

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

Frédéric Besson¹, Sandrine Blazy², and Pierre Wilke³

¹ Inria, Rennes, France

² Université Rennes 1 - IRISA, Rennes, France

³ Yale University, USA

Abstract. The COMP CERT 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, COMP CERTS, which is essentially the COMP CERT 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. COMP CERTS is based on an enhanced memory model where, unlike COMP CERT but like GCC, the binary representation of pointers can be manipulated much like integers and where, unlike COMP CERT, allocation may fail if no memory is available.

The whole proof of COMP CERTS 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.

1 Introduction

Over the past decade, the COMP CERT compiler has established a milestone in compiler verification. COMP CERT is a formally verified C compiler written with the COQ proof assistant, which initially targeted safety-critical embedded software. 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, COMP CERT only provides formal guarantees for programs that do not exhibit undefined behaviours – a property that is in general undecidable.

COMP CERT’s memory model is a central component of the compiler. In this paper, we show how to adapt COMP CERT for a more expressive memory model which lifts two main limitations. First, memory allocation in COMP CERT 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, COMP CERT’s memory model limits pointer arithmetic: any implementation-defined operations on pointers 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 show how to tackle this challenge using oracles, aiming at ensuring 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 safer 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 (i.e. 12 compiler passes) including constant propagation, common subexpression elimination and dead-code elimination. In particular, we detail how the existing alias analyses of COMPCERT [15] 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 highlights the proof challenges related to treating pointers as integers. In particular, we explain the impact on optimisations and on the proof of one important pass of the front-end of COMPCERT. Section 4 shows how we ensure that the compiler reduces the memory usage of programs and proves that the absence of memory overflows is preserved. Section 5 mentions related work and finally, Section 6 concludes.

2 Background on CompCert

This section describes the architecture of the COMPCERT compiler [12]. It also summarises the main features and properties of our memory model [3,4].

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

Fig. 1: Runtime and memory values

2.1 Architecture of the CompCert Compiler

COMP CERT compiles C programs into assembly code, through 8 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. Every transformation from one language to another is proved to be semantics preserving using simulation relations, stating that every step in the source language can be *simulated* by a number of steps in the target language, such that some matching relation between program states is preserved by those steps. The composition of all the simulation lemmas for the individual compiler passes forms the semantic preservation theorem given below. For the sake of simplicity, we consider that the semantics observe behaviours that are either defined behaviours, with a trace of I/O events, or undefined behaviours.

Theorem 1. *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 COMP CERT is the cornerstone of the semantics of all 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 $\text{int}(i)$, a pointer or the token undef . A pointer is a pair $\text{ptr}(b, o)$ consisting of a block identifier b and an offset o . COMP CERT 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, COMP CERT 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 $\text{Byte}(b)$, a special Undef byte that represents uninitialised memory, or a byte-sized fragment of a pointer value $\text{Pointer}(b, o, n)$ (read: n -th byte of pointer $\text{ptr}(b, o)$). Therefore, a pointer $\text{ptr}(b, o)$ is encoded in memory as a sequence of 4 memvals , from $\text{Pointer}(b, o, 0)$ to $\text{Pointer}(b, o, 3)$. The memory model exports four 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

$$\begin{aligned}
sval \ni sv &:= val \mid \mathbf{unop}(u, sv) \mid \mathbf{binop}(b, sv_1, sv_2) \\
smemval \ni smv &:= \mathbf{Symbolic}(sv, n)
\end{aligned}$$

Fig. 3: Symbolic runtime and memory values

2.3 A Symbolic Memory Model for CompCert

In previous work [3], 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. We first give a motivating example; then we recall the principles of symbolic values and their normalisations.


Motivation for Pointers as Integers. In previous work [4], we introduced a low-level memory model that enables reasoning about the bit-level encoding of pointers within COMPCERT. We give in Fig. 2 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...100`). 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.

