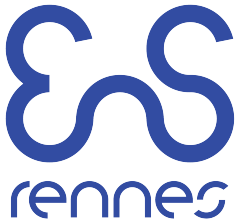
Implementing a C Memory Model Supporting Integer-Pointer Casts in CompCert

Aurèle Barrière

École Normale Supérieure de Rennes, France
aurele.barriere@ens-rennes.fr

Supervisor: Professor Chung-Kil Hur
Software Foundations Laboratory
Seoul National University, South Korea

May 15 2017 - August 4 2017



Abstract. The ISO standard for the C programming language does not define semantics for integer-pointer casts. The certified C compiler CompCert uses an abstract memory model which allows for many optimizations, but in which the behavior of such casts is undefined. In [10], Kang et al. present a formal memory model that supports integer-pointer casts semantics, while still allowing common optimizations. We show the relevance of this new memory model by implementing it in CompCert. We present the changes that need to be done, both in the way CompCert transforms C programs and in the proofs.

Keywords: CompCert, C Memory Model, Integer-Pointer Cast, Optimization

1 Introduction

When compiling critical software written in C, one expects from the compiler to not introduce any bugs, or any behavior that wasn't specified in the C source code. To meet this expectation, CompCert [13] is a formally verified C compiler. It uses the Coq Proof Assistant [5] to prove that the compiled code and the source code have the same observable behavior, as defined by the ISO C Standard [2]. CompCert also aims to provide performance of the generated code, and implements several common optimizations. Compiled code runs approximately 10% slower than code compiled with `GCC4 -O1` [4]. As of today, CompCert is a trusted compiler; despite many efforts [22], no bug have been discovered within the verified parts of CompCert. CompCert currently supports all of the ISO C 99 Standard, with very few exceptions [4].

However, the ISO C standard itself does not define semantics for every syntactically valid C program. Many C programs are said to have *unspecified behavior* or *undefined behavior*, meaning that conforming compilers can produce any compiled code. Despite the lack of semantics, many C programmers are using such programs and expect a precise result. This leads to difficult bugs [20] and the impossibility of proving that the compiled code behaves as expected.

One popular unspecified feature of the C language is the casting between integer and pointer values. Such casts have many uses in real C programs. For instance, pointer to integer cast is used in the Linux Kernel or JVM implementations for bit-wise operations on pointers. Integer arithmetic on pointers is used in Linux, FreeBSD, QEMU and others [1]. Another common usage is to use the bit representation of a pointer as an indexing key of a hash table (used for instance in the C++ standard library). When compiled with most compilers, those programs behave as expected from the programmers. But these intuitive semantics have not been formalized in the C standard.

Defining a precise, formal semantics for integer-pointer casting and pointer manipulation would allow CompCert to compile even more C programs in a certified way. The semantics of pointer manipulation depends on the memory model of the compiler. As of today, CompCert uses a logical memory model [14], where every memory block is an abstract object without a concrete memory address. Such a memory model enables many optimizations, because a program can never guess the location of a block and modify it without a pointer (see section 2). However, integer-pointer casting isn't possible. Other works have investigated the use of a concrete memory model, to reflect the memory state of a real machine [19][16]. But then, most optimizations cannot be done anymore without changing the behavior of the program.

In [10], Kang et al suggest a quasi-concrete model, in which there are both logical and concrete memory blocks. The main idea is to use logical blocks by default, that can allow optimizations, and use concrete blocks when the concrete address of a memory block is needed.

We implemented this new memory model in CompCert. In this paper, we discuss this implementation. We show that it is relevant and supports integer-

pointer casts while still allowing common optimizations. We present the difficulties of the implementation, and the changes that needed to be done in CompCert.

At first, we remind the reader about the different memory models in section 2. Then, we present the work that has been done in CompCert. In section 3, we show how the definition of memory have been modified to fit into CompCert. In section 4, we show how the definition of memory injection have been modified. It changes a lot of proofs in CompCert. In section 5, we present how adding non-determinism in every language of CompCert prevents us from using the same correctness proofs. We design a new proof, using mixed simulations. It relies on the observation that non-deterministic behavior is only encountered when using the *capture* function. In section 6, we discuss the implementation of the *capture* function. Finally, we discuss in section 7 the results of the implementation and its effect on optimizations.

2 Preliminaries

2.1 CompCert

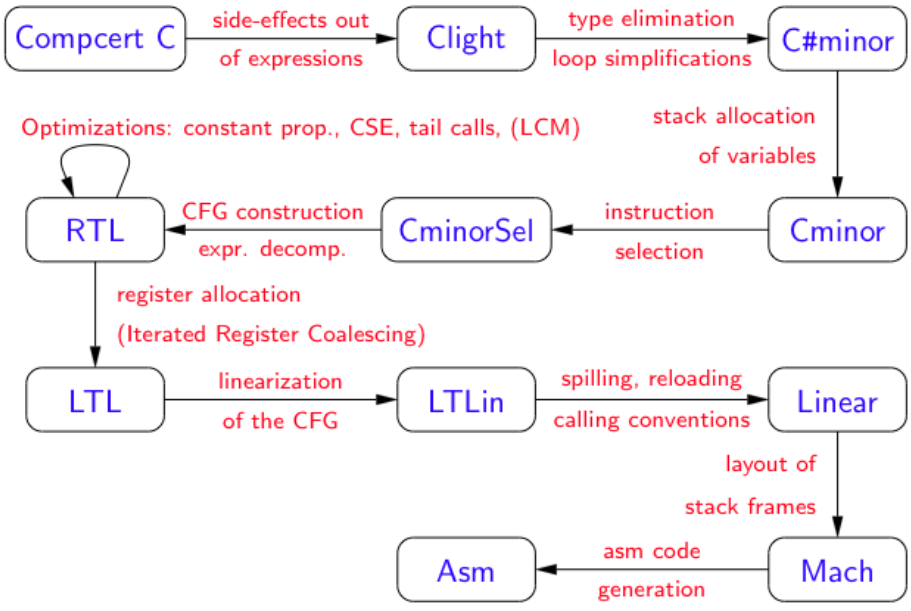


Fig. 1: Verified CompCert passes
Source: `http://compcert.inria.fr/compcert-C.html`

