocate block b at addresses 4 and 20. However, there is a new possible configuration where block b can be allocated at address 16. The normalisation of sv is now undefined because sv evaluates to different values (1 or 0) depending on the concrete memory used. This contradicts Theorem 3, which we are trying to prove.

The essence of the problem illustrated by the above example is that blocks may have more allowed positions after the injection than before, meaning that the set of valid concrete memories is larger after the injection. Therefore, the normalisation may be less defined after a partial injection and Theorem 3 cannot be generalised for arbitrary partial injections.

4.2 Well-behaved injections.

We identify a restricted class of *well-behaved* injections functions f, for which we show that blocks that are injected by f (those for which $f(b) \neq \emptyset$) do not gain new valid concrete addresses after the injection. The criterion for well-behavedness of injection functions f is stated in Definition 3.

Definition 3 (Well-behaved injection) An injection function f is said to be *well-behaved* if the blocks that are forgotten by f are at most 8-byte wide and at most 8-byte aligned. Formally,

$$\texttt{well\_behaved}\ (f, m)\ \triangleq\ \forall\ b,\ f(b) = \emptyset \Rightarrow \texttt{size}(m, b) \leq 8 \wedge \texttt{align}(m, b) \leq 8.$$

The injection used for the correctness proof of SimplLocals satisfies this constraint because only scalar variables may be removed from the memory, i.e. the largest are long-typed variables that are 8-byte wide and 8-byte aligned. Using

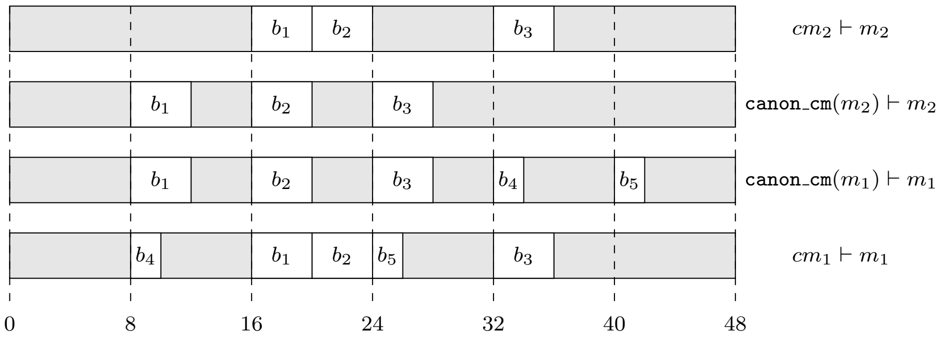


Fig. 7: Inverting partial injections.

such well-behaved injections, we can prove Lemma 1, from which a generalised version of Theorem 3 can be derived, as we explain at the end of this section.

Lemma 1 *Let f be a well-behaved injection function. Let m_1 and m_2 be memory states in injection by f. For every concrete memory cm_2 valid for m_2, there is a corresponding concrete memory cm_1 valid for m_1, such that every non-forgotten block has the same address in cm_1 and cm_2. Formally,*

$$\forall f,\ \texttt{well\_behaved}\ f \Rightarrow$$
$$\forall\ m_1\ m_2,\ \texttt{mem\_inject}\ f\ m_1\ m_2 \Rightarrow \forall\ cm_2 \vdash m_2, \exists\ cm_1 \vdash m_1 \wedge cm_1 \equiv_f cm_2$$
$$\textit{where}\ cm_1 \equiv_f cm_2 \triangleq \forall\ b\ b', f(b) = \lfloor (b', 0) \rfloor \Rightarrow cm_1(b) = cm_2(b')$$

The problem that Lemma 1 solves can be thought of as follows: for every concrete memory cm_2 valid for m_2 ($cm_2 \vdash m_2$), it is possible to insert back all the blocks that have been forgotten by f, without moving the others. In other words, all block positions that are allowed in m_2 were already allowed in m_1, therefore we avoid the problems illustrated by Example 3.

The proof of Lemma 1 goes by counting 8-byte wide and 8-byte aligned regions of memory that we call *boxes*, delimited by dashed lines in Fig. 7. Our allocation algorithm [4] entails that for every memory state m, there exists a concrete memory cm that we call the canonical concrete memory of m and write $\texttt{canon\_cm}(m)$, that is built by allocating all the blocks of m at *maximally-aligned*, i.e. 8-byte aligned, addresses. We call $\texttt{nbox}(cm)$ the number of used boxes for a given concrete memory cm. For example, we have $\texttt{nbox}(cm_2) = 2$, and $\texttt{nbox}(\texttt{canon\_cm}(m_2)) = 3$. In general, thanks to alignment constraints, we have that for any memory m and any concrete memory cm valid for m, cm uses no more boxes than $\texttt{canon\_cm}(m)$, i.e. $\texttt{nbox}(cm) \leq \texttt{nbox}(\texttt{canon\_cm}(m))$. This is a direct consequence of the relation between the size and the alignment properties of blocks. More precisely, a block b of size s has an alignment m such that $s < 2^m$, when b is a small block (smaller than 8 bytes, i.e. those that are likely to be forgotten). Due to this alignment property, a properly aligned small block cannot span over more than 1 box. Larger blocks are 8-byte aligned and therefore use as many boxes in any valid concrete memory. This reasoning could be extended to slightly different definitions of well-behaved injections where larger blocks can be forgotten, hence considering larger boxes, so that properly aligned forgotten blocks never span over more than 1 box.

Consider now two memory states m_1 and m_2 in injection by some well-behaved injection function f, such that m_2 is the result of *forgetting* F blocks from m_1. We have that $\mathtt{nbox}(\mathtt{canon\_cm}(m_2)) = \mathtt{nbox}(\mathtt{canon\_cm}(m_1)) - F$. This can be verified on Fig. 7, where $F = 2$ blocks have been forgotten, $\mathtt{nbox}(\mathtt{canon\_cm}(m_1)) = 5$ and $\mathtt{nbox}(\mathtt{canon\_cm}(m_2)) = 3$, indeed satisfying the equation.