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 $\mathbf{Symbolic}(sv, n)$, 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 $\text{ptr}(b, o)$. Similarly, the condition cond of a conditional statement must be normalised into an integer $\text{int}(i)$ 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.

Concrete Memories and Evaluation. A concrete memory is a mapping from blocks to concrete addresses, represented as 32-bit integers. Each memory block b has a size $size$ and an alignment constraint al ; a pointer $\text{ptr}(b, o)$ is *valid* if the offset o is within the bounds $[0, size[$, written $\text{valid}(m, b, o)$. We can retrieve the alignment of a block b with the accessor $\text{align}(m, b)$.

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

The evaluation of a symbolic value sv in a concrete memory cm (written $\llbracket sv \rrbracket_{cm}$) 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_1 that maps a block b to the address 32. The evaluation of the symbolic value $sv = \text{ptr}(b, 5) \& \text{int}(1)$ results in $\text{int}(1)$ because $\llbracket sv \rrbracket_{cm} = (cm(b) + 5) \& 1 = (32 + 5) \& 1 = 37 \& 1 = 1.$

Specification of the normalisation. The normalisation of sv in m returns a value v if for every $cm \vdash m$, sv and v evaluate identically in cm .

$$(\forall cm \vdash m \Rightarrow \llbracket sv \rrbracket_{cm} = \llbracket v \rrbracket_{cm}) \Rightarrow \text{normalise}(m, sv) = v$$

If no such value v can be found, the normalisation returns undef .

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 $\text{ptr}(b, 0)$ is $sv = \text{ptr}(b, 0) \mid 3$. 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 $\text{ptr}(b, 0)$ because for any valid concrete memory cm :

$$\llbracket sv' \rrbracket_{cm} = \llbracket (\text{ptr}(b, 0) \mid 3) \& \sim 3 \rrbracket_{cm} = (cm(b) \mid 3) \& \sim 3 = cm(b)$$

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

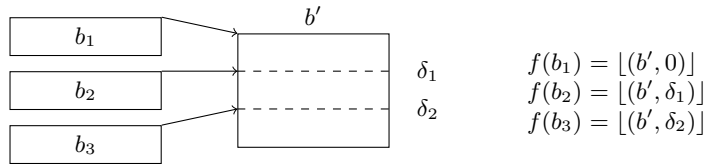



Fig. 4: Injecting several blocks into one

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 $C\sharp\text{minor}$ to $C\text{minor}$. At the $C\sharp\text{minor}$ level, each local variable is allocated in its own block. In $C\text{minor}$, a single block contains all the local variables, stored at different offsets. This mapping from local variable blocks in $C\sharp\text{minor}$ to offsets in the stack block in $C\text{minor}$ is captured by a memory injection. A memory injection is characterised by an injection function $f : \text{block} \rightarrow [\text{block} \times \mathbb{Z}]$ 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 $\text{ptr}(b, o)$, v_2 is $\text{ptr}(b', o + \delta)$ and $f(b) = [(b', \delta)]$ ⁴. For example, in Fig. 4, the pointer $\text{ptr}(b_2, o)$ is in injection with the pointer $\text{ptr}(b', o + \delta_1)$.

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. We proved a central result that relates injections and normalisations, recalled in Theorem 2.

Theorem 2.  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 2, which covers the case of more general injections. As we shall see in Section 3.1, it is required to prove the `SimplLocals` pass of COMPCERT.

⁴ $[\cdot]$ denotes the option type. We write $[v]$ for `Some(v)` and \emptyset for `None`.

3 Proof Challenges for Pointers as Integers

This section presents the proof challenges that we tackle for porting COMPCERT to a semantics with symbolic values, where pointer operations behave as integer operations, e.g. bitwise operators are defined on pointers. The first challenge concerns the SimplLocals pass of COMPCERT, which modifies the structure of the memory. The second challenge is 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.

3.1 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 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 all the subsequent optimisations can operate.

