A Formal C Memory Model Supporting Integer-Pointer Casts

Jeehoon Kang
Seoul National University, South Korea
jeehoon.kang@sf.snu.ac.kr

Chung-Kil Hur *
Seoul National University, South Korea
gil.hur@sf.snu.ac.kr

William Mansky
University of Pennsylvania, USA
wmansky@seas.upenn.edu

Dmitri Garbuzov
University of Pennsylvania, USA
dmitri@seas.upenn.edu

Steve Zdancewic
University of Pennsylvania, USA
stevez@cis.upenn.edu

Viktor Vafeiadis
Max Planck Institute for Software Systems (MPI-SWS), Germany
viktor@mpi-sws.org

Abstract

The ISO C standard does not specify the semantics of many valid programs that use non-portable idioms such as integer-pointer casts. Recent efforts at formal definitions and verified implementation of the C language inherit this feature. By adopting high-level abstract memory models, they validate common optimizations. On the other hand, this prevents reasoning about much low-level code relying on the behavior of common implementations, where formal verification has many applications.

We present the first formal memory model that allows many common optimizations and *fully* supports operations on the representation of pointers. All arithmetic operations are well-defined for pointers that have been cast to integers. Crucially, our model is also simple to understand and program with. All our results are fully formalized in Coq.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords C Memory Model, Integer-Pointer Cast, Compiler, Optimization, Verification

1. Introduction

The ISO C standard [5] famously does not give semantics to a significant subset of syntactically valid C programs. Instead, many programs exhibit implementation-defined, unspecified, or undefined behavior, with the latter imposing no requirements on a conforming implementation. This has led to the somewhat controversial practice of sophisticated C compilers reasoning backwards from instances of undefined behavior to conclude that, for example, certain code paths must be dead. Such transformations can lead to surprising non-local changes in program behavior and difficult-to-find bugs [13, 14].

Accordingly, there have been numerous efforts to capture the subtleties of the C standard formally, either by giving an alternative language definition or a conforming implementation [2, 9, 11].

The C memory model has been of particular interest: cross-platform low-level access to memory is a defining feature of C-family languages and is essential for applications such as operating system kernels and language runtimes. However, subtle pointer aliasing rules [6], reliance on implementation-specific behavior, and the treatment of pointers to uninitialized memory makes reasoning about even single-threaded programs non-trivial.

One popular extension of the standard C memory model that has not previously been formalized is the *unrestricted* manipulation of pointers as integer values. While the language definition provides an integer type `uintptr_t` that may be legally cast to and from pointer types, it does not require anything of the resulting values [5, §7.20.1.4p1]. Nevertheless, there are many important use cases for manipulating the representation of pointers in low-level code.

For example, casting pointers to integers is widely used in the Linux kernel and JVM implementations to perform bitwise operations on pointers. Another common usage pattern occurs in the C++ standard library (`std::hash`), where the pointer's bit representation is used as a key for indexing into a hash table. This is useful since taking a pointer is a cheap way to get a unique key.

The most straightforward way to support bit-level pointer manipulation is to adopt what is often called a *concrete* memory model. This approach most closely resembles what the machine is actually doing: pointers have the same representation as integer values of the appropriate width, and they simply index into a single flat array representing memory.

However, the combination of finite memory and allowing casts of arbitrary integers to pointers invalidates many basic compiler optimizations. Consider, for example, a function `f` that initializes a local variable `a` and then calls some unknown external function `g`. We might expect the compiler to deduce that the value of `a` is unaffected by the call to `g` and perform constant propagation:

```
                  extern void g();

int f(void) {      int f(void) {      int f(void) {
  int a = 0;         int a = 0;
  g();         →     g();         →     g();
  return a;          return 0;          return 0;
}                  }                  }
```

* corresponding author

PLDI'15, June 13–17, 2015, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3468-6/15/06…$15.00.
http://dx.doi.org/10.1145/2737924.2738005

Using a simple concrete memory model, however, we have to consider the possibility that `g` is able to "guess" the location of `a` in memory and changes its value. This can happen if, for instance, our semantics allocates memory deterministically and the caller of `f` sets up the state of memory appropriately for `g`.

The compiler may further optimize the program by completely removing the now unused local variable `a`. This latter transformation is again disallowed by concrete models because it might change the behavior of the program: by virtue of there being one fewer memory cells allocated, the call to `g` might succeed where initially it exhausted memory.

In order to enable such compiler optimizations, most work on verified compilation instead relies on *logical* memory models. These models represent pointers as pairs of an allocation block identifier and an offset within that block, where typically the set of valid allocation block identifiers is infinite. In the above example, the first optimization is allowed since there is no way to forge the logical address of the variable `a` from those of other blocks. Also, the second one is allowed since the memory is infinitely large.

Logical models allow most compiler optimizations, but cannot support many low-level C programming idioms using casts between pointers and integers, treating programs containing them as undefined (*i.e.,* erroneous).

In this paper, we propose a *quasi-concrete* memory model for C/C++ that combines the strengths of the aforementioned approaches. It gives semantics to programs that manipulate the bit-level representation of pointers, yet permits the same optimizations as logical models for code not using these low-level features. Crucially, we achieve this without substantially complicating the proof techniques required for a verified compiler while retaining a model that is simple for the programmer to reason about.

The key technical ingredient for making this work is having two distinct representations of pointer values, a concrete and a logical one, and a process for converting between the two. By default, a pointer is represented logically, and only when it is cast to an integer type, is the logical pointer value *realized* to a concrete 32-bit (or 64-bit) integer. When an integer is cast (back) to a pointer value, it is mapped to the corresponding logical address.

The quasi-concrete model conservatively extends the logical model. It gives semantics to strictly more programs than those supported by the logical model without changing the semantics of those programs that do have semantics under the logical model. Thus, any sound reasoning about programs in the logical model also holds in the quasi-concrete model, but the quasi-concrete model also supports reasoning about pointer arithmetic as in the concrete model.

Finally, the quasi-concrete model is not intended to replace the memory model in the C standard. Like the concrete and logical models, it is a formal refinement of the (informal) C standard that can be used for formally reasoning about programs and program transformations (as in compiler verification).

Our contributions are:

- The first formal memory model that *fully* supports integer-pointer casts and yet allows the standard compiler optimizations.
- A technique for proving program equivalence under our memory model, and its application to verify a number of standard optimizations that are difficult to verify in the presence of integer-pointer casts.

All the proofs reported in this paper have been fully formalized in Coq and can be found in the following project webpage.

http://sf.snu.ac.kr/intptrcast

2. Technical Background

In this section, we introduce a minimal C-like programming language and review the concrete memory model (§2.1) as well as CompCert's logical model (§2.2), because our quasi-concrete model builds on them. We then review behavioral refinement (§2.3), a key definition used for proving compiler optimizations correct.

To simplify the presentation, in this paper, we focus on handling integer-pointer casts and do not discuss many of the orthogonal aspects of C memory models. Specifically:

- We assume a 32-bit architecture: words are 4 bytes wide, and the size of the address space is 2^{32}.
- We consider only integer and pointer values, and omit values of other types such as `float` or `char`.
- We also omit subword arithmetic, and assume each address stores a 32-bit value.
- We do not consider concurrency.

Except for concurrency, the other simplifications we made can easily be lifted.