Starting from a concrete memory $cm_2 \vdash m_2$, we derive that $\mathtt{nbox}(cm_2) + F \leq \mathtt{nbox}(\mathtt{canon\_cm}(m_1))$. In other words, it is possible to find F free boxes in cm_2. In our example, those 2 boxes can be for example the boxes $[8; 16[$ and $[24; 32[$. Because the blocks we forgot each fit in a box, all we have to do at this point is fill each of these F boxes in cm_2 with the F forgotten variables. The result is the concrete memory cm_1 shown in the last line of Fig. 7.

Theorem 4 is the generalised version of Theorem 3 for well-behaved injections.

Theorem 4 *For any well-behaved injection f, for any memory states m_1 and m_2 in injection by f, for any symbolic values sv_1 and sv_2 in injection by f, the normalisations of sv_1 in m_1 and of sv_2 in m_2 are in injection by f.*

Proof The proof is performed in two steps.

- First, we exhibit some value v such that the normalisation of sv_1 injects into v. This shows that if the normalisation of sv_1 is a pointer, then this pointer is injected by f. This is a consequence of the fact that sv_1 is injected into another symbolic value.
- Then, we show that this v is necessarily the normalisation of sv_2 in m_2. This boils down to showing that: $\forall\ cm_2 \vdash m_2,\ [\![v]\!]_{cm_2} = [\![sv_2]\!]_{cm_2}$. Using Lemma 1 and the specification of the normalisation, we conclude this proof.

This theorem is a central piece of the proof of the SimplLocals pass, which is now fully proved in CompCertS. It is worth noting that we did not modify the behaviour of the SimplLocals pass. The work we have done here is simply to strengthen the proof so that the original SimplLocals pass is still correct with our more defined semantics, in particular with respect to the set of valid concrete memories across memory injections.

5 Optimisations

CompCert features several standard optimisations. Among them, constant propagation, strength reduction and common sub-expression elimination exploit the result of a dataflow analysis computing the combination of a numeric analysis and an alias analysis. In this section, we explain why the existing dataflow transfer functions are not sound for CompCertS and how to fix them. This demonstrates that the semantics of CompCertS is a provably strong safeguard preventing the miscompilations of low-level pointer arithmetic.

For the sake of explanation, we will present a simplified version of CompCert's abstract domains and transfer function that is sufficient for our needs. A more thorough description can be found in [19].

5.1 The abstract value domain of CompCert

The abstract value domain of CompCert is made of a pointer domain and a numeric domain. The purpose of the pointer domain is to infer aliasing information and get an abstract model of memory reads and writes. In particular, if the current stack pointer does not escape through global variables or arguments of functions, the compiler gets the valuable information that the content of the current stack frame cannot be modified by function calls. A representative but simplified abstract domain of pointers, *aptr*, is given below.

$$aptr ::= \bot \mid Stk\ ofs \mid Stack \mid \neg Stack \mid \top$$

Its semantics is given by its concretisation function γ_{sb} where *sb* stands for the memory block of the current stack frame. The empty set of pointers is denoted by $\bot$. *Stk o* represents the stack pointer $\texttt{ptr}(sb, o)$. The set of all pointers to the current stack frame (block *sb* at any offset) is captured by *Stack*. All pointers to blocks different from the stack block *sb* are abstracted by $\neg Stack$. Finally, $\top$ is the set of all pointers.

$$\begin{array}{ll} \gamma_{sb}(\bot) & = \{\} \\ \gamma_{sb}(Stk\ o) & = \{\texttt{ptr}(sb,o)\} \\ \gamma_{sb}(Stk) & = \{\texttt{ptr}(sb,o) \mid o \in int\} \\ \gamma_{sb}(\neg Stack) & = \{\texttt{ptr}(b,o) \mid b \neq sb \wedge o \in int\} \\ \gamma_{sb}(\top) & = \{\texttt{ptr}(b,o) \mid b \in blocks \wedge o \in int\} \end{array}$$

The numeric domain *anum* tracks constant values and intervals of the form $[0; 2^n-1]$ and $[-2^n; 2^n-1]$.

$$anum ::= \bot \mid Cst\ c \mid [0; 2^n-1] \mid [-2^n; 2^n-1] \mid \top$$

Conceptually, the domain of abstract values is of the form $aval = aptr \times anum$ such that $\gamma_{sb}(ap, an) = \gamma_{sb}(ap) \cup \gamma_n(an)$. The union of concretisations is relevant because a value can be either a pointer or an integer but not both. Moreover, as certain operators may return the value `undef`, `undef` belongs to every concretisation of the numeric domain i.e. $\texttt{undef} \in \gamma_n(\bot)$.

According to the original semantics of CompCert, the bitwise conjunction & between a pointer $\texttt{ptr}(b, o)$ and an integer $\texttt{int}(i)$ returns `undef`. As a result, the most precise transfer function for the bitwise & is such that

$$(p, \top)\&(\bot, \top) = (\bot, \top)$$

For the pointer part, it returns $\bot$ because a bitwise & with a pointer argument returns `undef` (it cannot be a pointer). For the integer part, it returns $\top$ because a bitwise & between arbitrary integers is still an arbitrary integer. This formulation is semantically sound. Yet, as shown by Example 4, this aggressive transfer function can be responsible for miscompilation.