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. $f(b) = [b, 0]$), 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 2 which proves this preservation for total injections. Here, the injection is partial (i.e. some blocks are not injected) and therefore Theorem 2 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. Consider 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. 5a 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 $sv = \text{ptr}(b, 0)! = 16$. It normalises into 0 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. 5b 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.

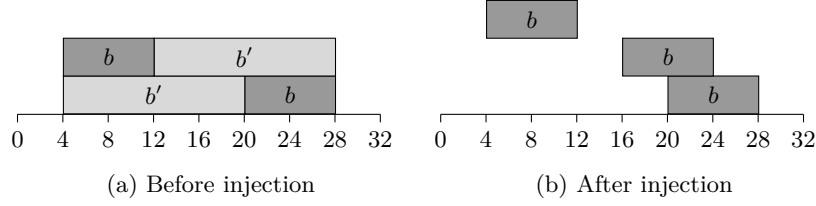


Fig. 5: Concrete memories and partial injections

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 2 cannot be generalised for arbitrary partial injections.

Well-behaved injections. We identify a restricted class of *well-behaved* injection 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 defined in Definition 2.

Definition 2 (Well-behaved injection). \clubsuit An injection function f is said to be well-behaved if only the blocks that are at most 8-byte wide and at most 8-byte aligned may be forgotten by f . Formally,

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

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 such well-behaved injections, we can prove Lemma 1, from which a generalised version of Theorem 2 can be derived, as we explain at the end of this section.

Lemma 1. \clubsuit 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,


$$\forall f, \text{well_behaved } f \Rightarrow \forall m_1 m_2, \text{mem_inject } f m_1 m_2 \Rightarrow \forall cm_2 \vdash m_2, \exists cm_1 \vdash m_1 \wedge cm_1 \equiv_f cm_2 \text{ where } cm_1 \equiv_f cm_2 \triangleq \forall b b', f(b) = [(b', 0)] \Rightarrow cm_1(b) = cm_2(b')$$

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*. We call $\text{nbox}(cm)$ the number of used boxes for a given concrete memory cm . 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 $\text{canon_cm}(m)$, that is built by allocating all the blocks of m at *maximally-aligned*, i.e. 8-byte aligned, addresses. Thanks to alignment constraints, we have that for any concrete memory cm valid for m , cm uses no more boxes than $\text{canon_cm}(m)$, i.e. $\text{nbox}(cm) \leq \text{nbox}(\text{canon_cm}(m))$.

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 $\text{nbox}(\text{canon_cm}(m_2)) = \text{nbox}(\text{canon_cm}(m_1)) - F$. Starting from a concrete memory $cm_2 \vdash m_2$, we derive that $\text{nbox}(cm_2) + F \leq \text{nbox}(\text{canon_cm}(m_1))$. In other words, it is possible to find F free boxes in cm_2 . Because the blocks we forgot each fit in a box, all we have to do at this point is use each of these F boxes to contain the F forgotten variables.

Theorem 3 is the generalised version of Theorem 2 for well-behaved injections.

Theorem 3.  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: $\forall cm_2 \vdash m_2, \llbracket v \rrbracket_{cm_2} = \llbracket sv_2 \rrbracket_{cm_2}$. 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.

3.2 Optimisations

COMPCERT features several standard optimisations. Among them, constant propagation, strength reduction and common subexpression elimination exploit the result of a dataflow analysis computing the combination of an interval 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.