The compiler used in this work is CompCert [3]. It supports most of the ISO-C-99 Standard. It can generate PowerPC, ARM and x86 assembly code. Because formal program verification is often done at source level, CompCert is a promising tool for the development of critical software [6].

Between the source code and the target assembly code, CompCert goes through 25 passes, including several changes of language, all of which using the same memory model. The first three parse and convert the source code to CompCert C, and the last three perform printing of the assembly code, assembling and linking. In the middle, 19 back-end passes perform various transformations, 8 of which are optimizations.

Correctness The parser and most of the back-end passes (see **Fig.1.**) are verified with Coq. In CompCert, the behavior of a program is a trace (a list of I/O operations) and an indication on the program's termination. The correctness theorem of CompCert states that the behavior of the generated code is one of the possible behaviors of the source code (C is non-deterministic). To prove it, it uses a backward simulation (see section 5).

Optimizations CompCert performs the following optimizations: Instruction selection, Common sub-expression elimination, Tail call elimination, Dead code elimination, Function inlining, Branch tuneling, Constant propagation and Register allocation [9]. Some examples are given in Appendix, section 10.1.

2.2 Memory Models

In this section, we present the logical and concrete models with their limits, which motivates the introduction of the quasi-concrete model that has been implemented in this work.

In C semantics, the memory is divided in several *memory blocks*, that contain several memory addresses. Memory blocks can be allocated (through the `malloc` function for instance), loaded, freed... A memory block can contain the values of several variables.

Logical Model The logical model described here is similar to the one used in [10]. However, it slightly differs from CompCert's current logical model described in [14] (see section 3). In this model, blocks are all logical, meaning that they are not mapped to a physical memory address. Blocks are a fixed-size array of values. A validity flag v indicates if the block has been freed. Pointers are a pair (l, i) of a block identifier and an offset inside that block. The set `Block` of blocks is defined with:

$$\texttt{Block} = \{(v, n, c) \mid v \in \texttt{Bool}, n \in \mathbb{N}, c \in \texttt{Val}^n\}$$

With a logical model, programs have infinite memory. Moreover, a logical model can allow functions to have exclusive control over a logical block. When

a block identifier does not escape when calling a function (*i.e.* if a pointer to the block identifier cannot be found inside the global variables or the function arguments), then the called function cannot access the block (see section 3). This allow for many optimizations (see Figure 2).

However, logical models do not support integer-pointer casts. They can allow some arithmetic operations on offsets, but it is impossible to get a physical address from a block.

Concrete Model The concrete model aims to reflect the memory of a real machine, to give a more intuitive semantics to pointer manipulation. The memory itself is a 2^{32}-sized array of values, and `Allocated`, a list of allocated blocks. Blocks are simply a pair of a concrete address and a size.

$$\texttt{Block} = \{(p, n) \mid p \in \texttt{int32}, n \in \texttt{int32}\}$$

The concrete memory should require from the allocated blocks to be consistent:

$$\textit{No overflow:} \quad \forall (p, n) \in \texttt{Allocated}, [p, p + n] \subseteq]0, 2^{32}[$$

No overlap: $\forall (p_1, n_1), (p_2, n_2) \in \texttt{Allocated}, [p_1, p_1{+}n_1]$ and $[p_2, p_2{+}n_2]$ are disjoint.

In the concrete model, integer-pointer cast is possible because concrete addresses already are integer values.

However, optimizations such as constant propagation and dead allocation elimination are not supported in many cases, because external functions might change the value of any address. In the concrete model, there is no ownership of memory blocks, and every address is always accessible.

Quasi-concrete model In [10], a new memory model is presented. The motivation is to have a memory model which allows integer-pointer casting, but still supports common optimizations. This is achieved with the following definitions. A block can be either logical or concrete, in which case it has a concrete address. This is represented by the value p.

$$\texttt{Block} = \{(v, p, n, c) \mid v \in \texttt{Bool}, n \in \mathbb{N}, c \in \texttt{Val}^n, p \in \texttt{int32} \cup \{\textit{undef}\}\}$$

The concrete blocks need to be consistent:

$$\textit{No overflow:} \quad \forall (v, p, n, c) \in \texttt{Block}, (p \neq \textit{undef} \wedge v = \textit{true}) \implies [p, p{+}n] \subseteq]0, 2^{32}[$$

$$\textit{No overlap:} \quad \forall (p_1, n_1), (p_2, n_2) \in \texttt{Block}, (p_1 \neq \textit{undef} \wedge p_2 \neq \textit{undef} \wedge v_1 = v_2 = \textit{true}) \implies$$
$$[p_1, p_1 + n_1] \text{ and } [p_2, p_2 + n_2] \text{ are disjoint.}$$

A pointer is a pair (l, i) of a block identifier and an offset inside that block. If the block l starts at the address $p \neq \textit{undef}$, then (l, i) can be cast as the integer $p + i$ and vice versa (thanks to the property of no overlapping, an integer can correspond to at most one valid concrete block).