Example 4 Consider the red-black tree code of Fig. 8. The code is annotated by the result of a sound dataflow analysis using the previous domain. At function entry, the current stack frame has just been created and is therefore free of aliases. As a result, the parameter r and the local variable rpc can be abstracted by $(\neg Stack, \top)$. Line 6, the aggressive analysis is using the previous transfer function for the bitwise & and obtains $(\bot, \top)$ for the abstraction of p. This makes the reasoning that p can only be an integer. As the dereference of an integer has no semantics, the aggressive analysis infers that the rest of the code is not reachable. Line 8, this is encoded by $\emptyset$. Based on this information, a live-variable analysis

```
rb_node* get_parents_right_child(rb_node* r){
//  r: (¬Stack,⊤)
uintptr_t rpc = r->rb_parent_color;//get the parent/color field
// rpc: (¬Stack,⊤)
rb_node* p = (rb_node*) (rpc & ~3);//get the parent of r
// p: (⊥,⊤)
rb_node* rchild = p->rb_right;      // access its right child
// ∅
return rchild; }
```

Fig. 8: Aggressive dataflow analysis for red-black trees

and an aggressive dead-code removal could replace the whole function body by a no-op which is obviously a miscompilation.

To avoid such dramatic effects, the transfer functions of CompCert are written with *prudence* with the objective of preventing miscompilations and "*[track] leakage of pointers through arithmetic operations*".[2] This is done by computing carefully crafted transfer functions which are purposely non-optimal in order to prevent aggressive optimisations (which would be sound by relying on undefined behaviours of the CompCert semantics). For instance, the transfer function for the bitwise & becomes:

$$(p,\top)\&(\bot,\top) = (\widehat{p},\top)$$

where $\widehat{p}$ reads as *provenance of the pointer* p and has the informal meaning that the result is some value derived from the pointer p and is defined by:

$$\widehat{p} = \mathit{if}\ \ p = \mathit{Stk}\ \ o\ \ \mathit{then}\ \ \mathit{Stack}\ \ \mathit{else}\ \ p.$$

This formulation is semantically sound and *prudent*. Yet, this is not completely satisfactory because it is not grounded on any palpable semantics notion.

5.2 A formally prudent dataflow analysis.

With our semantics, the program of Figure 8 may have a defined semantics, hence the aggressive dataflow analysis of Example 4 is not sound and therefore no such miscompilation can occur. The reason is that, for our semantics, arithmetic operations (e.g. the bitwise &) are always defined and compute symbolic values. To adapt the existing abstract domains to our semantics, we need to adapt the concretisation so that they denote symbolic values instead of values. A direct lifting consists in using the evaluation of symbolic values. This approach is effective for the numeric domain and we get: $\gamma_n^*(an) = \{sv \mid \forall cm, [\![sv]\!]_{cm} \in \gamma_n(an)\}$.

For the pointer domain, the same lifting is such that the concretisation of the *Stack* element represents *any* symbolic value whose evaluation has value $[\![sb]\!]+o$ for some o. As o is unrestricted, this concretisation captures any symbolic expression and collapses with the $\top$ element. A more restricted lifting could be based on the `normalise` function. This appealing option is however too restrictive because it rules out symbolic values which may not have a normalisation. Interestingly, we

[2] See https://github.com/AbsInt/CompCert/blob/a968152051941a0fc50a86c3fc15e90e22ed7c47/backend/ValueDomain.v#L707.

eventually noticed that, to get a concretisation that is both sound and robust to syntactic variations, what was needed was a formal account of pointer tracking. It is formalised, using Definition 4, by a notion of pointer *dependence* of a symbolic value sv with respect to a set S of memory blocks.

Definition 4 A symbolic value sv depends at most on the set of blocks S if sv evaluates identically in concrete memories that are identical for all the blocks in S. Formally, we have:

$$dep(sv, S) \triangleq \forall\ cm \equiv_S cm',\ [\![sv]\!]_{cm} = [\![sv]\!]_{cm'}$$

where $cm \equiv_S cm' \triangleq \forall\ b \in S, cm(b) = cm'(b)$.

Note that, for any other block $b \notin S$, the memory may differ arbitrarily. The concretisation function γ^*_{sb}, where sb is the current stack block, is defined in Fig. 9.

$$\begin{array}{ll}
\gamma^*_{sb}(\bot) & = \{\} \\
\gamma^*_{sb}(Cst) & = \{sv \mid dep(sv, \emptyset)\} \\
\gamma^*_{sb}(Stk\ o) & = \{sv \mid \forall cm, [\![sv]\!]_{cm} = cm(sb) + o\} \\
\gamma^*_{sb}(Stack) & = \{sv \mid dep(sv, \{sb\})\} \\
\gamma^*_{sb}(\neg Stack) & = \{sv \mid dep(sv, block \setminus \{sb\}) \\
\gamma_{sb}(\top) & = \{sv \mid sv \in sval\}
\end{array}$$

Fig. 9: CompCertS concretisation for the pointer domain

Intuitively, Cst represents any symbolic value which always evaluates to the same value whatever the concrete memory (*i.e.*, it does not depends on pointers); $Stack$ represents any symbolic value which depends at most on the current stack block sb and $\neg Stack$ represents any symbolic value which may depend on any block except the current stack block sb. Our abstract domain is still a pair of values $(ap, an) \in aptr \times anum$ but it represents a (reduced) product of domains. For symbolic values, there is no syntactic distinction between pointer and integer values. Hence, the concretisation is given by an intersection of concretisations (instead of a union): $\gamma_{sb}(ap, an) = \gamma_{sb}(ap) \cap \gamma_n(an)$.

In CompCert, a *prudent* transfer function for the pointer domain is defined by $\widehat{p_1} \sqcup \widehat{p_2}$. Theorem 5 gives the formal guarantee that this transfer function is sound for our semantics.

