An Axiomatic Specification for Sequential Memory Models

William Mansky, Dmitri Garbuzov, and Steve Zdancewic

University of Pennsylvania

Abstract. Formalizations of concurrent memory models often represent memory behavior in terms of sequences of operations, where operations are either reads, writes, or synchronizations. More concrete models of (sequential) memory behavior may include allocation and free operations, but also include details of memory layout or data representation. We present an abstract specification for sequential memory models with allocation and free operations, in the form of a set of axioms that provide enough information to reason about memory without overly constraining the behavior of implementations. We characterize a set of "well-behaved" programs that behave uniformly on all instances of the specification. We show that the specification is both feasible—the CompCert memory model implements it—and usable—we can use the axioms to prove the correctness of an optimization that changes the memory behavior of programs in an LLVM-like language.

Keywords: memory models, optimizing compilers, deep specifications

1 Introduction

When reasoning about compilers and low-level code, it is not enough to treat memory as an assignment of values to locations; aliasing, pointer arithmetic, allocation, concurrency behavior, and many other factors complicate the picture, and without accounting for these factors our reasoning says nothing about the programs that actually run on processors. Memory models provide the necessary abstraction, separating the behavior of a program from the behavior of the memory it reads and writes. There have been many formalizations of concurrent memory models, beginning with sequential consistency [1] (in which memory must behave as if it has received an ordered sequence of read and write operations) and extending to more relaxed memory models. Most of these models include a theorem along the lines of "well-synchronized programs behave as if the memory model is sequentially consistent," characterizing a large class of programs that behave the same regardless of the concurrent memory model [7].

What, then, is the behavior of a sequentially consistent memory model? When the only memory operations are reads and writes (and possibly synchronization operations), the answer is simple: each read of a location reads the value that was last written to that location. In other words, the memory does in fact act as an assignment of values to locations. If we try to model other memory operations, however, the picture becomes more complicated. C and many related intermediate and low-level languages include at least allocation and free operations, and we might also want to include casts, structured pointers, overlapping locations, etc. Even restricting ourselves to sequential memory models, we can see that the space of possible models is much larger than "sequential consistency" suggests.

Formalizing memory models is a crucial step in compiler verification. Projects such as CompCert [4], CompCertTSO [9], Vellvm [11], and Compositional CompCert [8] specify memory models as part of the process of giving semantics to their various source, target, and intermediate languages, and use their properties in proving the correctness of optimizations and program transformations. The (in most cases sequential) memory models in these works include some of the complexity that more abstract formalisms lack, but they are also tightly tied to the particular languages and formalisms used in the projects. Compiler verification stands to benefit from memory model specifications that generalize away from the details of particular memory models, specifications which encompass most commonly used models and allow reasoning about programs without digging into the details of particular models. Generic specifications of memory models have the potential to lead to both simpler proofs—since all the reasoning about a particular model is encapsulated in a proof that it satisfies the specification—and more general ones—since a proof using a specification is true for any instance of that specification.

In this paper, we develop a specification for sequential memory models that support allocation and free operations as well as reads and writes, and demonstrate its use in reasoning about programs. We prove a sequential counterpart to the "well-synchronized programs are sequentially consistent" theorem, characterizing the set of programs that have the same behavior under any sequential memory model that meets our specification. We also show that CompCert's memory model is an instance of our specification, and verify a dead code elimination optimization for an LLVM-like language using the specification, resulting in a proof that is measurably simpler than the corresponding proof in Vellvm. All definitions and proofs have been formalized in the Coq proof assistant, so that our specification can be used for any application that requires mechanized proofs about programs with memory.

2 An Abstract Sequential Memory Model

A memory model is a description of the allowed behavior of a set of memory operations over the course of a program. A memory model can be defined in various ways: as a set of functions that can be called along with some guarantees on their results, as a description of the set of valid traces of operations performed by the execution of a program, or as an abstract machine that receives and responds to messages. In each case, the memory model makes a set of operations available to programs and provides some guarantees on their behavior. These operations always include reading and writing, and in many models these are the only operations; however, there are many other memory-related operations used in real-world programs. The main question is one of where we draw the line between program and memory. Is the runtime system that handles memory allocation part of the memory, or a layer above it? Does a cast from a pointer to an integer involve the memory, or is it a computation within the program? Does the memory contain structured blocks in which different references may overlap, or are structured pointers program objects that must be evaluated to references to distinct locations before they are read or written?