Optimizations are still possible with logical blocks, because they do not have concrete addresses, and integer-pointer casts are possible with concrete blocks. To allow as many optimizations as possible, we should use as many logical blocks as possible. Thus, new blocks should be made logical when allocated.

The capture function However, for each pointer to integer cast, we need a concrete block. Then we transform each pointer to integer cast by adding a *capture* function just before it. This new builtin function transforms a logical block into a concrete one, giving it a concrete address that still satisfies the memory consistency. It introduces non-determinism in every language of CompCert (including the assembly), because a block can be captured at several addresses. This is handled in section 5.

The quasi-concrete memory model is described in many details in [10].

```
extern void g();
int f(void) {
  int a = 0;

  g();
  return a;
}
```

(a) Logical block example

```
extern void g();
int f(void) {


  g();
  return 0;
}
```

(b) After CP and DAE

```
extern void g();
int f(void) {
  int a = 0;
  int p = (int) &a;
  g();
  return a+p;
}
```

(c) Concrete block example

Fig. 2: Examples of optimizations and casts

Optimization and casts examples Consider the program Fig.2a. Using a logical memory model, Constant Propagation is allowed, because no pointer to the block of `a` is available from `g()`, and thus the external call cannot change the value of `a`. Then, the compiler can replace `return a;` with `return 0;`. After that, Dead Allocation Elimination can remove the allocation of `a`, now unused. The optimized program can be seen Fig.2b. Using a concrete model, the block containing the value of `a` is mapped to a concrete address. Without more information on `g()`, it should be assumed that it might change the value of `a`. Thus, the program cannot be optimized. Using the quasi-concrete model, a new block is allocated for the allocation of `a`. Since it is new, it is a logical block, without a concrete address. Thus, `g()` cannot modify the value of `a` and the program can once again be transformed into Fig.2b.

Consider the program Fig.2c, where the address of `a` is cast as an integer. Unlike with a concrete model, using a logical model defines no semantics for this program. With the quasi-concrete model, the block containing the value of `a` is transformed into a concrete one just before the cast. No optimization is possible, because `g()` might modify any concrete block, but the semantics of the program is defined.

3 Memory update

To begin, the quasi-concrete memory model described in [10] has been implemented in CompCert. However, the definition has to be changed to adapt to CompCert's current memory model [14].

3.1 Adapting the definition to CompCert

In CompCert, the memory is not a list of blocks, but several maps. The first one, `mem_contents`, is a map from block identifiers and offsets to memory values. It describes the content of the memory. The second one, `mem_access`, is a map from block identifiers and offsets to permissions. It gives a permission to each logical address of the memory. The memory also contains the identifier of the next block to be allocated.

We start by adding to the memory a map `mem_concrete` from blocks to a value that is either `None`, for logical blocks, or `Some` address, for concrete blocks.

Because the size of blocks was not remembered by CompCert, we add a map `mem_offset_bounds` from blocks to a pair of numbers, describing the range of offsets for which the block has been allocated. We need to add both bounds, because CompCert does not always allocate blocks starting at offset 0.

CompCert uses the permissions to know what addresses have been freed. This differs from the block-wise validity boolean described in section 2.2. In fact, CompCert's `free` operation is not performed on whole blocks, but on a range of offsets of a block. It allows memory blocks to contain more than one variable.

The new implementation can be found in Appendix, section 10.2.

3.2 Consistency

Because we added concrete addresses to some blocks, we need to define several consistency properties.

We add the predicate `addresses_in_range`, implementing the *No overflow* property of section 2.2. It states that every allocated address is in the range $]0, 2^{32}[$ (the first and last address should not be allocated).

We add the predicate `no_concrete_overlap`, implementing the *No overlap* property of section 2.2. Because the permissions are address-wise in CompCert and blocks can be freed partially, we change the definition of *No overlap*. Informally, we use the following definition: for each concrete address, there is at most one block where the address is inside the allocated range and hasn't been freed. This allows allocating blocks over freed memory.

For every memory operation that changes the memory (allocation, store, free, capture), we prove that it preserves the memory consistency.

3.3 Abstract Analysis

For some optimization passes, CompCert performs abstract analysis of the code.

For this purpose, any function call is analyzed and classed as either *public* or *private*. If the stack block is not accessible for the called function (i.e. if no pointer to the stack can be found in global variables or the function's arguments), then the call is considered private, and CompCert can perform analysis knowing that the called function cannot change the stack block. However, if such a pointer to the stack exists in the function's arguments or in global variables, then the call is considered public, and the optimizations made are weaker.

With the new memory model, the stack block can also be accessible by a called function when the block has been captured and assigned to a concrete address. To reflect that, we changed the abstract memory definition to include a Boolean that states if the stack might have been captured. We also define its least upper bound, and change the way functions calls are analyzed to add that all function calls should be public if the stack block may have been captured.

4 Memory injection

4.1 Memory injection in CompCert

To prove the correctness of many passes, CompCert uses memory injections. Informally, memory injections are relations between two memories that are true when the two memories are similar. For instance, this is used when doing optimizations, to show that the memory of the source program is similar to the memory of the optimized programs.