Theorem 5 *Suppose that sv_1 is modelled by the abstract pointer p_1 and sv_2 is modelled by the abstract pointer p_2. The symbolic value $sv_1 \bowtie sv_2$ is modelled by the least upper bound of the provenance of p_1 and p_2 i.e.*

$$sv_1 \in \gamma_{sb}(p_1) \land sv_2 \in \gamma_{sb}(p_2) \Rightarrow sv_1 \bowtie sv_2 \in \gamma_{sb}(\widehat{p_1} \sqcup \widehat{p_2})$$

Depending on the operator, the transfer function can be specialised sometimes using additional information from the numeric domain. In particular, for bitwise operators, we have the following transfer functions.

$$\begin{array}{l}
p_1 \& p_2 = if\ p_1 = p_2 = Stk\ o\ then\ Stk\ o\ else\ \widehat{p_1} \sqcup \widehat{p_2} \\
p_1 \mid p_2 = if\ p_1 = p_2 = Stk\ o\ then\ Stk\ o\ else\ \widehat{p_1} \sqcup \widehat{p_2} \\
p_1 \,\hat{}\, p_2\ = if\ p_1 = p_2 = Stk\ o\ then\ Cst\ else\ \widehat{p_1} \sqcup \widehat{p_2}
\end{array}$$

When the pointer is known to be a constant of the form $\mathtt{ptr}(sb, o)$, the transfer functions exploits numeric properties of bitwise operators. In particular, they exploit the property that bitwise & and bitwise | are idempotent i.e.

$$\texttt{ptr}(sb, o) \& \texttt{ptr}(sb, o) = \texttt{ptr}(sb, o) \mid \texttt{ptr}(sb, o) = \texttt{ptr}(sb, o)$$

For bitwise ^, we have that $\texttt{ptr}(sb, o) \hat{~} \texttt{ptr}(sb, o) = \texttt{int}(0)$. In the pointer domain, the most precise abstraction is *Cst*. This is however an example where the pointer domain may refine the numeric domain as we have:

$$(\mathit{Stk}\ o, \top) \hat{~} (\mathit{Stk}\ o, \top) = (\mathit{Cst}, [\mathit{0}; \mathit{0}])$$

While adapting the proof, we found and fixed several minor but subtle *bugs* in CompCert related to pointer tracking, where the existing transfer functions were unsound for our low-level memory model. Though unlikely, each of them could potentially be responsible for a miscompilation. For instance, the right shift operator $x >> y$ ignores the leak of information that would be due to the shift amount y. Though it makes little sense to pass a pointer as a shift amount, there is nonetheless some form of information flow that is captured by our semantics and forces our transfer function to include the dependence $\widehat{y}$.

Using its more conservative dataflow analysis, CompCertS forbids program transformations that are otherwise valid for CompCert but may result in miscompilations. In this particular case, we generate the right code not because our optimisations are designed with prudence but because our more defined semantics provides a formal safeguard.

5.3 Instruction selection and symbolic values

For dataflow analysis, our semantics makes optimisations more conservative. Yet, a more defined semantics may also enable new optimisations that would be unsound for a less defined semantics. This phenomenon has already been observed e.g. by Muellen *et al.* [17] in the context of peephole optimisations for CompCert. The motivating example of Muellen *et al.* essentially transforms the expression $y-x-1$ into $y+\tilde{~}x$ where $\tilde{~}$ is bitwise negation. In CompCert, the transformation is unsound because when x and y are pointers to the same block e.g. $\texttt{ptr}(b, o)$ and $\texttt{ptr}(b, o')$, the expression $y - x - 1$ evaluates to $\texttt{int}(o' - o - 1)$ but the expression $y + \tilde{~}x$ evaluates to `undef` because of the bitwise negation that is undefined for pointers. With our semantics, both expressions have the same evaluation:

$$[\![y - x - 1]\!]_{cm} = [\![y + \tilde{~}x]\!]_{cm}.$$

and therefore the transformation is sound. We have introduced it in the instruction selection pass which performs strength reduction over the subtraction operator .

There are nonetheless standard transformations that our semantics is unable to validate. For instance, an efficient way of setting a register r to 0 consists in performing a bitwise ^ with itself. Unfortunately, we cannot prove that the symbolic values 0 and $sv \hat{~} sv$ have always the same evaluation. A counterexample is when sv evaluates to `undef` because

$$[\![0]\!]_{cm} \neq [\![\texttt{undef}]\!]_{cm} \hat{~} [\![\texttt{undef}]\!]_{cm} = \texttt{undef}.$$

For our semantics, this is a corner case because the optimised expression depends on more variables than the original expression. In order to perform this optimisation, CompCert introduces, at assembly level, a pseudo instruction which has the semantics of setting a register r to 0 and is assembled as a genuine bitwise ^. This approach also works for our semantics.

6 Preservation of Memory Consumption

The C standard does not impose a model of memory consumption. In particular, there is no requirement that a conforming implementation should make a disciplined use of memory. A striking consequence is that the possibility of stack overflow is not mentioned. From a formal point of view, CompCert models an unbounded memory and therefore, as the C standard, does not impose any limit on stack consumption of the binary code. As a result, the existing CompCert theorem is oblivious to memory consumption of the assembly code. Though CompCert makes a wise usage of memory, this is not explicit in the correctness statement and can only be assessed by a thorough inspection of the code.

Our memory model is finite and the memory allocation fails when no more memory is available. As a consequence, in order to prove the forward simulations for each compiler pass, we now also need to show the preservation of memory allocation steps. This means there is more proof effort required, but also that CompCertS provides a stronger formal guarantee about memory consumption than CompCert. It ensures that if the source code does not exhaust the memory, then neither does the assembly code. In other words, the compilation ensures that the assembly code consumes no more memory than the source code does.

Although this memory consumption preservation behaviour could exist in its own right (without symbolic values and normalisation), the converse is not true: we need to have a finite memory so that at least one concrete memory exists for every memory state, and we need to preserve a bound on the memory across compiler passes.