The abstract value domain of CompCert is made of the sum of a pointer domain and a numeric domain. One purpose of the pointer domain is to distinguish pointers to the current stack frame from other pointers. A representative

```

1 rb_node* get_parents_right_child(rb_node* r){ // r: ( $\neg Stack$ ,  $\top$ )
2 uintptr_t rpc = r->rb_parent_color;//get the parent/color field
3 // rpc: ( $\neg Stack$ ,  $\top$ )
4 rb_node* p = (rb_node*) (rpc & ~3);//get the parent of r
5 // p: ( $\perp$ ,  $\top$ ), ( $\neg Stack$ ,  $\top$ )
6 rb_node* rchild = p->rb_right; // access its right child
7 // rchild: ( $\perp$ ,  $\perp$ ), ( $\neg Stack$ ,  $\top$ )
8 return rchild; }

```

Fig. 6: Dataflow analysis for red-black trees

but simplified abstract pointer-domain ($aptr$) is given below. Its semantics is given by its concretization function γ_{sb} where sb is the memory block of the current stack frame. The empty set of pointers is denoted by \perp . $Stk\ ofs$ represents the stack pointer $ptr(sb, ofs)$. 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.

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

The numeric domain $anum$ is standard: it tracks intervals of integers and floating-point constants. The domain of abstract values $aval = aptr \times anum$ is the sum domain such that $\gamma_{sb}(ap, an) = \gamma_{sb}(ap) \cup \gamma(an)$. The sum domain is relevant because a value can be either a pointer or an integer but not both.

In COMPCERT, the transfer functions are written with *prudence* in order to avoid miscompilations and "[Track] leakage of pointers through arithmetic operations".⁵ This is done by computing carefully crafted transfer functions which are purposely non-optimal in order to prevent aggressive optimisations (which are sound by relying on undefined behaviours of the COMPCERT semantics). For instance, the most precise transfer function for a bitwise $\&$ is such that

$$(\neg Stack, \top) \& (\perp, \top) = (\perp, \top).$$

For the pointer part, it returns \perp because a bitwise $\&$ between pointers 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 but is not *prudent* because several bits of the pointer may leak through the bitwise $\&$.

Example 4. To illustrate the severe consequence of not tracking the leakage of pointers, consider the red-black tree code of Fig. 6. The code is annotated by an aggressive dataflow analysis and a *prudent* dataflow analysis, both being semantically sound. When both analyses differ (e.g. Lines 5 and 7), we write the aggressive result first. 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 5, the aggressive analysis is using the previous transfer function for the bitwise $\&$ and obtain (\perp, \top) for the

⁵ See <https://github.com/AbsInt/CompCert/blob/a968152051941a0fc50a86c3fc15e90e22ed7c47/backend/ValueDomain.v#L707>.

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 7, this is encoded by the abstraction (\perp, \perp) for the variable `rchild`. Based on this information, a live-variable analysis and an aggressive dead-code removal could replace the whole function body by a no-op which is obviously a miscompilation.

A formally prudent dataflow analysis. With our semantics, the aggressive dataflow analysis of Example 4 is not sound and therefore such miscompilations cannot occur. The reason is that our semantics computes symbolic values for arithmetic operations (e.g. the bitwise $\&$) that need to be captured by the concretisation function. Interestingly, we 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 by a notion of pointer *dependence* of a symbolic value sv with respect to a set S of memory blocks. We say that 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 ; they may differ arbitrarily for other blocks. Formally, $dep(sv, S) = \forall cm \equiv_S cm', \llbracket sv \rrbracket_{cm} = \llbracket sv \rrbracket_{cm'}$, where $cm \equiv_S cm' = \forall b \in S, cm(b) = cm'(b)$. The concretisation function γ_{sb} , where sb is the current stack block, is defined in Fig. 7. Intuitively, Cst represents any symbolic value which always evaluates to the same value whatever the concrete memory (*i.e.*, it does not depend 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)

$$\gamma_{sb}(ap, an) = \gamma_{sb}(ap) \cap \gamma(an),$$

where the concretisation of the numeric abstract domain is defined in terms of the evaluation of symbolic expressions: $\gamma(an) = \{sv \mid \forall cm, \llbracket sv \rrbracket_{cm} \in \gamma(an)\}$. With this formulation, the most precise transfer function for a bitwise $\&$ is given by $(\neg Stack, \top) \& (\perp, \top) = (\neg Stack, \top)$.