For concreteness, we consider the following simple C-like language:

$$\begin{array}{lll}
\mathit{Typ} & ::= & \texttt{int} \mid \texttt{ptr} \\
\mathit{Bop} & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{\&\&} \mid \texttt{=} \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Var} \mid \mathit{Global} \mid \mathit{Exp}\ \mathit{Bop}\ \mathit{Exp} \\
\mathit{RExp} & ::= & \mathit{Exp} \mid \texttt{malloc}(\mathit{Exp}) \mid \texttt{free}(\mathit{Exp}) \mid \texttt{(}\mathit{Typ}\texttt{)}\ \mathit{Exp} \\
 & & \mid \texttt{input()} \mid \texttt{output}(\mathit{Exp}) \\
\mathit{Instr} & ::= & \mathit{Fid}(\mathit{Exp}, \ldots, \mathit{Exp}); \mid \mathit{Var}\ \texttt{=}\ \mathit{RExp} \mid \mathit{Var}\ \texttt{=}\ \texttt{*}\mathit{Exp} \\
 & & \mid \texttt{*}\mathit{Exp}\ \texttt{=}\ \mathit{Exp} \mid \texttt{if (}\mathit{Exp}\texttt{)}\ \overline{\mathit{Instr}}\ \texttt{else}\ \overline{\mathit{Instr}} \\
 & & \mid \texttt{while (}\mathit{Exp}\texttt{)}\ \overline{\mathit{Instr}} \\
\mathit{Decl} & ::= & \mathit{Fid}(\mathit{Typ}\ \mathit{Var}, \ldots, \mathit{Typ}\ \mathit{Var}) \\
 & & \{\texttt{var}\ \overline{\mathit{Typ}\ \mathrm{Var};}\ \overline{\mathit{Instr}}\}
\end{array}$$

The `input` and `output` operations produce externally visible events; all other operations are intended to model the corresponding operations in C. By $\overline{T}$ we mean a list of T. For simplicity, we omit `return` instructions and instead return values via pointer-valued arguments to functions.

2.1 Concrete Model

Concrete memory consists of a 2^{32}-sized array of values, and a list of allocated blocks, represented as pairs (p, n) of the block's starting address and its size. Loading from or storing to an unallocated address raises an error (*i.e.,* undefined behavior). Values are just 32-bit integers, since pointers are merely integers in the concrete model. As a result, *the concrete model natively supports integer-pointer casts*.

$$\begin{array}{ll}
\mathrm{Mem} & \overset{\text{def}}{=} (\texttt{int32} \to \mathrm{Val}) \times \texttt{list}\ \mathrm{Alloc} \\
\mathrm{Alloc} & \overset{\text{def}}{=} \{\, (p, n) \mid p \in \texttt{int32} \wedge n \in \texttt{int32} \,\} \\
\mathrm{Val} & \overset{\text{def}}{=} \{\, i \in \texttt{int32} \,\}
\end{array}$$

Memory allocation inserts a block into the list of allocated blocks, whereas deallocation removes one. Overall, the list of allocated blocks should be *consistent*:[1]

- If (p, n) is allocated, then $\emptyset \neq [p, p+n) \subseteq (0, 2^{32}-1)$.
- If blocks (p_1, n_1) and (p_2, n_2) are distinct allocations, their ranges $[p_1, p_1+n_1)$ and $[p_2, p_2+n_2)$ are disjoint.

[1] These are a subset of `malloc`'s properties according to the C11 Standard [5]. For more details, see §7.22.3 paragraph 1 and §6.5.8 paragraph 5.

As we have seen, however, the concrete model *does not support standard compiler optimizations* such as constant propagation and dead allocation elimination in the presence of external function calls. This is because the model does not provide a mechanism for ensuring that a module has exclusive control over some part of memory, thereby assuming that unknown code can read and update the contents of every allocated memory cell.

2.2 CompCert's Logical Model

In CompCert's logical model [9, 10], memory consists of a finite set of logical blocks with unique block identifiers. Each block is a fixed-sized array of values together with a validity flag v that indicates whether the block is accessible or has been freed. As before, accessing a freed block raises an error. Values are either 32-bit integers or logical addresses. Here, a logical address (l, i) consists of a block identifier l and an offset i inside the block.

$$\begin{aligned}
\text{Mem} &\overset{\text{def}}{=} \text{BlockID} \rightharpoonup_{\text{fin}} \text{Block} \\
\text{Block} &\overset{\text{def}}{=} \{\, (v, n, c) \mid v \in \texttt{bool} \land n \in \mathbb{N} \land c \in \text{Val}^n \,\} \\
\text{Val} &\overset{\text{def}}{=} \{\, i \in \texttt{int32} \,\} \uplus \{\, (l, i) \in \text{BlockID} \times \texttt{int32} \,\}
\end{aligned}$$

An important advantage of the logical model over the concrete one is that it *allows functions to have exclusive control over a logical block* as long as they do not allow its address to escape. The reason is that it is not possible to manufacture the logical address of an already allocated block. This property guarantees the correctness of many useful optimizations, such as constant propagation across function calls and dead allocation elimination.

A secondary advantage is that programs have *infinite memory*, rendering their allocation behavior unobservable, which in turn makes it easy for compilers to remove dead allocations.

Apart from that logical models have a slightly more complicated semantics, their main disadvantage is that *they do not support integer-to-pointer casts* very well. As a result, CompCert has very weak support for integer-pointer casts. Generally, they are treated as `nops` (*i.e.,* the identity function) rather than undefined (*i.e.,* erroneous), and thus variables of integer (or pointer) types can contain both integers and logical addresses. In CompCert's higher-level languages (CompCert C and Clight), once a pointer is cast into an integer, any arithmetic on that integer returns an unspecified value. In its lower-level languages (Cminor, RTL, *etc.*), however, some of the integer operations (namely, addition, subtraction, and equality tests) are also well-defined for pointer values in special cases. More specifically, for instance, the addition of an integer to an address, the subtraction between addresses when they are in the same logical block, and the equality test between an address and `NULL` are well-defined.

2.3 Behavioral Refinement

The standard notion of compiler correctness is behavioral refinement, which states that *the set of target program behaviors must be a subset of the set of source program behaviors*. We consider sets of behaviors as opposed to single behaviors because in general a program may have multiple behaviors due to non-determinism.

Given a set of I/O events that the program may generate, a behavior is one of the following three forms:

1. A terminating execution producing a finite sequence of I/O events, $e_1, \cdots, e_n,$ `term`
2. A diverging execution that has produced only a finite sequence of I/O events, $e_1, \cdots, e_n,$ `nonterm`.
3. A diverging execution producing an infinite sequence of I/O events, $e_1, \cdots, e_n, \cdots$.

We regard *the undefined behavior as described in the C11 standard as the set of all behaviors*. This captures the intuitive properties of compilers on undefined behavior: if the target program's behavior is undefined, then the source program's behavior is also undefined. If the source program's behavior is undefined, then compiler can choose any program as its result.

Then what is the set of behaviors of out of memory?[2] Intuitively, we should allow the target program to run out of memory even if the source program does not, because important compiler transformations such as register allocation may increase the program's memory requirements. Dually, since running out of memory is not that serious a problem for programmers as is genuine undefined behavior (*e.g.,* accessing freed memory), the target behavior of a source program running out of memory should not be arbitrary, but should match the source behavior and also run out of memory.

Therefore, CompCertTSO [12] models *out of memory as the empty set of behaviors*, that is no behavior. However, what happens if there were I/O events prior to the out-of-memory error? Discarding I/O events before running out of memory is absurd: the target program should always perform a prefix of the events the source program could have performed. To handle this, CompCert-TSO also observes *partial behaviors*, possibly before discarding behavior due to running out of memory:

4. A partial execution of the program that has produced a finite sequence of I/O events, $e_1, \cdots, e_n,$ `partial`.

As before, refinement is defined as inclusion of the set of (possibly partial) behaviors of the target program into that of the source program.