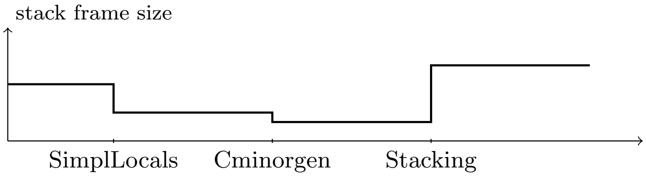


Fig. 10: Evolution of the size of stack frames

6.1 Evolution of Stack Memory Usage throughout Compilation

The memory is split into three distinct uses in CompCert: global variables, dynamically allocated memory (e.g. through malloc) and stack memory. The memory for global variables is statically known and dynamically allocated memory does not change throughout the compilation passes. Only the stack memory is deeply impacted by the compiler. Fig. 10 shows the evolution of the size of the stack frame for one given function across compiler passes. We define the size of a stack frame as the sum of the maximally aligned sizes of its blocks. Formally, if a stack frame is composed of blocks $\{b_1, \ldots, b_n\}$, the size is defined as:

$$\texttt{size\_frame}(\{b_1, \ldots, b_n\}, m) \triangleq \sum_{i}^{n} \texttt{next\_aligned}(\texttt{size}(m, b_i), 8)$$

where `next_aligned`(x, a) returns the smallest integer larger than or equal to x which is divisible by a. Reasoning about maximally aligned sizes of blocks is consistent with our allocation algorithm (see [5]) and will be important in the following. Three passes are distinguished, which modify the memory usage:

- First, the SimplLocals pass introduces pseudo-registers for certain variables, which are pulled out of memory. This pass reduces the memory usage of functions and therefore satisfies the requirement that compilation should reduce memory usage.
- Then, the Cminorgen pass allocates a unique stack frame containing all the remaining variables of a function. This pass may introduce some padding to ensure proper alignment properties. However, the size of the frames always decreases, thanks to the fact that we are considering maximally aligned sizes, therefore we have already accounted for the maximal amount of padding necessary. It might even be the case that we have counted too much padding and the global size of the frame will decrease. Hence, this pass preserves the memory usage.
- Finally, the remaining problematic pass is the Stacking pass which builds activation records from stack frames. This pass makes explicit some low-level data (e.g. the return address or the space for spilled locals) and is responsible for an increase of the memory usage. In the following, we explain how we solve this discordance and ensure nonetheless a decreasing usage of memory across the compiler passes.

6.2 The Stacking Compiler Pass

The Stacking pass transforms Linear programs into Mach code. The Linear stack frame consists of a single block containing the local variables of the function. The Mach stack frame embeds the Linear stack frame together with additional data, namely the return address of the function, the spilled pseudo-registers that could not be allocated in machine registers, the callee-save registers, and the outgoing arguments to function calls.

6.2.1 Provisioning memory.

In order to fit the Stacking pass into the *decreasing memory usage* framework, our solution is to provision memory from the beginning of the compilation chain, i.e. from the C language. Hence, we parameterise the semantics of all intermediate languages, from C to Linear, with an oracle *ns* which specifies, for each function f, the additional space that is needed. The semantics therefore include special operations that reserve some space at function entry and release it at function exit. Below are the relevant rules for the RTL language (other languages have similar, if not identical rules).

FunEntry

$$\frac{\texttt{alloc}\ m_1\ 0\ (\texttt{stacksize}\ f) = \lfloor m_2, \mathit{stk} \rfloor \qquad \texttt{reserve\_boxes}\ m_2\ (\mathit{ns}\ f) = \lfloor m_3 \rfloor}{\texttt{Callstate}\ s\ f\ \mathit{args}\ m_1 \rightarrow \texttt{State}\ s\ f\ \mathit{stk}\ (\texttt{entrypoint}\ f)\ (\mathit{init\_rs}\ f\ \mathit{args})\ m_3}$$

FunExit

$$\frac{f!\texttt{pc} = \lfloor \texttt{Ireturn}\ r \rfloor \\ \texttt{free}\ m_1\ \mathit{stk}\ 0\ (\texttt{stacksize}\ f) = \lfloor m_2 \rfloor \qquad \texttt{release\_boxes}\ m_2\ (\mathit{ns}\ f) = \lfloor m_3 \rfloor}{\texttt{State}\ s\ f\ \mathit{stk}\ \texttt{pc}\ \mathit{rs}\ m_1 \rightarrow \texttt{Returnstate}\ s\ (\mathit{rs}\ r)\ m_3}$$

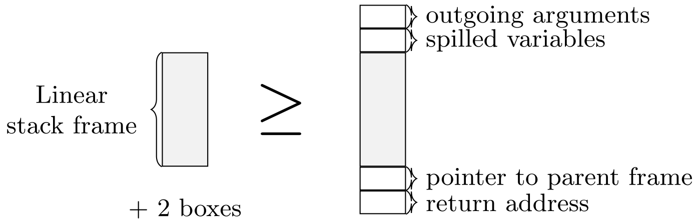
The FunEntry rule describes the transition from a `Callstate`, with a call stack s (which represents the stack of program points in parent functions where the execution should return afterwards, i.e. a continuation) where we are just about to enter a function f with arguments *args* in memory state m, to a regular `State` with the appropriate stack block *stk*, program counter `entrypoint` f, register state initialised from the arguments $\mathit{init\_rs}\ f\ \mathit{args}$ and memory m_3 set up. In CompCert, the end memory is simply the result of allocating the stack block with the `alloc` operation of the memory model. In CompCertS, we also reserve a number of *boxes* (the same notion of boxes that was defined in Section 4) with the `reserve_boxes` operation for the additional space that will be needed to concretely lay out the stack frame of the function at the Mach and assembly levels.