In CompCert, memory injection are parametrized with an *injection function*, of type `block -> option(block * Z)`. This function establishes a correspondence between blocks of the source memory and blocks of the target memory. Informally, if $f(b_1) = \text{Some}(b_2, o)$, then the block b_1 in the source memory corresponds to the block b_2 in the target memory, with a shift in offsets of o. The values are preserved between the two blocks, and the pointers are modified to reflect the change of logical addresses. The injection function also define *private* and *public memory*. Public memory is the set of blocks that are mapped to some other block. These blocks should be preserved by optimizations. Private memory is the set of blocks that are mapped to `None`. These blocks are only privately used and can be changed separately. For instance, when performing an unknown external call in both the source and target programs, CompCert assumes that it preserves the memory injection. Thus, the new public memories are equivalent but the call may have changed the private part.

4.2 Memory injection in the quasi-concrete model

As described in [10], memory injection using the quasi-concrete model should be stronger.

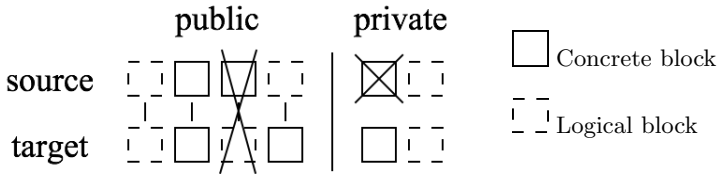


Fig. 3: Memory Injection, taken from [10]

To preserve behavior refinement, any successful memory access in the source memory should succeed as well in the target memory. This has several consequences (Fig. 3). Firstly, when a source concrete block is mapped to another target block with a given offset in a memory injection, the target should be concrete and their concrete addresses should differ with the same offset. Finally, there should be no concrete block in the source's private memory. Formally:

$$\forall b_1, b_2, o, p, \quad f(b_1) = (b_2, o) \wedge b_1 \text{ has address } p \implies b_2 \text{ has address } p + o$$

$$\forall b_1, \quad f(b_1) = \texttt{None} \implies b_1 \text{ has address } \texttt{None}$$

These two properties have been added to CompCert's memory injection. The proofs of every memory injection in CompCert have been updated.

The new implementation can be found in Appendix, section 10.3.

5 Mixed Simulations

In the quasi-concrete memory model, with the addition of the *capture* function, every language of CompCert is now non-deterministic. Indeed, giving a concrete address to a logical block is a non-deterministic operation, whereas the logical allocation used so far in CompCert was deterministic.

5.1 Simulations in CompCert and properties

To prove the correctness of CompCert, one must prove that the behavior of the generated target code is one of the behaviors of the original source code. To do so, CompCert uses *backward simulations* between the semantics of the source code and the generated code. In CompCert, the semantics of each language is defined. The semantics of a program is a set of states (with initial and final states) and a relation between states meaning that one can go from one state to another with a given observable trace.

Definition: Semantics of a program p : $Sem(p) = (States_p, I_p, F_p, Step_p)$ where $Step_p \subseteq Statesp \times Traces \times States_p$ and $I, F \subseteq States_p$. $t \in Traces$ is a trace, *i.e.* a list of observable events (system call, store or load).

A straightforward way to show that two programs p_s and p_t are semantically equivalent could be to show that their semantics are *bisimulated*, meaning that

there exists a relation between states of the source and target semantics such that every time the source program takes a step to a new state, the target program takes a step with the same trace to a matching state, and vice versa. However, such a relation is in general too strong, as it does not allow basic optimizations.

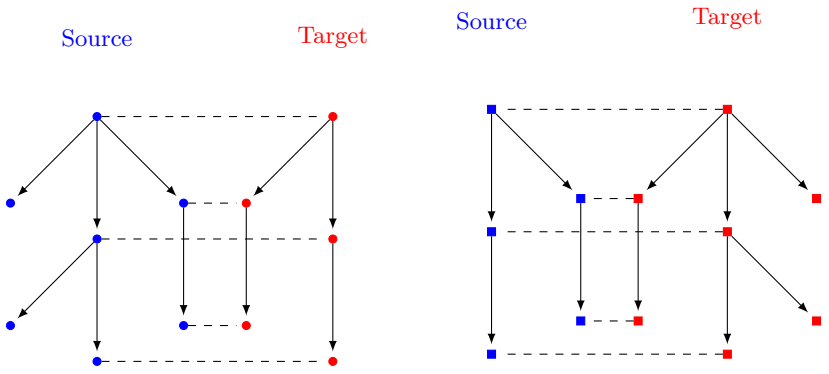
A backward simulation is a weaker simulation that proves semantics preservation while still allowing optimizations.

Definition: Backward Simulation Let sp a source program and tp a target program. A backward simulation is a relation R such that

- $\forall i \in I_{tp}, \exists i' \in I_{sp}, (i, i') \in R$
- $\forall t \in Traces, ss_1 \in States_{sp}, st_1, st_2 \in States_{tp}$,
  $(ss_1, st_1) \in R \ \wedge \ (st_1, t, st_2) \in Step_{tp} \implies$
  $\exists ss_2 \in States_{sp}, (ss_1, t, ss_2) \in Step_{sp}$

Informally, each time the target program takes a step to a new state, the source program also takes a step to a matching state. This is enough to claim that the target's behavior is a refinement of the source's behavior. An example can be seen Figure 4a. The relation is described with the dashed lines. We can see that every target behavior matches a source behavior.

Forward Simulations One can similarly define a forward simulation in a symmetric way: each time the source program takes a step to a new state, the target program also takes a step to a matching state. This means that the target's behavior extends the source's behavior. An example can be seen Figure 4b.