In this paper, we follow CompCertTSO's approach of handling out of memory. However, unlike CompCertTSO, where only the target language can run out of memory, our source language can also run out of memory due to pointer-to-integer casts, as we explain below.

3. The Quasi-Concrete Model

Our quasi-concrete model is simply a hybrid of the fully concrete model and the fully logical model. However, there are several issues with how to combine the two models to minimize their disadvantages. In this section, we introduce the quasi-concrete model and discuss how we address the design issues at a high level. More detailed discussions with concrete examples will follow in the subsequent sections. All the optimization examples presented in this section are performed by `clang -O2`.

3.1 Memory Representation

Our quasi-concrete model slightly generalizes the logical model to allow both concrete blocks (as in the concrete model) and logical blocks (as in the logical model). For this, we add one more attribute p to a logical block, which is either *undefined* or a concrete address. The attribute p indicates whether the block is logical (when p is undefined) or it is a concrete block starting at the address p (when p is defined).

$$\begin{aligned}
\text{Block} \overset{\text{def}}{=} \{\, (v, p, n, c) \mid\, & p \in \texttt{int32} \uplus \{\text{undef}\} \\
& \land v \in \texttt{bool} \land n \in \mathbb{N} \land c \in \text{Val}^n \,\}
\end{aligned}$$

We say that an address (l, i) is concrete when the block l is a concrete block starting at an address p. In this case, the address (l, i) can be cast to the integer $p + i$ and vice versa (see §4 for details).

[2] In C11 Standard [5], `malloc` returns 0 in case of out-of-memory [§7.22.3 paragraph 1]. However, this semantics does not justify dead allocation eliminations. The reason is that, for example, if a dead malloc is eliminated, then the source may have more allocated blocks than the target has and thus a subsequent allocation may return 0 in the source but return a valid pointer in the target.

```
a = (a - b) + (2 * b - b);
q = (ptr) a;                       →     q = (ptr) a;
*q = 123;                                *q = 123;
```

Figure 1. Arithmetic Optimization Example I

As in the concrete model, the list of allocated (*i.e.,* valid) blocks with concrete addresses must be consistent: they should not include 0 or the maximum address, and their ranges should be disjoint. Logical blocks have no such requirement, since they are non-overlapping by construction.

In the subsequent sections, we discuss several issues that arose during the design of our quasi-concrete model and justify our solutions to the issues.

3.2 Combining Logical and Concrete Blocks

Our quasi-concrete model is a hybrid model that allows both concrete and logical blocks to coexist. Though we allow both, concrete and logical blocks still have their own disadvantages: concrete ones do not provide exclusive ownership and logical ones do not allow casting to integers.

Thus, one may naturally ask why we do not develop a new notion of block that has the advantages of both concrete and logical blocks. Our answer is that such a model would not justify other important optimizations such as simplification of integer operations, while our quasi-concrete model justifies them.

For instance, consider a model in which some blocks have both concrete addresses and some extra permission information, so that we can tell when a block is exclusively owned. In such a model, we would like to know that we do not lose permission information when a pointer is cast to an integer, even if integer operations are performed on it (*e.g.,* `base64_encoding` a pointer and then `base64_decoding` it).

However, this prevents the optimization presented in Figure 1. Suppose the variable `b` contains an integer with permission to access some valid block l, and `a` contains an integer without any permission that is equal to the concrete address of the block l. Then the source program successfully stores 123 into the block l because `q` has the relevant permission, whereas the target program fails because `q` does not have the permission. See §6.1 for how to verify this optimization in our quasi-concrete model.

Furthermore, while our quasi-concrete model does disallow some optimizations based on exclusive ownership (since once a block has become concrete it is non-exclusive for the remainder of the execution), we expect that our model would not lose many optimization opportunities in practice. This is because exclusively owned blocks are mostly local or temporary ones, so that their concrete addresses are unlikely to be used by integer operations. See §3.7 for further details.

3.3 Choosing Concrete Blocks

As discussed in §2.1, using concrete addresses for memory locations provides no guarantees of ownership, and thus prevents certain optimizations. In the worst case, one function could guess the concrete address of a supposedly-private resource of another function, and then forge a pointer to that address and modify it. In order to maximize the range of optimizations that can be performed, a hybrid model should assign concrete addresses to as few blocks as possible.

The natural choice, then, is to make concrete only those blocks whose concrete addresses are really used in some operation. If we perform some computation with the value of a pointer that only makes sense when that value is an integer (*e.g.,* comparing it with an integer value) then the target of that pointer must have a real address. In all other cases, even if the address of the block is taken, we could conceivably use a logical value and maintain the ownership guarantees of the logical model. However, this approach has a serious problem: it does not justify some important integer optimizations, such as a dead code elimination presented in Figure 2.

```
foo(int a) {                foo(int a) {
  a = a & 123;                a = a & 123;
  // return a;                // return a;
}                     →     }
...                         ...
a = (int) p;                a = (int) p;
foo(a);
bar();                      bar();
```

Figure 2. Dead Code Elimination Example

```
p = malloc (1);                 p = malloc (1);
*p = 123;                       *p = 123;
bar();                 →        bar();
a = *p;                         a = *p;
hash_put(h, p, a);              hash_put(h, p, 123);
```

Figure 3. Ownership Transfer Example

Suppose the pointer `p` contains a logical block l. In the source program, since its concrete address is used in the function `foo`, the block l must be given a concrete address. In the target, the read-only call to `foo` is optimized away, and the block l may not be given a concrete address. That is, the source may have more concrete blocks than the target. Thus, if `bar()` accesses an arbitrary concrete memory location, then that access might succeed in the source but fail in the target. Since a failure is possible in the target that did not exist in the source, the optimization has introduced new behavior, and is invalid.

Instead, we make concrete those blocks whose addresses are cast to integers, even if the cast integers are not used in any operation. This gives us a simple way to determine which blocks should be made concrete, and avoids making integer operations memory-relevant (see §6.2 for how to verify this example in the quasi-concrete model). In practice, this choice also allows most of the optimizations that would be performed in the minimally-concrete model (see §3.7 for details).

3.4 Assigning Concrete Addresses

Once we know which blocks will need concrete addresses, we need to decide when during a program's execution to assign those addresses.

One approach would be to make such a decision as early as possible (*i.e.,* at allocation time). We allocate blocks as either logical or concrete, and cause concrete operations (namely integer casts) on logical blocks to raise out-of-memory-type behavior (*i.e.,* no behavior). Since it is difficult to determine whether a block will need a concrete address, we would need to choose the kind of block to allocate non-deterministically. However, this would add unintuitive failures to our model, effectively allowing out-of-memory-type behavior when the allocator chooses the wrong kind of block even when concrete blocks are available.

Our solution is to instead allocate all blocks as logical blocks, and assign concrete addresses to logical blocks at casting time. This casting can result in out of memory only when there is not enough free concrete space. By waiting to make blocks concrete until we reach the casting point, we can remove the non-determinism about whether the blocks are concrete or logical. That is, blocks are always logical until the first casting point, and concrete afterward.

```
                                      t = a + b;
d1 = a + (b - c1);        →          d1 = t - c1;
d2 = a + (b - c2);                   d2 = t - c2;
```

Figure 4. Arithmetic Optimization Example II

This also allows *ownership transfer* optimizations such as the constant propagation example in Figure 3, in which pointers are privately owned up until some point and then become publicly available. In this example, the allocated block is initially logical and becomes concrete when cast to an integer (possibly in the call to hash_put). At this point, the ownership of the block is transferred from private to public. Since a is treated as logical up until the call to hash_put, we can perform constant propagation as normal before the call. (For formal details, see §6.3.)