Symmetrically, the FunExit rule describes the transition from a regular `State` where the program counter points to an `Ireturn` r instruction (return with the value stored in register r). In this case, the resulting state is a `Returnstate` with an updated memory state m_3. In CompCert, m_3 is simply the result of freeing (deallocating) the stack block *stk*. In CompCertS, we also release the appropriate number of boxes with the `release_boxes` operation.

In the Mach and assembly languages, no more boxes are reserved or released because the stack is completely laid out and no extra memory will be needed in the future.

These boxes that we reserve and release are just abstract information that we keep in the memory state but are not related to actual memory blocks. We maintain the invariant that the size of all blocks plus the size of all reserved boxes does not exceed some predetermined threshold . For most compiler passes, the amount of boxes reserved for a function call doesn't change and these reserve and release operations are easy to preserve across these passes. For the Stacking pass, we leverage these boxes associated with the Linear function call to justify the larger stack block in Mach.

Consider the example in the following picture. On the left, the stack frame for Linear is represented, together with 2 additional boxes. On the right, the stack frame for Mach is represented: no additional boxes are reserved but the stack block is larger to accomodate for the outgoing arguments to function calls, spilled variables, or the return address. The oracle *ns* is correct if the amount of boxes that is reserved is sufficient to hold this extra space in Mach. In such a case, we maintain that the memory usage for a Linear function is not smaller than the memory usage for the corresponding Mach function and therefore preserve that the memory usage for the whole program in Mach does not exceed the maximum memory size we allow.



The question of how to compute such a correct oracle *ns* remains to be discussed. It may be possible to derive an over-approximation of the needed stack space for each function from a static analysis. However, the estimate would probably be very rough as, for instance, it seems unlikely that the impact of register allocation could be modelled accurately. Instead, as the exact amount of additional memory space is known during the Stacking pass, we construct the oracle *ns* as a byproduct of the compilation. In other words, the compiler returns not only an assembly program but also a mapping that associates with each function of the program the quantity of additional stack space required. Note that the construction is not circular since the oracle is only needed for the correctness proof of the compiler and not by the compiler itself. CompCertS' final theorem takes the form of Theorem 6.

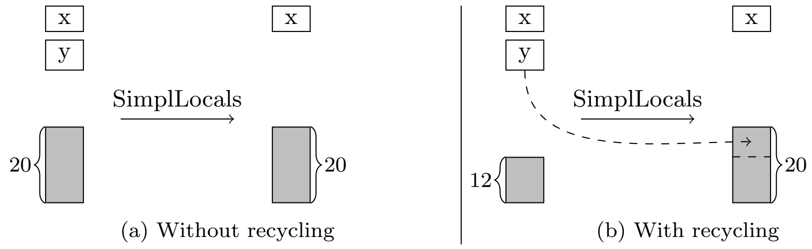
Theorem 6 *Suppose that (tp, ns) is the result of the successful compilation of the program p. If tp has the behaviour bh', then there exists a behaviour bh such that bh is a behaviour of p with oracle ns and bh' improves on the behaviour bh.*

$$bh' \in ASem(tp) \Rightarrow \exists bh. bh \in CSem(p, ns) \wedge bh \subseteq bh'.$$

The only difference with CompCert is that the C semantics is instrumented by the oracle *ns* computed by the compiler. Though not completely explicit, Theorem 6 ensures that the absence of memory overflows is preserved by compilation. The fundamental reason is that the failure to allocate memory results in an observable going wrong behaviour. On the contrary, if the source code does not have a going wrong behaviour, neither does the assembly. It follows that if the C source succeeds at allocating memory, so does the assembly. Hence, CompCertS ensures that the absence of memory overflows is preserved by compilation.

6.2.2 Recycling memory.

The semantics of function calls now reserve some amount of memory space on top of the space for the stack data. Since this operation may fail if too much memory is requested, we should thrive to make this amount as low as possible so that as many programs as possible have a defined semantics. We have seen that our oracle *ns* accurately predicts the total amount of stack space that will be needed at the Mach level (by construction), however some compiler passes – SimplLocals in particular – may forget some blocks and therefore *throw away* some memory space. We can reuse this freed space and therefore have a weaker requirement on the source semantics. To do so, we introduce another parameter *sl* (for SimplLocals) that gives for every function the amount of memory space that will be freed by SimplLocals, and that can therefore not be reserved in advance with a `reserve_boxes` operation.



(a) Without recycling

(b) With recycling

Fig. 11: Recycling memory

Example 5 Consider a function with long-integer local variables x and y, as illustrated in Fig. 11, where $ns(f) = 20$ additional bytes are needed for the Stacking pass. During SimplLocals , y is transformed into a temporary while x is kept and allocated on the stack. The naive first solution that we implemented was to reserve directly from the C level the 20 needed bytes, as shown in Fig. 11a. However, this results in over provisioning memory because we request 36 bytes in total (2 long-typed variables and 20 reserved bytes), where we need no more than 28 bytes in the next compilation stage. Instead of throwing away the space for the y variable, we can reuse it as additional space (see Fig. 11b). As a result we only require 12 additional bytes at the C level, or 28 bytes in total. This memory consumption then stays the same in the next compilation stage.

The amount of requested stack space is therefore lower at the C level than it would be using the naive approach of requesting the whole amount necessary. Below is a picture representing the amount requested for a selection of intermediate languages, for a function f. The parameter sl is also obtained as a byproduct of the compiler, just like the oracle ns discussed above.