(a) Example of backward simulation

(b) Example of forward simulation

Fig. 4: Examples of simulations

Theorem: forward to backward If the target language is *determinate* (a weaker version of determinism) and the source language is *receptive* and there is a forward simulation between the semantics of *sp* and *tp*, then we can construct a backward simulation between the semantics of *sp* and *tp*. Informally, if the target language is determinate, then for a given trace, there can be only one target step. However, with the source language being receptive, there will exist one matching step for any possible trace. We can then deduce that any target step is matched with a corresponding source step.

Theorems: simulation composition Let S_1, S_2, S_3 be semantics. If there is a forward simulation between S_1 and S_2, and between S_2 and S_3, then there is a forward simulation between S_1 and S_3. If there is a backward simulation between S_1 and S_2, and between S_2 and S_3, then there is a backward simulation between S_1 and S_3.

5.2 Atomic Semantics and properties

Definition: Atomic Semantics $Atomic(States_p, I_p, F_p, Step_p) = (Traces \times States_p, \{[]\} \times I_p, \{[]\} \times F_p, AtomicStep_p)$ where [] is the empty trace and $AtomicStep_p$ is defined as follow:

- $(s_1, [], s_2) \in Step_p \implies ((([], s_1), [], ([], s_2)) \in AtomicStep_p$
- $(s_1, ev :: t, s_2) \in Step_p \implies ((([], s_1), [ev], (t, s_2)) \in AtomicStep_p$
- $((ev :: t, s), [ev], (t, s)) \in AtomicStep_p$

Informally, *Atomic* of a semantics is the same semantics where steps with multiple events traces have been replaced with several single-event steps.

Theorem: Factor backward simulation If there is a backward simulation between S_1 and S_2 and S_1 has *single events*, then there is a backward simulation between S_1 and $Atomic(S_2)$.

5.3 Non-determinism

To prove the correctness of the compilation, we need to prove that for every source C program *sp*, if *tp* is the compiled ASM program, then we have a backward simulation between $Sem(sp)$ and $Sem(tp)$.

Previously, in CompCert, the proof of correctness used forward simulations for almost every pass, then used the forward to backward simulation theorem to deduce a backward simulation between CompCertC and ASM. More details on this proof can be found in Appendix, section 10.4.

However, in the quasi-concrete model, the intermediate and target languages of CompCert are no longer *determinate*. Indeed, the *capture* function can give any address to a logical block, as long as it doesn't overlap with other blocks. Thus, the use of forward simulations is no longer relevant.

5.4 Mixed simulations and properties

To prove the correctness of CompCert with the new memory model, we introduce *mixed simulations*. Informally, at each state of the semantics, either we have a local forward simulation, or a local backward simulation. For most steps, we use forward reasoning (and it can be proved by adapting the previous forward simulation proof that used to be in CompCert), but for external calls (such as *capture* and other unknown functions that might capture blocks), we use a backward reasoning. This is illustrated Figure 5, with round states being the states where an external function or a builtin function is called (these states are called *external states*). Because the capture function is the only non-deterministic function of CompCert C, builtin functions and unknown external functions are the only place where non-deterministic behavior can occur.

Definition: Mixed Simulation Let sp a source program and tp a target program. A mixed simulation is a relation R such that

- $\forall i \in I_{tp}, \exists i' \in I_{sp}, (i, i') \in R$
- $\forall t \in Traces, ss_1 \in States_{sp}, st_1 \in States_{tp}, \quad (ss_1, st_1) \in R \implies ($
  Forward: $\wedge \forall ss_2, (ss_1, t, ss_2) \in Step_{sp} \implies \exists st_2, (st_1, t, st_2) \in Step_{tp} \quad \vee$
  Backward: $\forall st_2, (st_1, t, st_2) \in Step_{tp} \implies \exists ss_2, (ss_1, t, ss_2) \in Step_{sp} \quad)$
- Every state where forward reasoning is applied should have *local determinacy*

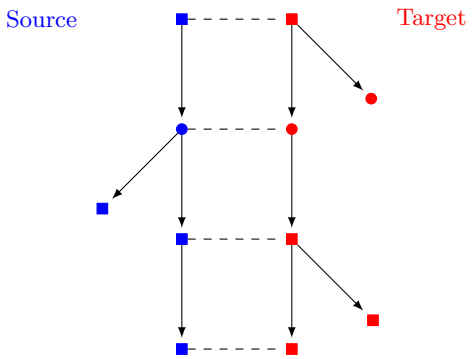


Fig. 5: Example of mixed simulation

Theorem: Mixed to Backward simulation If there is a mixed simulation between S_1 and S_2, then there is a backward simulation between $Atomic(S_1)$ and S_2.

To prove it, we use the local determinacy and local receptivity of states with forward reasoning, in a similar way to the Forward to Backward simulation theorem. This theorem also requires several hypothesis on the source and target

languages (`single_events`, `well_behaved`) that we proved for each mixed simulation used in the correctness proof.

5.5 The new correctness proof

1. Prove a mixed simulation for each pass between CStrategy and ASM. We can use the previous forward simulation proof for every non-external states. The backward simulation between CompCert C and CStrategy is unchanged.
2. We use the mixed to backward simulation theorem on every mixed simulation.
3. We use the factor backward simulation theorem on every backward simulation.
4. We use the composition of backward simulations to deduce a backward simulation between CompCert C and ASM.

The proof is illustrated Figure 6.