On the other hand, the above model with non-deterministic allocation does not allow such optimizations. The reason why this optimization is not allowed in the above model is as follows. When the allocated block in the target is concrete, the corresponding allocation in the source must be concrete; otherwise, when the function hash_put casts the address of the block to an integer the source program raises no behavior, while the target succeeds. Thus, you lose the ownership over the block and cannot justify the constant propagation due to the presence of bar().

However, note that this may not be a decisive example, since this optimization is not used in all real-world compilers (it is performed by clang -O2 and higher, but not by gcc -O2 or higher).

3.5 Operations on Pointers

In §3.3, we explained that the fewer blocks we make concrete, the more we can take advantage of the ownership guarantees provided by logical blocks. We have seen that operations that require pointers to have integer values force a lower bound on the number of blocks that must be made concrete. As such, we can improve our model by reducing the number of operations that require concrete addresses. We take our cue from CompCert's memory model, in which several arithmetic operations, such as integer-pointer addition and subtraction of pointers from pointers in the same block, are well-defined even in the absence of a concrete address (see §2.2).

One disadvantage of CompCert's approach to arithmetic operations is that it invalidates some important arithmetic optimizations, such as the optimization presented in Figure 4, by introducing logical addresses as possible values for integer-typed variables. To see this, suppose that the integer variables a, b, c1, and c2 contain the same logical address $(l, 0)$. The source program shown will successfully assign $(l, 0)$ to the variables d1 and d2, because b - c1 and b - c2 evaluate to 0. However, in the target, the variable t gets an unspecified value, because the addition of two logical addresses is undefined. Thus, the target has more behaviors than the source, and the optimization is invalid.

We avoid this disadvantage through the use of type checking. As in the LLVM IR, we use types to ensure that integer variables contain only integer values. This allows us to justify the full range of arithmetic optimizations on integer variables (see §6.4 for details), while also giving semantics to the operations on logical addresses when possible. See §4 for an example of type-dependent semantics of arithmetic operations in our model.

3.6 Dead Cast Elimination

As a result of our design decisions thus far, casts have become important effectful operations in our model, determining the points at which logical blocks are given concrete addresses. This leads to a potential problem with dead code elimination. Since casting a pointer to an integer has a side effect in memory, removing

```
foo(ptr p, int n) {              foo(ptr p, int n) {
  var ptr q, int a, r;             var ptr q, int a, r;
  q = malloc (n);                  q = malloc (n);
  a = (int) p;            →        a = (int) p;
  r = a * 123;                     r = a * 123;
  // return r;                     // return r;
}                                }
...                              ...
foo(p, n);                →
bar();                           bar();
```

Figure 5. Dead Cast Elimination Example

dead cast operations is not obviously justified in the quasi-concrete model.

However, in fact, we can solve this problem and support dead cast elimination in our framework. The solution stems from our model's place in a broader compilation framework. We expect the quasi-concrete model to be used for mid-level intermediate representations in a compiler, while the back-end low-level language will use a fully concrete model. In the quasi-concrete model, casting a pointer to an integer has a side effect on memory, and we cannot eliminate cast operations. In the fully concrete model, however, a cast from pointer to integer is a no-op, and such casts can always be eliminated. Thus, we can perform dead-cast-elimination optimizations in the backend.

However, we still have a problem when dead cast is combined with dead allocation. In the concrete model allocations of dead blocks cannot be removed, because otherwise an arbitrary access in the source may succeed but fail in the target.

Our solution to this problem is to remove dead casts combined with dead blocks during the translation from the quasi-concrete to the fully concrete model. For instance, consider the dead call elimination optimization presented in Figure 5. This optimization is not valid when both the source and the target use the quasi-concrete model, due to the cast operation in the function. It is also not valid when both the source and the target use the concrete memory, due to the allocation in the function. However, the optimization is valid when the source uses the quasi-concrete model and the target uses the fully concrete model (see §6.5 for formal details).

Although our solution does not justify the removal of all dead casts, it should cover most of them in practice (see §3.7).

3.7 Drawbacks of the Quasi-Concrete Model

Although our quasi-concrete model is designed to allow as many optimizations as possible, it still disallows some reasonable optimizations. In particular, if a function newly allocates a block and casts its address to an integer, then it loses the ownership guarantees on the block. Even if the block is still effectively locally owned, once its address is cast to an integer, we can no longer perform optimizations that rely on its locality, such as dead code elimination or constant propagation.

However, we think that blocks whose addresses are cast to integers in actual programs are unlikely to be completely local. There are few reasons to cast a local pointer to an integer unless the address will be shared with other code sections.

For instance, consider the following example of a simple program in which we might want to perform a locality optimization even after casting a pointer to an integer. The program is a variant of the example in §3.6, in which we cast q into an integer instead of p, so that in our model the local block becomes concrete and cannot be eliminated. Although this optimization cannot be performed in our framework, the function foo is nothing but an unpredictable number generator, and is unlikely to occur in real programs.

```
foo(ptr p, int n) {            foo(ptr p, int n) {
  var ptr q, int a, r;           var ptr q, int a, r;
  q = malloc (n);                q = malloc (n);
  a = (int) q;           →       a = (int) q;
  r = a * 123;                   r = a * 123;
  // return r;                   // return r;
}                              }
...                            ...
foo(p, n);               →
bar();                         bar();
```

Another, more reasonable limitation of our model occurs when one privately uses a local block for some time, then casts its address to an integer and releases it to the public (*e.g.,* by using the integer as a key for hash table). Consider the following example, which is a variant of the example in §3.4, where we cast the address of a newly allocated block into an integer and use the integer as a key for hash table:

```
 ...                           ...
 p = malloc (1);               p = malloc (1);
 *p = 123;                     *p = 123;
 b = (int) p;          →       b = (int) p;
 bar();                        bar();
 a = *p;                       a = *p;
 hash_put(h, b, a);            hash_put(h, b, 123);
 ...                           ...
```

This constant propagation optimization is invalid in the quasi-concrete model because the newly allocated block is cast to an integer before the call to `bar`. (It becomes valid if the cast is moved after the call to `bar`, though.) However, while ownership transfer optimizations of this sort are indeed performed by `clang -O2`, they are not performed by `gcc -O2` or higher, and can be viewed as minor optimizations that are not often used.

4. Language Semantics

This section describes how to use the ideas of the quasi-concrete memory model to give semantics to the C-like language of §2.

NULL pointer We represent the NULL pointer as the logical address $(0,0)$ and initialize the block 0 as follows:

$$m(0) = (v,p,n,c) \text{ with } v = \texttt{true}, p = 0, n = 1.$$

The only special treatment of the block 0 is that we (i) raise undefined behavior when accessing it via a load or a store; and (ii) do nothing when freeing it (because `free(0)` is allowed in C).