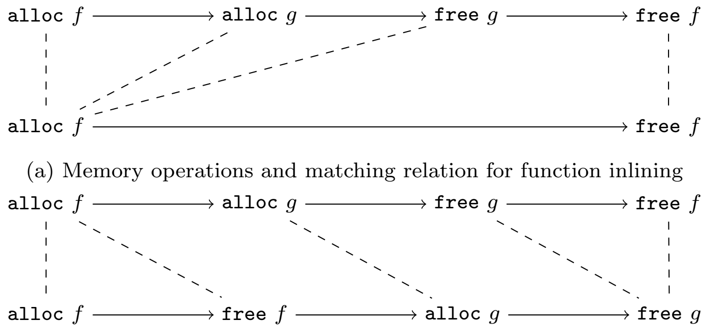
$$\begin{tikzcd}
\boxed{\text{C}} \arrow[r] & \boxed{\text{Clight}} \arrow[r] & \boxed{\text{RTL}} \arrow[r] & \boxed{\text{Mach}} \arrow[r] & \boxed{\text{Asm}} \\
ns(f) - sl(f) & ns(f) & ns(f) & 0 & 0
\end{tikzcd}$$

Using this recycling principle, we slightly relax the requirements for having a defined C semantics, therefore making our formal semantic preservation theorem applicable to more programs.

6.3 About function inlining and tailcall recognition

Our current implementation of CompCertS does not support compiler optimisations that deeply modify the structure of stack blocks such as function inlining and tailcall recognition. We briefly explain the difficulties raised by these optimisations and sketch our ideas to deal with those in future work.

Those optimisations change the order in which stack blocks are allocated/freed and additional boxes are reserved/released. Looking only at stack blocks allocations/deallocations, the inlining of a function f into a function g transforms the sequence of events `alloc` f; `alloc` g; `free` g; `free` f into the sequence `alloc` f; `free` f (as shown in Fig. 12a). If the function call to g gets transformed into a tail-call,



(a) Memory operations and matching relation for function inlining

(b) Memory operations and matching relation for tailcall recognition

Fig. 12: Transformations induced by function inlining and tailcall recognition

the same sequence becomes `alloc` f; `free` f; `alloc` g; `free` g instead (as shown in Fig. 12b).

Fig. 12 pictures the matching relation (with dashed lines) that we should capture between source and target programs. The issue is that all theorems we have about memory injection and memory allocation/deallocation require that every operation in the source program has a matching operation in the target program, and the transformations induced by function inlining and tailcall recognition do not fit in that setting. Instead, there are allocations and deallocations that have no counterpart in the target program (for the inlined function); or operations are reordered, making it impossible to use the available lemmas. While appropriate lemmas exist in the original CompCert, they are more subtle to prove in CompCertS because allocations and deallocations affect the set of valid concrete memories and therefore the behaviour of normalisations: it is unclear how to preserve the behaviour of normalisations in such cases; a more thorough study of these transformations is needed to reprove such theorems.

We would also need to record a subtle relation between the sizes of the memories in the source and target programs, to capture the fact that the target program has already freed its stack block (and associated provisioned memory boxes), while the source program has not yet (e.g. in the second matching of Fig. 12b).

7 Related Work

Formal semantics for C. The first formal realistic semantics of C is due to Norrish [18]. More recent works [9,12,13] aim at providing a formal account of the subtleties of the C standard. Hathhorn *et al.* [9] present an executable C semantics within the K framework. They extend the previous work of Ellison *et al.* [8] to precisely characterise the undefined behaviours of C. Krebbers [12,13] gives a formal account of sequence points and non-aliasing. These notions are probably the most intricate of the ISO C standard. Memarian *et al.* [16] realise a survey among C experts, in which they aim at capturing the *de facto* semantics of C. They remark that uninitialised values and pointer arithmetic are commonly used.

Our work builds upon the CompCert C compiler [14]. The semantics and the memory model used in the compiler are close to ISO C. Our previous works [3, 4] show how to extend the support for pointer arithmetic and adapt most of the front-end of CompCert to this extended semantics with the notable exception of the SimplLocals pass which requires a sophisticated proof argument detailed in the present paper.

CompCert *and memory consumption.* CompCert observes the I/O behaviour of programs but not their resource usage. Carbonneaux *et al.* [7] propose a logic for reasoning, at source level, on the resource consumption of target programs compiled by CompCert. They instrument the event traces to include resource consumption events that are preserved by compilation, and use the compiler itself to determine the actual size of stack frames. We borrow from them the idea of using a compiler-generated oracle. Their approach to finite memory is more lightweight than ours and does not require modifying the memory model. However, our ambition to reason about symbolic values in CompCert requires more intrusive changes.

CompCertTSO [20] is a version of CompCert implementing a TSO relaxed memory model. It also models a finite memory where pointers are pairs of integers. Their soundness theorem is oblivious to out-of-memory errors. They remark that they could exploit memory bounds computed by the compiler, but do not implement it. In terms of expressiveness, their semantics and ours seem to be incomparable. For instance, CompCertTSO gives a defined semantics to the comparison of arbitrary pointers, we do not. That is because our semantics requires that the evaluation of symbolic values is the same in every valid concrete memory, and a comparison p1 < p2 may evaluate differently depending on the memory layout, if p1 and p2 are pointers to different objects; this would therefore result in undefined behaviour, just like in CompCert. Yet, the example of Section 2.3.1 is not handled by the formal semantics of CompCertTSO.

Pointers as integers. Kang *et al.* [11] propose a hybrid memory model where an abstract pointer is mapped to a concrete address at pointer-integer cast time. Their semantics may get stuck at cast-time if there is not enough memory available. For our semantics, a cast is a no-op and our semantics may get stuck at allocation time. They study aggressive program optimisations but do not preserve memory consumption. In CompCertS, we consider simpler optimisations but implemented in a working compiler for a real language. Moreover, we ensure that the memory consumption is preserved by compilation. Mullen *et al.* [17] present Peek, a framework to certify peephole optimisations within CompCert. Peek leverages a low-level memory model, $\mathrm{ASM}_{\mathbb{Z}32}$, for the assembly language of CompCert where pointers are integers. This more defined semantics allows to validate peephole optimisations that are unsound for the more abstract model of CompCert. They give an axiomatic definition of a memory allocator and prove that, in the absence of memory exhaustion, their low-level memory model simulates the memory model of CompCert. In CompCertS, we provide a stronger guarantee and ensure the preservation of memory usage using a more high-level memory model. In theory, because our `normalise` function may return `undef`, our semantics is less defined than $\mathrm{ASM}_{\mathbb{Z}32}$. Nonetheless, we believe that most, if not all, of the peephole optimisations presented by Mullen *et al.* are also sound for our semantics.