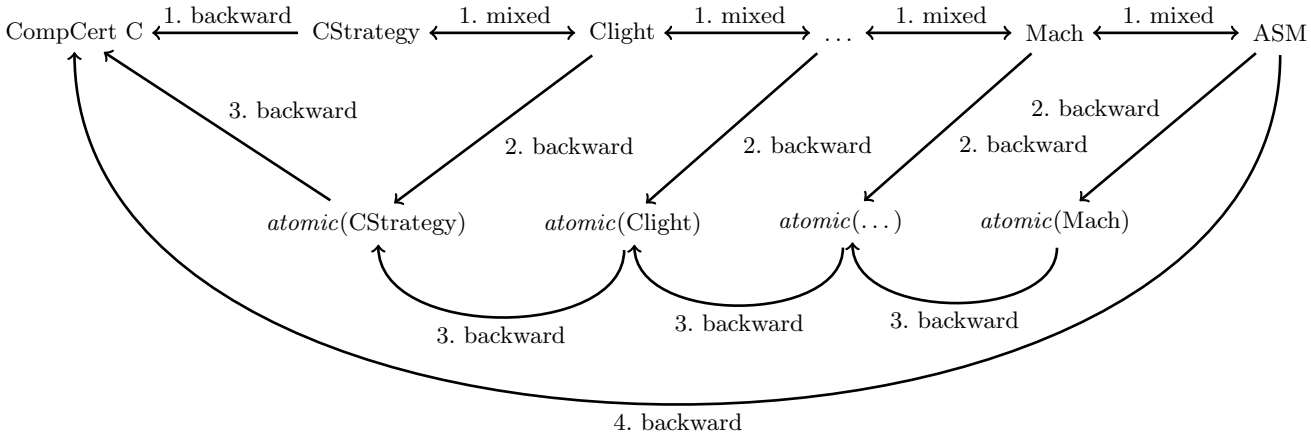


Fig. 6: The new correctness proof

6 Adding The Capture Function

In this section, the implementation has been done by Juneyoung Lee.

The capture function is the new builtin function to turn logical blocks into concrete blocks. To be sure that casting from pointer to integer is always possible, the capture function should be called before every such cast.

The function is added in the first verified pass of CompCert, when translating CompCert C programs into Clight.

The results can be seen on the Figure 7. We can notice the capture of the memory block of `a` just before casting its address to an integer.

```
int main() {
  int a = 0;
  int b = (int) &a;
  return 0;
}
```

(a) C source program

```
int main(void)
{
  int a;
  int b;
  a = 0;
  builtin builtin __capture(&a);
  b = (int) &a;
  return 0;
  return 0;
}
```

(b) Clight program

Fig. 7: CompCert now automatically inserts the capture function before casts

7 Evaluation

The correctness of CompCert with the new memory model hasn't been entirely proved yet. The remaining proofs are most of the proofs of mixed simulations. However, we remain confident that such a mixed simulation exists for every pass. One mixed simulation has been proved (`CSEproof`, an optimization pass), and others should be very similar. The intuition behind any mixed simulation proof is the same: use the previous forward simulation proof to prove a local forward simulation of non-external states, then use the new properties of external calls to prove the local backward simulation of external states.

Even if the semantics of casting has not been fully implemented yet, it is already possible to compile some C programs with our modified version of CompCert. We can see that the new model successfully gives semantics to integer-pointer casting, but also allow optimizations when using logical blocks. This is illustrated Figure 8. We can see that constant propagation is done, and the program returns 0. This is explained by the fact that the memory block of `a` is logical, and thus the function `f()` has ownership of this memory block. The compiler deduces that `a` cannot be accessed by the function `g()`, and thus performs constant propagation. However, if the block is made concrete, then `f()` looses ownership of `a` and constant propagation should not be done. This is illustrated Figure 9, where the memory block of `a` has to be captured due to the pointer-to-integer cast.

```
extern void g();
int f() {
  int a = 0;
  int* q = &a;
  g();
  return a;
}
```

```
f() {
    6:  x4 = 0
    5:  int32[stack(0)] = x4
    4:  nop
    3:  x3 = "g"()
    2:  x2 = 0
    1:  return x2
    }
```

Fig. 8: Constant propagation on logical blocks

```
extern void g();
int f() {
  int a = 0;
  int* q = &a;
  int b = (int) q;
  g();
  return a;
}
```

```
f() {
    8:  x5 = 0
    7:  int32[stack(0)] = x5
    6:  x2 = stack(0) (int)
    5:  _ = builtin __capture(x2)
    4:  nop
    3:  x4 = "g"()
    2:  x3 = int32[stack(0)]
    1:  return x3
    }
```

Fig. 9: No Constant propagation on concrete blocks

8 Conclusion

We successfully implemented the new memory model in CompCert. We also changed many definitions (for instance abstract analysis or memory injection) to reflect the new properties of this model. We designed the correctness proof to deal with local non-determinism, and although the proof is not finished, we are confident that the remaining work should be very similar to what we've already done. Casting semantics have not been changed yet, but we believe it to be straightforward. We already added the *capture* function when needed. Currently, we modified or added more than 5 kloc of Coq in CompCert.

This work shows that the memory model introduced in [10] can be used in CompCert. We can verify that it allows optimizations when logical blocks are used. The model also allows semantics for any integer-pointer casts that are not too complicated for a programmer to keep in mind.

We also believe that the work on mixed simulations could be used in other contexts, when dealing with non-deterministic behaviors. As long as this non-determinism can be located, we can prove a mixed simulation where forward simulations wouldn't be possible, and still use it to construct the backward simulations needed for compiler correctness.

Future work should first focus on finishing the implementation.