Casting between Integers and Pointers We first define casting between integers and logical addresses via *reification* and *validity checking*. The reification function $\downarrow_m$ under memory m converts a logical address to a corresponding integer if its block in memory has a concrete address. The validity predicate valid_m checks if a logical address is inside the range of a valid block.

$$(l,i)\downarrow_m \overset{\text{def}}{=} p+i \quad \text{if } m(l) = (v,p,n,c) \land p \text{ is defined}$$

$$\text{valid}_m(l,i) \text{ iff } m(l) = (v,p,n,c) \land v = \text{true} \land (0 \le i < n)$$

Casting a logical address (l,i) into an integer first realizes the block l and then reifies the address (l,i) if it is valid; otherwise, raises undefined behavior. Casting an integer i yields a valid logical address (l,j) that is reified to i if there is such an address; otherwise, raises undefined behavior. Note that an integer is cast to a unique address if it succeeds.

$$\begin{array}{lll}
\texttt{cast2int}_m(l,i) \overset{\text{def}}{=} (l,i)\downarrow_m & \text{if } \text{valid}_m(l,i) & \{\text{after realizing } l\} \\
\texttt{cast2ptr}_m(i) \overset{\text{def}}{=} (l,j) & \text{if } \text{valid}_m(l,j) \land (l,j)\downarrow_m = i &
\end{array}$$

Computing with Logical Values We now give semantics to the binary operations based on the static types of their operands. When both operands are of type `int`, we perform ordinary integer addition, subtraction, etc. When one or more arguments are of type `ptr`, we give the operations special semantics for the well-defined cases and raise undefined behavior otherwise:

$$\begin{array}{lll}
(\texttt{p + a}, m) \Downarrow (l, i_1 + i_2) & \text{if } \texttt{p} = (l,i_1) \land \texttt{a} = i_2 \\
(\texttt{a + p}, m) \Downarrow (l, i_1 + i_2) & \text{if } \texttt{a} = i_1 \land \texttt{p} = (l,i_2) \\
(\texttt{p - a}, m) \Downarrow (l, i_1 - i_2) & \text{if } \texttt{p} = (l,i_1) \land \texttt{a} = i_2 \\
(\texttt{p}_1\texttt{ - p}_2, m) \Downarrow i_1 - i_2 & \text{if } \texttt{p}_1 = (l,i_1) \land \texttt{p}_2 = (l,i_2) \\
(\texttt{p}_1 = \texttt{p}_2, m) \Downarrow i_1 = i_2 & \text{if } \texttt{p}_1 = (l,i_1) \land \texttt{p}_2 = (l,i_2) \\
(\texttt{p}_1 = \texttt{p}_2, m) \Downarrow \texttt{false} & \text{if } \texttt{p}_1 = (l_1,i_1) \land \texttt{p}_2 = (l_2,i_2) \land l_1 \neq l_2 \\
 & \quad \land \text{valid}_m(l_1,i_1) \land \text{valid}_m(l_2,i_2)
\end{array}$$

This definition of equality is a refinement of the pointer equality given in the ISO C standard [5]; for instance, it allows us to conclude that $\texttt{p} = \texttt{p}$ even when `p` is not a pointer to an allocated block, while in the C standard the result of this comparison is undefined.

Quasi-Concrete and Concrete Semantics Using these definitions, we can give the usual operational semantic definitions to our language constructs, and perform memory operations (loads, stores, allocations, and casts) in the quasi-concrete memory model. Static type checking allows us to split variables into pointer-typed variables (whose values are always logical addresses and treated as described above) and integer-typed variables (whose values are always ordinary integers and require no special handling).

We also give the language a purely concrete semantics and use it as a low-level intermediate language with the concrete memory. In this semantics, all memory blocks are realized and all values are just integers, interpreted as either integer values or physical addresses of memory cells.

5. Reasoning Principles

In this section, we give a high-level overview of our reasoning principles with a running example.

5.1 Running Example & Informal Verification

Consider the following example transformation, which is indeed performed by "clang -O2". This transformation involves four different optimizations: constant propagation (CP), dead load elimination (DLE), dead store elimination (DSE), and dead allocation elimination (DAE).

```
      foo(ptr p) {              foo(ptr p) {
        var ptr q, int a;
  1:    q = malloc (1);                     // DAE
  2:    *q = 123;                           // DSE
  3:    bar(p);              →    bar(p);
  4:    a = *q;                             // DLE
  5:    *p = a;                   *p = 123; // CP
      }                         }
```

We will argue that at each line in the two versions of foo (source and target), the effects of the instruction executed (if any) are equivalent. To do so, we will assume an initial relationship between the memory of the source and target programs, and show that some variant of that relationship persists throughout the function, relying on any call to other functions to maintain a similar relationship.

This relationship will designate one section of each memory as *public*, and require that related locations in the source and target public memories have equivalent values; it will also designate a *private* section of each memory, such that the source program can make changes to its private memory when the target does not make corresponding changes and vice versa. For the technical details of this relation and our notion of equivalence, see §5.2.

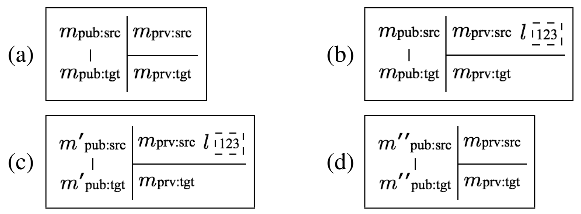


Figure 6. Memory Invariants for the Running Example

We begin at line 1 by assuming the following conditions on the memory of the source and target programs (see Figure 6 (a)):

assume (equivalent arguments) the parameter `p` contains *equivalent* arguments v_{src} in the source and v_{tgt} in the target;

assume (equivalent public memories) there are a set of memory blocks $m_{\text{pub:src}}$ in the source and $m_{\text{pub:tgt}}$ in the target that are *equivalent* and publicly accessible by arbitrary functions;

assume (source private memory) there is a disjoint set of blocks $m_{\text{prv:src}}$ in the source, each of which is exclusively owned by a single function;

assume (target private memory) there is a disjoint set of blocks $m_{\text{prv:tgt}}$ in the target, each of which is exclusively owned by a single function.

After executing line 1, we add the newly allocated block (call it l) to the private source memory $m_{\text{prv:src}}$. It is important to note that we can add the block l to the private source memory because it is a fresh logical block and thus exclusively owned by `foo`. After executing line 2, the block l contains 123 (see Figure 6 (b)).

At line 3, we guarantee that the function calls to `bar` are *equivalent* as follows:

guarantee (equivalent arguments) the arguments v_{src} and v_{tgt} to `bar` are equivalent;

guarantee (equivalent public memories) $m_{\text{pub:src}}$ and $m_{\text{pub:tgt}}$, which are equivalent and publicly accessible;

guarantee (source private memory) each location in $m_{\text{prv:src}} \uplus [l \mapsto 123]$, is exclusively owned by a single function;

guarantee (target private memory) each location in $m_{\text{prv:tgt}}$ is exclusively owned by a single function.

When the calls to `bar` return, we can assume that the new public memories are equivalent to each other (though they may not be the same as the previous public memories), and the private memories are untouched (see Figure 6 (c)):

assume (equivalent public memories) we have new public memories $m'_{\text{pub:src}}$ and $m'_{\text{pub:tgt}}$, which are *evolved* from $m_{\text{pub:src}}$ and $m_{\text{pub:tgt}}$ (see §5.3 for the definition of memory evolution), and are equivalent and publicly accessible;

assume (source private memory) $m_{\text{prv:src}} \uplus [l \mapsto 123]$ is unchanged;

assume (target private memory) $m_{\text{prv:tgt}}$ is unchanged.

At line 4, we load the value 123 from the source's private memory and store it in the variable `a`. At line 5, in the source, we store the value of `a`, which is 123, in the memory cell located at the address $v_{\texttt{src}}$. In the target, we store the constant 123 in the cell at $v_{\texttt{tgt}}$. Since we stored equivalent values at equivalent locations $v_{\texttt{src}}$ and $v_{\texttt{tgt}}$, we will have equivalent public memories $m''_{\text{pub:src}}$ and $m''_{\text{pub:tgt}}$, while leaving the private memories unchanged.