8 Conclusion

We present CompCertS, an extension of the CompCert compiler that is based on a more defined semantics and provides additional guarantees about the compiled code. Programs performing low-level bitwise operations on pointers are now covered by the semantics preservation theorem, and can thus be compiled safely. CompCertS also guarantees that the compiled program does not require more memory than the source program. This is done by instrumenting the semantics with an oracle providing, for each function, the size of the stack frame.

CompCertS compiles down to assembly; compared to CompCert, we adapted all the 4 passes of the front-end and 12 out of 14 passes of the back-end. This whole work amounts to more than 210k lines of Coq code, which is 60k more than the original CompCert 2.4. This is the result of approximately 3 person years. CompCertS does not feature the inlining and tailcall optimisations. The inlining optimisation may increase the memory consumption of functions. This disagrees with our decreasing memory size policy, but we should be able to provision memory in a similar way as we did for the Stacking pass. The tail call recognition transforms regular function calls into tail calls when appropriate. Its proof cannot be adapted in a straightforward way because of the additional stack space we introduced for the Stacking pass: the release of those blocks does not happen at the same place before and after the transformation. We need to investigate further the proof of this optimisation and come up with a more complex invariant on memory states.

As future work, we shall investigate how security-related program transformations would benefit from the increased expressiveness of CompCertS. Recently, Blazy and Trieu [6] pioneered the integration of state-of-the-art obfuscations within CompCert. Data obfuscations based on bitwise operations cannot be proved sound for pointers with CompCert. Lastly, currently every function stores its stack frame in a distinct block, even at the assembly level. An ultimate compiler pass that merges blocks into a concrete stack is possible with our finite memory and would bring even more confidence in CompCertS.

Acknowledgments

This work has been partially funded by the ANR project AnaStaSec ANR-14-CE28-0014, NSF grant 1521523 and DARPA grant FA8750-12-2-0293.

References

1. Companion website. URL http://www.cs.yale.edu/homes/wilke-pierre/jar18/
2. Bedin Franca, R., Blazy, S., Favre-Felix, D., Leroy, X., Pantel, M., Souyris, J.: Formally verified optimizing compilation in ACG-based flight control software. In: ERTS 2012: Embedded Real Time Software and Systems (2012)
3. Besson, F., Blazy, S., Wilke, P.: A precise and abstract memory model for C using symbolic values. In: APLAS, *LNCS*, vol. 8858 (2014)
4. Besson, F., Blazy, S., Wilke, P.: A concrete memory model for CompCert. In: ITP, *LNCS*, vol. 9236. Springer (2015)
5. Besson, F., Blazy, S., Wilke, P.: A Verified CompCert Front-End for a Memory Model supporting Pointer Arithmetic and Uninitialised Data. Journal of Automated Reasoning pp. 1–48 (2017). URL https://doi.org/10.1007/s10817-017-9439-z

6. Blazy, S., Trieu, A.: Formal verification of control-flow graph flattening. In: CPP. ACM (2016)
7. Carbonneaux, Q., Hoffmann, J., Ramananandro, T., Shao, Z.: End-to-end verification of stack-space bounds for C programs. In: PLDI. ACM (2014)
8. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. SIGPLAN Not. **47**(1) (2012). URL http://doi.acm.org/10.1145/2103621.2103719
9. Hathhorn, C., Ellison, C., Rosu, G.: Defining the undefinedness of C. In: PLDI. ACM (2015)
10. ISO: ISO C Standard 2011. Tech. rep. (1999)
11. Kang, J., Hur, C., Mansky, W., Garbuzov, D., Zdancewic, S., Vafeiadis, V.: A formal C memory model supporting integer-pointer casts. In: PLDI (2015)
12. Krebbers, R.: Aliasing restrictions of C11 formalized in Coq. In: CPP, *LNCS*, vol. 8307. Springer (2013). doi: 10.1007/978-3-319-03545-1_4. URL http://dx.doi.org/10.1007/978-3-319-03545-1_4
13. Krebbers, R.: An operational and axiomatic semantics for non-determinism and sequence points in C. In: POPL. ACM (2014)
14. Leroy, X.: Formal verification of a realistic compiler. C. ACM **52**(7), 107–115 (2009). URL http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf
15. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning **41**(1) (2008)
16. Memarian, K., Matthiesen, J., Lingard, J., Nienhuis, K., Chisnall, D., Watson, R.N., Sewell, P.: Into the depths of C: elaborating the de facto standards. In: PLDI. ACM (2016)
17. Mullen, E., Zuniga, D., Tatlock, Z., Grossman, D.: Verified peephole optimizations for CompCert. In: PLDI, pp. 448–461. ACM (2016). doi: 10.1145/2908080. URL http://doi.acm.org/10.1145/2908080
18. Norrish, M.: C formalised in hol. Ph.D. thesis, University of Cambridge (1998)
19. Robert, V., Leroy, X.: A formally-verified alias analysis. In: CPP, *LNCS*, vol. 7679. Springer (2012). URL http://gallium.inria.fr/~xleroy/publi/alias-analysis.pdf
20. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: CompCertTSO: A verified compiler for relaxed-memory concurrency. J. ACM **60**(3), 22:1–22:50 (2013). doi: 10.1145/2487241.2487248. URL http://doi.acm.org/10.1145/2487241.2487248