9 Related Works

Writing programs with undefined behavior can lead to difficult-to-find bugs. For instance, some compilers can optimize code with the assumption that the program never encounters undefined behavior. This can result in produced code that does not behave as expected by the programmers [20].

A first solution to this issue is to identify the code whose optimization is based on undefined behaviors, as presented in [21].

However, a more common approach is to extend the semantics expressiveness of the C language, ruling out undefined behaviors. Many works have revolved around giving a formal semantics to the C language refining the informal semantics of the ISO standard. This is what [10] and this paper aim to achieve by refining the C semantics for pointer manipulation.

For the same purposes, some have investigated the use of a concrete memory model for C semantics [19][16]. More recently, [7] and [8] present the use of a new memory model using symbolic values. The idea is to use symbolic values instead of expressions to delay their evaluation. This successfully gives semantics to several C idioms: alignment constraints, bit-fields... However, as it is a deterministic semantics, programs that introduce non-deterministic behaviors due to allocation are still undefined.

In this work, we presented a new approach to deal with non-determinism in CompCert, and give semantics to every integer-pointer casts and pointer manipulation.

Another approach to deal with finite memories has been used in CompCertTSO [18], where all memory operations are done on a single finite logical block. Even if it uses a different memory model, the authors of [10] are confident that the quasi-concrete model could handle address threads like CompCertTSO. The quasi-concrete model could also be used with other works of semantics extension, such as the union types and strict aliasing of [11] or the universal pointer type of [12].

Acknowledgments

I am grateful to Professor Chung-Kil Hur for this internship. I am also grateful to Jeehoon Kang, who supervised my work, reviewed my code and helped me. I thank Juneyoung Lee for his work. I thank the Software Foundations Laboratory for being so welcoming.

References

1. Cerberus Survey v2. http://www.cl.cam.ac.uk/ pes20/cerberus/notes50-survey-discussion.html.
2. ISO/IEC 9899:1999. https://www.iso.org/standard/29237.html.
3. The CompCert C verified compiler, Documentation and users manual. http://compcert.inria.fr/man/.
4. The CompCert Project. http://compcert.inria.fr/.
5. The Coq Proof Assistant. https://coq.inria.fr/.
6. Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally verified optimizing compilation in ACG-based flight control software. In ERTS2 2012: Embedded Real Time Software and Systems, Toulouse, France, February 2012. AAAF, SEE.
7. Frédéric Besson, Sandrine Blazy, and Pierre Wilke. A Precise and Abstract Memory Model for C Using Symbolic Values. In 12th Asian Symposium on Programming Languages and Systems (APLAS 2014), volume 8858 of LNCS, pages 449 – 468, Singapore, Singapore, 2014. Springer.
8. Frédéric Besson, Sandrine Blazy, and Pierre Wilke. A concrete memory model for compcert. In Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings, pages 67–83, 2015.
9. Ben Greenman. CompCert Overview, 2015.
10. Jeehoon Kang, Chung-Kil Hur, William Mansky, Dmitri Garbuzov, Steve Zdancewic, and Viktor Vafeiadis. A formal C memory model supporting integer-pointer casts. In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, pages 326–335, 2015.
11. Robbert Krebbers. Aliasing restrictions of C11 formalized in coq. In Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings, pages 50–65, 2013.
12. Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. Formal C semantics: Compcert and the C standard. In Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings, pages 543–548, 2014.
13. Xavier Leroy. Formal verification of a realistic compiler. Commun. ACM, 52(7):107–115, 2009.
14. Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012.
15. Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. In Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015, pages 166–178, 2015.
16. Michael Norrish. C formalised in HOL. Technical report, 1998.
17. Peter W. O'Hearn. A primer on separation logic (and automatic program verification and analysis). In Software Safety and Security - Tools for Analysis and Verification, pages 286–318. 2012.
18. Jaroslav Sevcík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Compcerttso: A verified compiler for relaxed-memory concurrency. J. ACM, 60(3):22:1–22:50, 2013.

19. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007, pages 97–108, 2007.
20. Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: what happened to my code? In Asia-Pacific Workshop on Systems, APSys '12, Seoul, Republic of Korea, July 23-24, 2012, page 9, 2012.
21. Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013, pages 260–275, 2013.
22. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011, pages 283–294, 2011.

10 Appendix

10.1 Optimization examples

```
int main() {
  int a = 0;
  int b = a + 2;
  return a;
}
```

(a) Source

```
int main() {
  int a = 0;
  int b = 0 + 2;
  return 0;
}
```

(b) Target

Fig. 10: Example of Constant Propagation

```
int main() {
  int a = 0;
  int b = 0 + 2;
  return 0;
}
```

(a) Source

```
int main() {
  return 0;
}
```

(b) Target

Fig. 11: Example of Dead Allocation Elimination

```
int main() {
  int a = 0;
  int * p;
  *p = 1;
  a = *p;
  return 0;
}
```

(a) Source

```
int main() {
  int a = 0;
  int * p;
  return 0;
}
```

(b) Target

Fig. 12: Example of Dead Store and Load Elimination

```
foo(ptr p) {
  var ptr q, int a;
  q = malloc(1);
  *q = 123;
  bar(p);
  a = *q;
  *p = a;
}
```

(a) Source

```
foo(ptr p) {

                //DAE
                //DSE
  bar(p);
                //DLE
  *p = 123; //CP
}
```

(b) Target

Fig. 13: Example with multiple optimizations, taken from [10]