Finally, we return to the callers of `foo` with (see Figure 6 (d))

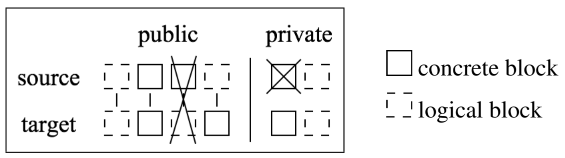


Figure 7. Memory Invariants for Quasi-Concrete Model

guarantee (equivalent public memories) $m''_{\text{pub:src}}$ and $m''_{\text{pub:tgt}}$ that are evolved from $m_{\text{pub:src}}$ and $m_{\text{pub:tgt}}$, and are equivalent and publicly accessible;

guarantee (source private memory) $m_{\text{prv:src}}$;

guarantee (target private memory) $m_{\text{prv:tgt}}$,

where we can ignore the block l because it is not going to be used any more. Note that we here guarantee foo returns with the same private memories it was given initially, as we assumed the same property for the function `bar` after line 3.

5.2 Memory Invariants

As informally discussed above, we prove program refinement using a memory invariant that places conditions on public memories (which must be equivalent in the source and target programs) and private memories (which can differ between them). We now formally define the notion of memory equivalence used in our informal example, and the conditions on the private memories.

The idea of memory equivalence is a simplification of CompCert's memory injection [10]. Our conditions for concrete blocks are inspired from CompCertTSO's support for finite memory [12]. See §7 for more comparisons.

Memory Equivalence We define a more relaxed notion of equivalence than simple equality, which would be too strong in the presence of (unrelated) private memories. We say that a set of blocks m_{src} in the source is equivalent to a set of blocks m_{tgt} in the target when they satisfy the following conditions. First, there should be a bijection, say α, between the block identifiers in m_{src} and those in m_{tgt}. Second, corresponding blocks (*i.e.,* those related by α) should have the same size and validity, and the values they hold at each offset should be *equivalent*. Values are equivalent (w.r.t. α) when either both are the same integer, or they are logical addresses that are at the same offset in corresponding blocks (w.r.t. α). When m_{src} and m_{tgt} are equivalent in this sense, we write $m_{\text{src}} \simeq_{\alpha} m_{\text{tgt}}$.

The condition on the concrete addresses of corresponding blocks merits further explanation. We have four possible cases regarding whether two corresponding blocks are concrete or logical (see the *public* side of Figure 7). The first case in the figure (*i.e.,* source: logical, target: logical) obviously should be allowed. The second case (*i.e.,* source: concrete, target: concrete) should also be allowed but only when the concrete addresses coincide.

The third case (*i.e.,* source: concrete, target: logical) should not be allowed. To allow this case would be to allow the source memory to contain more concrete blocks than the target, which leads to two problems: (i) an arbitrary concrete memory access may succeed in the source but fail in the target; and (ii) a pointer-to-integer cast may raise out-of-memory in the source but succeed in the target. In both cases, the target may have more behaviors than the source, which is disallowed. On the other hand, the final case (*i.e.,* source: logical, target: concrete) is allowed because the situation is exactly the opposite: the source may have more behaviors than the target, which is allowed.

Private Memory For blocks in private memories, we have four possible cases regarding whether the block is in the source or the target, and whether the block is concrete or logical (see *private* side of Figure 7). All the cases are allowed except for source private memory blocks that are concrete, for the same reason that blocks that are concrete in the source and logical in the target are not allowed in memory equivalence: the source memory should not contain more concrete blocks than the target memory.

Memory Invariants A memory invariant β consists of (i) a bijection α between their block identifiers, (ii) the source's private memory $m_{\text{prv:src}}$, and (iii) the target's private memory $m_{\text{prv:tgt}}$. An invariant $\beta = (\alpha, m_{\text{prv:src}}, m_{\text{prv:tgt}})$ holds on a pair of memories m_{src} and m_{tgt} when they contain the private sections $m_{\text{prv:src}}$ and $m_{\text{prv:tgt}}$ and some public sections $m_{\text{pub:src}}$ and $m_{\text{pub:tgt}}$ such that:

$$(m_{\text{src}} \supseteq m_{\text{pub:src}} \uplus m_{\text{prv:src}}) \wedge (m_{\text{tgt}} \supseteq m_{\text{pub:tgt}} \uplus m_{\text{prv:tgt}}) \wedge m_{\text{pub:src}} \simeq_{\alpha} m_{\text{pub:tgt}}$$

where $\uplus$ and $\subseteq$ are the disjoint union and the subset relation.

5.3 Proving Simulation

We are now ready to present our reasoning principle formally. Our basic approach is to verify programs via local simulation in the style of [4].

A function, say `foo`, in the source and target is locally simulated if it satisfies the following conditions. First, consider a typical lifecycle of the source and target functions:

$$\begin{array}{lcll}
\texttt{foo(..) \{} & & \texttt{foo(..) \{} & \texttt{//}\boxed{\beta_{\mathtt{s}}} \\
\quad\texttt{...} & & \quad\texttt{...} & \texttt{//}\ \beta_{\mathtt{c}} \quad \beta_{\mathtt{s}} \sqsubseteq \beta_{\mathtt{c}} \\
\quad\texttt{bar(..);} & & \quad\texttt{bar(..);} & \texttt{//}\boxed{\beta_{\mathtt{r}} \quad \beta_{\mathtt{c}} \sqsubseteq \beta_{\mathtt{r}} \wedge \beta_{\mathtt{c}} =_{\text{prv}} \beta_{\mathtt{r}}} \\
\quad\texttt{...} & \rightarrow & \quad\texttt{...} & \texttt{//}\ \beta'_{\mathtt{c}} \quad \beta_{\mathtt{r}} \sqsubseteq \beta'_{\mathtt{c}} \\
\quad\texttt{gee(..);} & & \quad\texttt{gee(..);} & \texttt{//}\boxed{\beta'_{\mathtt{r}} \quad \beta'_{\mathtt{c}} \sqsubseteq \beta'_{\mathtt{r}} \wedge \beta'_{\mathtt{c}} =_{\text{prv}} \beta'_{\mathtt{r}}} \\
\quad\texttt{...} & & \quad\texttt{...} & \texttt{//}\ \beta_{\mathtt{e}} \quad \beta'_{\mathtt{r}} \sqsubseteq \beta_{\mathtt{e}} \wedge \beta_{\mathtt{s}} =_{\text{prv}} \beta_{\mathtt{e}} \\
\texttt{\}} & & \texttt{\}} &
\end{array}$$

Here boxed conditions are assumed and the others are guaranteed.

First, in `foo`, unknown functional calls such as `bar(..)` and `gee(..)` should be synchronized (*i.e.,* when the target calls `bar`, the source should call `bar` as well). Note that when a known function is called, the verifier can either step into the called function and reason about its code, or treat it as an unknown function call.

Next, at the entry point of `foo`, we assume that we are given memories satisfying a given invariant β_{s}, and equivalent arguments w.r.t. the bijection in β_{s}. Then, we execute the code of foo in the source and the target until the first unknown function call to `bar(..)`. Here we have to show that there is some invariant β_{c} that holds on the current memories and that the arguments to the function `bar` are equivalent w.r.t. the bijection in β_{c}.