For the pointer part, it returns $\neg Stack$ because the resulting expression may still depend on a $\neg Stack$ pointer. For the integer part, it returns \top because (like before) a bitwise $\&$ between arbitrary integers is still an arbitrary integer.

$$\begin{aligned} \gamma_{sb}(\perp) &= \{\} & \gamma_{sb}(\top) &= sval \\ \gamma_{sb}(Cst) &= \{sv \mid dep(sv, \emptyset)\} \\ \gamma_{sb}(Stk\ o) &= \{sv \mid \forall cm, \llbracket sv \rrbracket_{cm} = cm(sb) + o\} \\ \gamma_{sb}(Stack) &= \{sv \mid dep(sv, \{sb\})\} \\ \gamma_{sb}(\neg Stack) &= \{sv \mid dep(sv, block \setminus \{sb\})\} \end{aligned}$$

Fig. 7: COMPCERTS concretisation for alias analysis

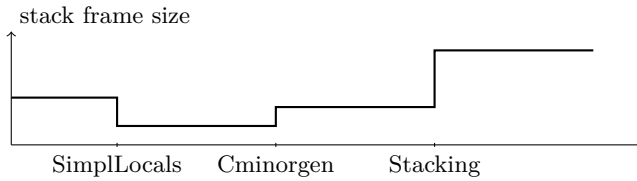


Fig. 8: Evolution of the size of stack frames

As a result, the aggressive transfer function of COMPCERTS implements the informally *prudent* transfer functions of COMPCERT. It follows that, for our semantics, miscompilation due to pointer leaking (e.g. Example 4) is impossible.

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. Note that COMPCERTS generates the right code not by chance but really because our semantics forbids program transformations that are otherwise valid for COMPCERT. In general, we believe that our semantics provides the right safeguard for avoiding any miscompilation of programs performing arbitrary arithmetic operations on pointers.

4 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 never 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 of 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 close inspection of the code. COMPCERTS provides a stronger formal guarantee. It ensures that if the source code does not exhaust the memory, then neither does the assembly code. Said otherwise, the compilation ensures that the assembly code consumes no more memory than the source code does.

4.1 Evolution of Stack Memory Usage Throughout Compilation

Fig. 8 shows the evolution of the size of stack frames across compiler passes. The figure distinguishes the three passes 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 our requirement that compilation reduces memory usage. The

Cminorgen pass allocates a unique stack frame containing all the remaining variables of a function. This pass makes the memory usage grow because some padding is inserted to ensure proper alignment. However, because our allocation strategy considers maximally aligned blocks, this pass still preserves the memory usage. 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. return address or spilled locals) and is responsible for an increase of the memory usage. In the following, we explain how to solve this discordance and ensure nonetheless a decreasing usage of memory across the compiler passes.


4.2 The Stacking Compiler Pass

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

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. Hence, we instrument 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. To justify that the Mach stack frame fits into our finite memory space, we can now leverage the fact that at the Linear level, there was enough space for the Linear stack frame plus $ns(f)$ additional bytes. Provided that the oracle ns is correct, this entails that the Mach stack frame fits in memory.

It may be possible to derive an over-approximation of the needed stackspace 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 function that associates with each function 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.

COMP CERTS' final theorem takes the form of Theorem 4.

Theorem 4.  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 4 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.

Recycling memory. Because our semantics are now parameterised by a bound on the memory usage of functions, this bound should be as low as possible so that as many programs as possible can be given a defined semantics.

In order to give a smaller bound, we notice that the SimplLocals pass forgets some blocks and therefore throws away some memory space. We can reuse this freed space and therefore have a weaker requirement on the source semantics.

Example 5. Consider a function with long-integer local variables *x* and *y*. During SimplLocals, *x* is transformed into a temporary while *y* is kept and allocated on the stack. During Stacking, say 20 additional bytes are needed to build the Mach activation record from the Linear stack frame. Then, we must reserve those 20 bytes from the beginning, i.e. from the C semantics. However, we can recycle the space from the local variable *x*, therefore saving 8 bytes and we only require 12 bytes at the C level, therefore making it easier to have a C semantics.