10.2 Memory implementation

The new definition of the memory is the following:

```
Record mem' : Type := mkmem {
mem_contents: PMap.t (ZMap.t memval);  (**r [block -> offset -> memval] *)
mem_access: PMap.t (Z -> perm_kind -> option permission);
(**r [block -> offset -> kind -> option permission] *)
mem_concrete: PMap.t (option Z); (** [block -> option Z] **)
mem_offset_bounds : PMap.t (Z*Z); (** [block -> Z * Z ] **)
nextblock: block;
access_max:
  forall b ofs, perm_order'' (mem_access#b ofs Max) (mem_access#b ofs Cur);
nextblock_noaccess:
  forall b ofs k, ~(Plt b nextblock) -> mem_access#b ofs k = None;
contents_default:
  forall b, fst mem_contents#b = Undef;
nextblocks_logical:
  forall b, ~(Plt b nextblock) -> mem_concrete#b = None;
addresses_in_range:
  forall bo addr (IN_BLOCK: addr_in_block mem_concrete mem_offset_bounds
      mem_access addr bo),
    in_range addr (1,max_address);
no_concrete_overlap:
  forall addr, uniqueness (addr_in_block mem_concrete mem_offset_bounds
      mem_access addr);
}.
```

10.3 Memory injection

The new definition of memory injection is the following:

```
Record inject' (f: meminj) (m1 m2: mem) : Prop :=
mk_inject {
  mi_inj:
```

```
    mem_inj f m1 m2;
  mi_freeblocks:
    forall b, ~(valid_block m1 b) -> f b = None;
  mi_mappedblocks:
    forall b b' delta, f b = Some(b', delta) -> valid_block m2 b';
  mi_no_overlap:
    meminj_no_overlap f m1;
  mi_representable:
    forall b b' delta ofs,
    f b = Some(b', delta) ->
    perm m1 b (Ptrofs.unsigned ofs) Max Nonempty \/
      perm m1 b (Ptrofs.unsigned ofs - 1) Max Nonempty ->
    delta >= 0 /\ 0 <= Ptrofs.unsigned ofs + delta <= Ptrofs.max_unsigned;
  mi_perm_inv:
    forall b1 ofs b2 delta k p,
    f b1 = Some(b2, delta) ->
    perm m2 b2 (ofs + delta) k p ->
    perm m1 b1 ofs k p \/ ~perm m1 b1 ofs Max Nonempty
}.
```

10.4 The previous correctness proof

Theorem: Factor forward simulation If there is a forward simulation between S_1 and S_2 and S_2 has *single events*, then there is a forward simulation between $Atomic(S_1)$ and S_2.

Correctness proof The previous proof in CompCert used the following reasoning:

1. Prove a forward simulation for each pass between CStrategy and ASM. Prove a backward simulation between CompCert C and Cstrategy.
2. Use the simulation composition theorem to deduce a forward simulation between CStrategy and ASM
3. Use the Factor forward simulation theorem to deduce a forward simulation between *Atomic*(CStrategy) and ASM.
4. Use the forward to backward theorem to deduce a backward simulation between *Atomic*(CStrategy) and ASM, as well as the factor backward theorem to deduce a backward simulation between CompCert C and *Atomic*(CStrategy).
5. Use the simulation composition theorem to deduce a backward simulation between CompCert C and ASM.

This proof is illustrated Figure 14.

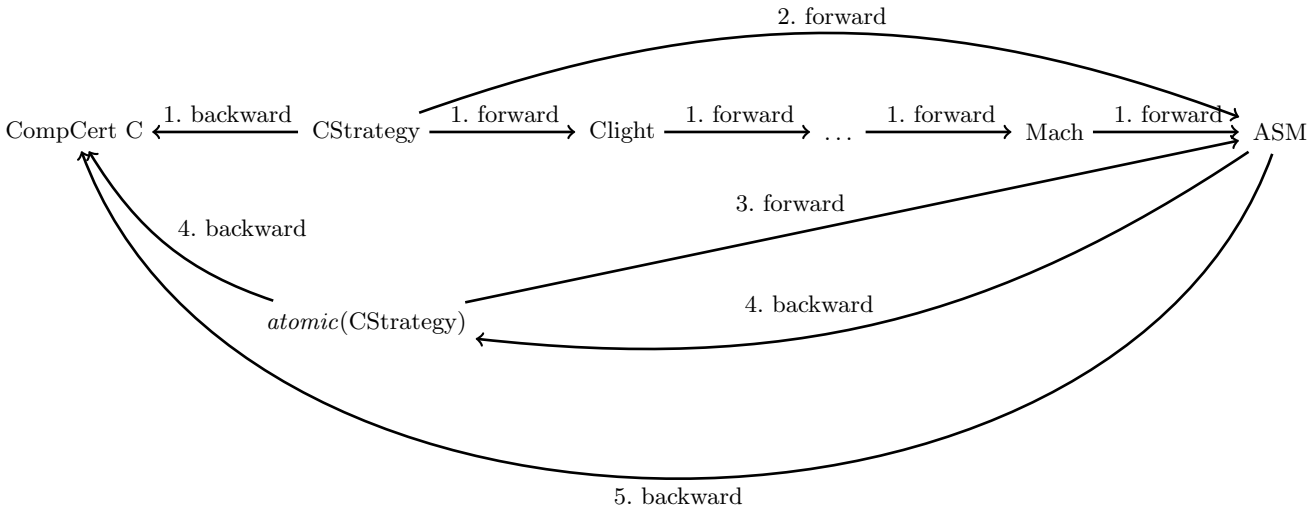


Fig. 14: The previous correctness proof