Here, we also have to show that the current memories are *evolved* from the memories given initially by showing that the current invariant β_{c} is a *future invariant* of the initial one β_{s} (denoted $\beta_{\text{s}} \sqsubseteq \beta_{\text{c}}$). We say that β_{c} is a future invariant of β_{s} when satisfying the following conditions, which rule out changes to the memory that cannot be caused by the language's operational semantics. First, the bijection in β_{c} should include the bijection of β_{s} because logical blocks cannot be removed during execution (a block becomes invalid rather than removed when it is freed). Second, the other conditions on the public memories in β_{s} and β_{c} are that (i) the size of a block does not change between β_{s} and β_{c}, (ii) an invalid block in β_{s} cannot become valid in β_{c}, and (iii) a concrete block in β_{s} cannot become logical in β_{c}. However, it is important to note that the contents of public memories can change between β_{s} and β_{c} because the operational semantics allows to update values in memory.

Then, we consider the case when the unknown function successfully returns. We can assume that the memories at return time also satisfy some *future* invariant β_{r}. We can also assume that the function `bar` does not change the private memories in β_{c} (denoted $\beta_{\text{c}} =_{\text{prv}} \beta_{\text{r}}$) because there is no way for `bar` to access them in our quasi-concrete model.

We continue through the function, evolving our invariant at non-call steps and performing similar reasoning at other call sites such as `gee(..)`. Finally, when `foo` returns to its caller, we have to show that there is some *future* invariant β_{e} that holds on the current memories. Furthermore, we have to show that we did not change the private memories given in the initial invariant β_{s} (*i.e.,* $\beta_{\text{s}} =_{\text{prv}} \beta_{\text{e}}$). This condition is necessary because, as seen above, we assume that this property holds at the end of any other function call. In this way, we construct a local simulation proof for the `foo`.

6. Verification Examples

In this section, we show how to verify the examples shown in §3. All results here are fully formalized in Coq.

6.1 Arithmetic Optimization I

Consider the transformation in Figure 1. If we assume that integer variables only contain integer values, not logical addresses, the instruction `a = (a - b) + (2 * b - b)` has no effect on the value of `a` and is equivalent to no operation, so the optimization is trivially correct.

How do we know that integer variables only contain integer values? The straightforward answer is that our language is statically type-checked, as in the LLVM IR. However, the key reason why this is possible is that in the quasi-concrete model we actually turn logical addresses into integers when they are cast to `int`, rather than placing logical addresses in integer variables. Also, when we load a value from memory to an integer variable (resp. a pointer variable), if the loaded value is a logical address (resp. an integer value), we raise undefined behavior (*i.e., error*). In other words, the quasi-concrete model induces a form of dynamic type checking in languages that use it. This allows us to verify integer arithmetic optimizations as in this example.

6.2 Dead Code Elimination

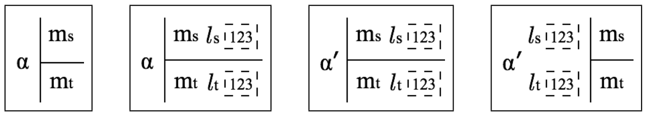
Consider the transformation in Figure 2. This example is similar to the previous one. Since we can assume that integer-typed variables contain only integers, the execution of the call `foo(a)` does not have any side effects. Furthermore, because we know the code of the function `foo`, we do not need to treat it as an unknown function call. Rather, we just step into the code of the function `foo` and execute it in the source.

6.3 Ownership Transfer

Consider the transformation in Figure 3. This example is similar to the running example in §5.1.

Assume that the first invariant below holds before the `malloc`. After allocating blocks l_{s} in the source and l_{t} in the target, and storing 123 in both blocks, we can move the blocks l_{s} and l_{t} into the private sections of the invariant because they are logical and disjoint from the public sections, yielding the second invariant below. Next we call the function `bar`. When it returns, we can assume that the third invariant holds (*i.e.,* the private sections are untouched). After loading, the variable `a` will contain 123, since `p` contains the logical address $(l_{\text{s}}, 0)$ in the source and $(l_{\text{t}}, 0)$ in the target.

Next, when we call `hash_put`, we have to make sure that the arguments are equivalent. The first arguments are equivalent because we assume that we start with equivalent values in variables, and the third arguments are equivalent because `a` contains 123. To show that the second arguments, $(l_{\text{s}}, 0)$ and $(l_{\text{t}}, 0)$, are equivalent, we move the blocks from the private sections to the public section and extend the bijection α' to relate l_{s} and l_{t}) (the fourth invariant below). Such ownership transfer from the private sections to the public section is allowed because the future invariant relation ($\sqsubseteq$) requires only the bijection to be non-decreasing, not the private sections.



6.4 Arithmetic Optimization II

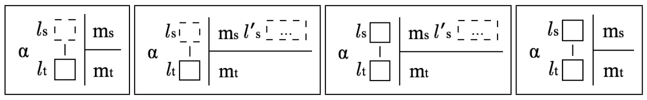
We can easily verify the transformation in Figure 4 for the same reason as in §6.1: because we can assume that all integer variables contain integer values.

6.5 Dead Cast Elimination

Consider the transformation in Figure 5, in the case in which the source uses the quasi-concrete model and the target uses the concrete model.

We begin by assuming that the first invariant below holds before the call to `foo`, where the variable `p` contains equivalent addresses (l_s, i) in the source and $(l_t, i)\downarrow_m$ in the target. Note that the block l_t is concrete, since the target is using the fully concrete model. After the allocation of a block, say l'_s, in `foo` in the source, we move it to the source's private memory, yielding the second invariant below. Here it is important to note that if the source was using the concrete model, we could not move the block l'_s into the private section because l'_s would be concrete, which would invalidate our proof.

After the cast, the block l_s becomes concrete, yielding the third invariant below. Here it is important to note that if the target language was using the quasi-concrete model and l_t were logical, then we would produce an invariant in which l_s is concrete and l_t is logical, which would be an invalid invariant. After `foo` returns, we simply drop the block l'_s from the source private section because we do not use it, yielding the fourth invariant below. Then we can proceed to verify the rest of the code.



6.6 Identity Compilers

As a sanity check for our reasoning principles, we wrote an identity compiler from our language with the quasi-concrete model to itself, and a simple compiler from our language with the quasi-concrete model to the same language with the concrete model. The latter compiler just eliminates dead casts of the form `_ = (int) p`. We successfully verified these two compilers in Coq using our reasoning principles.

7. Discussion and Related Work

The quasi-concrete model refines the C standard by giving semantics to more programs involving pointer operations. We intend to use this model for compiler verification tasks, extending the range of common optimizations that can be verified. Ultimately, we would like to build a verified translation validation framework for LLVM that supports all commonly-used features of C. We would also like to integrate our model with CompCert and use it to justify new CompCert optimizations. We believe that our ideas are readily applicable to CompCert(TSO) and related projects like Vellvm [15, 16] because our memory model and notion of memory invariant are technically very close to CompCert's. (Vellvm also uses CompCert's memory model.) Essentially, all that would have to change in the proofs are the cases handling pointer to integer casts.

Additional C language features There are numerous other C language features that have some interaction with the memory model. Some of them, such as *indeterminate values* [5, §3.19.2p1], *dangling pointers* [5, §6.2.4p2], and *infinite loops with no side-effects* [5, §6.8.5p6], have semantics that are largely orthogonal to the pointer realization used in our quasi-concrete model. Similarly, our model explicitly allows *unsafely derived pointers*, which are permitted in C11 and implementation-defined in C++11 [5, §3.7.4p4]. We allow them in order to support low-level programming idioms such as XOR linked lists and compressed oops in HotSpot JVM.

Our paper does not directly address threads, so we cannot claim with certainty that the model extends to handle them. However, we see no obstacles in this direction, and the quasi-concrete model is similar to CompCertTSO, which does support a weak memory model and threads, so we are optimistic that this extension to the semantics should follow similarly.