5 Related Work

Formal semantics for C. The first formal realistic semantics of C is due to Norrish [14]. More recent works [7,10,9] aim at providing a formal account of the subtleties of the C standard. Hathhorn *et al.* [7] present an executable C semantics within the K framework which precisely characterise the undefined behaviours of C. Krebbers [10,9] 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.* [13] realise a survey among C experts, in which they aim at capturing the *de facto* semantics of C. They consider problems such as uninitialised values and pointer arithmetic.

Our work builds upon the COMPCERT C compiler [12]. 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.

COMPCERT and memory consumption. Carbonneaux *et al.* [6] 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. However, our ambition to reason about symbolic values in COMPCERT requires more intrusive changes.

CompCertTSO [16] 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 of 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. Yet, the example of Section 2.3 is not handled by the formal semantics of CompCertTSO.

Pointers as integers. Kang *et al.* [8] 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.

6 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 backend. This whole work amounts to more than 210k lines of Coq code, which is 60k more than the original COMPCERT 2.4 we started with. COMPCERTS does not feature the two following optimization passes. First, the inlining optimisation makes functions use potentially more stack space after the transformation than before. 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, as described in Section 4.2. Second, 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 releases of those blocks do 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. Kroll

et al. [11] implement software isolation within COMPCERT. However, the transformation they define depends on a pointer masking operation which has no COMPCERT semantics and is therefore axiomatised. In COMPCERTS, pointer masking is defined and the isolated program could benefit from all the existing optimisations. Recently, Blazy and Trieu [5] pioneered the integration of an obfuscation pass within COMPCERT. Our semantics paves the way for aggressive obfuscations, which cannot be proved sound for pointers with COMPCERT.

Lastly, currently every function stores its stack frame in a distinct block, even in assembly. An ultimate compiler pass that merges blocks into a concrete stack would be possible with our finite memory and would bring even more confidence in COMPCERTS.

References

1. Companion website. <http://www.cs.yale.edu/homes/wilke-pierre/itp17/>.
2. R. Bedin Franca, S. Blazy, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS 2012: Embedded Real Time Software and Systems*, 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*, pages 449–468, 2014.
4. F. Besson, S. Blazy, and P. Wilke. A concrete memory model for CompCert. In *ITP'15*, volume 9236 of *LNCS*, pages 67–83. Springer, 2015.
5. S. Blazy and A. Trieu. Formal verification of control-flow graph flattening. In *CPP'16*, pages 176–187. ACM, 2016.
6. Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *PLDI '14*. ACM, 2014.
7. C. Hathhorn, C. Ellison, and G. Rosu. Defining the undefinedness of C. In *PLDI'15*, pages 336–345. ACM, 2015.
8. J. Kang, C. Hur, W. Mansky, D. Garbuzov, S. Zdancewic, and V. Vafeiadis. A formal C memory model supporting integer-pointer casts. In *PLDI'15*, 2015.
9. R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *CPP*, volume 8307 of *LNCS*. Springer, 2013.
10. R. Krebbers. An operational and axiomatic semantics for non-determinism and sequence points in C. In *POPL*. ACM, 2014.
11. J. A. Kroll, G. Stewart, and A. W. Appel. Portable software fault isolation. In *CFS 2014*, pages 18–32. IEEE, 2014.
12. X. Leroy. Formal verification of a realistic compiler. *C. ACM*, 52(7):107–115, 2009.
13. K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. Watson, and P. Sewell. Into the depths of C: elaborating the de facto standards. In *PLDI'16*.
14. M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.
15. V. Robert and X. Leroy. A formally-verified alias analysis. In *CPP 2012*, volume 7679 of *LNCS*, pages 11–26. Springer, 2012.
16. J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *J. ACM*, 60(3):22:1–22:50, June 2013.