A few language features require some adaptation of our memory model. For instance, we can adapt the quasi-concrete model to support *union types* and *strict aliasing*, following Krebbers' technique [6], which works regardless of whether the model is concrete, logical or hybrid.

As another example, in C, `char*` is a "universal" pointer type, which allows efficient bulk data moves via `memcpy`. Krebber's variant of CompCert [7] already supports this semantics using a logical memory, and the quasi-concrete model is compatible with that solution. Briefly: we let `char` types store byte-indexed logical values (such as $(l, 10) : 2$, which denotes the second byte of the logical address $(l, 10)$). This strategy works because a `char` is implicitly cast to an integer when used in arithmetic operations, and thus we can simply treat these casts as side-effecting (*i.e.*, realizing the logical addresses). This approach lose (almost) no optimization opportunities because byte-indexed logical addresses are typically loaded from the memory and thus (mostly) already treated as public by the compiler.

Alias analyses The quasi-concrete model is largely compatible with common alias analyses. For instance, it can be used to justify *size-based alias analysis*, which considers pointers to differently-sized objects as distinct. For example, in the code below, there is no alias between `p` and `q`: even if `q` points to the block pointed to by `p`, loading or storing a double value in the block will fail since the block is not big enough to contain double values.

```
int* p = malloc(sizeof(int));
double* q = foo(p);     // no alias between p and q
```

It also justifies *freshness-based alias analysis*, which assumes that the result of `malloc` is distinct from all other pointers. The following example of constant propagation is valid in the quasi-concrete model since `q` points to a fresh block that is different from the block pointed to by `p`. It is important to note that there is no alias between `p` and `q` even after the fresh block is realized. The reason is because even if `p` and `q` may be cast to the same integer, they still point to different blocks as pointer values.

```
foo(ptr p) {                     foo(ptr p) {
  var ptr q, int a,b,r;            var ptr q, int a,b,r;
  q = malloc (1);                  q = malloc (1);
  a = (int) q;                     a = (int) q;
  b = *p;                   →      b = *p;
  *q = 123;                        *q = 123;
  r = *p;                          r = b;          // CP
}                                }
```

Coq Formalization All the proofs reported in this paper have been fully formalized in Coq and can be found in the project webpage. Our Coq formalization is about 10,000 lines of code, excluding empty lines and library code. The formalization took about 2 person-months to complete.

Optimization Examples All optimization examples presented in the paper are performed by Clang 3.4.2 and/or GCC 4.8.3. Examples in C and their compilation results can be found in the project webpage.

Formal Memory Models There have been numerous efforts to formalize C semantics, both from the perspective of clarifying the specification and defining implementations with formal semantics [2, 3, 8, 9, 11]. These invariably use variations of the logical memory model, where each allocation is associated with some abstract identifier and pointers consist of an identifier and some path representing an offset into the memory block, except for the work of Norrish [11] which uses the concrete model.

Comparison with CompCert CompCert [9, 10] and its various extensions currently allow casting pointers to and from integers, but the semantics preserves the logical representation of pointers after the cast. As a result, integer variables can contain not only normal 32-bit integers, but also logical pointer representations. In the higher-level languages (CompCert C and Clight), performing arithmetic on cast pointer is treated as a program error, whereas in the low-level languages (from Cminor down to assembly), adding and subtracting integer values from converted pointers is defined and affects only the offset into the pointer's logical block. There has also been work on extending the semantics to support pointer fragments to allow, for example, `memcpy` to work on memory containing pointers [7], but these extensions still cannot fully support arithmetic operations on pointer values that have been cast to an integer type.

Comparison with CompCertTSO The CompCertTSO compiler [12] extends CompCert's Clight language with threading and atomic memory primitives following the x86-TSO relaxed memory model. Similar to us, CompCertTSO's memory model also supports finite memory, but uses a different mechanism to do so. It has a distinguished logical block, where the offset serves as essentially a concrete memory address. During compilation, all memory operations are lowered to act only on a single finite logical block. This allows the source and target languages, with infinite and finite memory respectively, to share a single memory model, and simplifies the correctness statements by removing the need for CompCert's memory injections. CompCertTSO handles pointer-integer casts in the same way as CompCert, with the same limitations.

Comparison with the Symbolic Value Approach Most recently, Besson et al. have proposed an extension to CompCert's memory model that gives semantics to bit-masking operations on pointers and uninitialized values [1]. Their approach involves adding lazily-evaluated symbolic expressions, including arbitrary operations on the representation of pointers, to the class of semantic values. Symbolic values are forced whenever a concrete value is needed to take a step, for example to access memory through a pointer or in the guard of a conditional. The mapping is performed by a normalization function given as a parameter of the semantics. The normalization function is partial, and is only defined precisely when the symbolic value evaluates to a unique result under every assignment from logical block identifiers to concrete addresses (subject to some validity conditions).

The semantics of Besson et al. is necessarily deterministic: non-determinism is interpreted as undefined behavior, while our model captures the non-deterministic allocation of concrete addresses. Furthermore, their semantics is complex and indeed intractable: their normalization is implemented with an SMT solver, and the semantics in general is too complex to serve as a mental model for ordinary C programmers. Normalization in our semantics, on the other hand, is a straightforward translation from pointers to concrete blocks and integers.

Most importantly, while their approach gives semantics to non-strictly-conforming C programs involving bit-masking of pointers and uninitialized values, it fails to define useful programs that use integer-pointer casts. Consider the `hash_put` example discussed in §3.4, where a pointer is hashed and then presumably used to index into an array. Since the resulting memory location will depend on the concrete layout of memory, the resulting program will have undefined behavior in their semantics. In general, any program that displays non-determinism due to the realization of pointers in our model is necessarily undefined in Besson et al.'s model.

Acknowledgements

We are grateful to Robbert Krebbers for useful discussions. This research is supported in part by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning(MSIP) / National Research Foundation of Korea(NRF) (Grant NRF-2008-0062609), in part by the NSF projects Validating Program Transformations in a Mechanized LLVM (NSF 1065166), WATCHDOG: Hardware-Assisted Prevention of All Use-After-Free Security Vulnerabilities (NSF 1116682), and Improving Parallel Program Reliability Through Novel Approaches to Precise Dynamic Data Race Detection (NSF 1337174), and in part by the EC FP7 project ADVENT (308830).

References

[1] F. Besson, S. Blazy, and P. Wilke. A precise and abstract memory model for C using symbolic values. In *APLAS*, 2014.

[2] C. Ellison and G. Rosu. An executable formal semantics of C with applications. In *POPL*, 2012.

[3] D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *PLDI*, 2014.

[4] C.-K. Hur, D. Dreyer, G. Neis, and V. Vafeiadis. The marriage of bisimulations and Kripke logical relations. In *POPL*, 2012.

[5] ISO. *ISO/IEC 9899:2011 Information technology – Programming languages – C*. 2011.

[6] R. Krebbers. Aliasing restrictions of C11 formalized in Coq. In *CPP*, 2013.

[7] R. Krebbers, X. Leroy, and F. Wiedijk. Formal C semantics: CompCert and the C standard. In *ITP*, 2014.

[8] R. Krebbers and F. Wiedijk. A formalization of the C99 standard in HOL, Isabelle and Coq. In *CICM*, 2011.

[9] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[10] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert memory model, version 2. Research report RR-7987, INRIA, June 2012.

[11] M. Norrish. C formalised in HOL. Computer Laboratory Technical Report 453, University of Cambridge, Nov. 1998.

[12] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *Journal of the ACM*, 60(3):22, 2013.

[13] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *SOSP*, 2013.

[14] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[15] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *POPL*, 2012.

[16] J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *PLDI*, 2013.