Our goal is to formalize the interface that memory provides to a programming language. We aim to give an abstract specification for memory models that can be used to define the semantics of a language, and to prove useful properties of programs in that language independently of the implementation details of any particular memory model. Our specification should describe the assumptions about memory that programmers can make when writing their programs and verifiers can make while reasoning. Since from the program's perspective the runtime system and the memory model are not distinct, our specification should include the operations provided by the runtime system. It should be operational, so that we can use it to define operational semantics for programming languages, and it should provide as many axioms as are needed to make the behavior of memory predictable without overconstraining the set of possible implementations.

For our specification, we begin with four operations: read, write, alloc, and free. These operations appear in code at almost every level. They are, for instance, the operations supported by the CompCert memory model [5], which has been used to verify a compiler from C to machine code. Although CompCert's model provides a realistic and usable formalization of the semantics of these operations, it is not the *only* such formalization. Other choices, such as CompCertTSO's [9] or the quasi-concrete model [2], may allow more optimizations on memory operations or a cleaner formulation of some theorems. We may want to store values in memory other than those included in CompCert, or abstract away from the details of blocks and offsets.

Our aim is to give a simple specification of memory models such that:

- Most (operational) memory models that support read, write, alloc, and free can be seen as instances of the specification.
- The specification provides the guarantees on these operations needed to reason about programs.

Then we can use this specification to reason about programs independently of the particular memory model being used, and by proving that particular models (such as CompCert's) meet the specification, be assured that our reasoning is valid for those models.

2.1 Memory Model Axioms

Previously, we mentioned three main approaches to specifying memory models. In the functional approach (e.g. CompCert [5]), each operation is a function with its own arguments and return type, and restrictions are placed on the results of the functions. In the abstract-machine approach (e.g. CompCertTSO [9]), memory is a separate component from the program with its own transition system, and steps of the system are produced by combining program steps and memory steps. In the axiomatic approach (taken in most concurrent memory models), a set of rules are given that allow some sequences of memory operations and forbid others. A definition in one of these styles is often provably equivalent to a definition in another style, although the axiomatic approach can be used to formalize some models that cannot be expressed in other ways (i.e. non-operational models). We will give our specification axiomatically, but in a way that guarantees that instances are operational. Our axioms should be true for all (reasonable) memory models, and also provide enough information to prove useful properties of a language that uses the specification.

Our model begins with a set $\mathcal{L}$ of *locations* and a set $\mathcal{V}$ of *values*. Every memory operation targets exactly one location, and locations are *distinct*: we can check whether two locations are equal, and a change to one location should not affect any other location. Locations may be thought of as unique addresses or memory cells. Values are the data that are stored in the memory; we do not account for size differences in stored data, and assume that any value can be stored in any location and that each location can hold one value.

Definition 1. Given a location $\ell \in \mathcal{L}$ and a value $v \in \mathcal{V}$, a *memory operation* is one of $\mathsf{read}(\ell, v)$, $\mathsf{write}(\ell, v)$, $\mathsf{alloc}(\ell)$, and $\mathsf{free}(\ell)$. The operations $\mathsf{write}(\ell, v)$, $\mathsf{alloc}(\ell)$, and $\mathsf{free}(\ell)$ *modify* the location ℓ. Over the course of execution, a program produces a series of memory operations. An operational memory model can be given as a predicate $\mathsf{can\_do}$ on a sequence of memory operations $m = op_1 \dots op_k$ (called the *history*) and an operation op, such that $\mathsf{can\_do}(m, op)$ holds if and only if, given that the operations in m have occured, the operation op can now be performed. A sequence of operations $op_1 \dots op_k$ is *consistent* with a memory model if $\mathsf{can\_do}(op_1 \dots op_{i-1}, op_i)$ for each $i < k$, i.e., each operation in the sequence was allowable given the operations that had been performed so far.

The axioms shown in Figure 1 restrict the possible behavior of a $\mathsf{can\_do}$ function. (We write $loc(op)$ for the location accessed by op.) The first two axioms state the distinctness of locations, requiring that operations on one location do not affect the operations possible on other locations. The remaining rules enforce (but do not completely determine) the intended semantics of each kind of memory operation: e.g., a $\mathsf{write}(\ell, v)$ operation must allow v to be read at ℓ. We do not completely constrain the semantics of the operations, but we attempt to capture the expectations of a programmer about each operation: it should be possible to allocate free memory and free allocated memory, write to allocated memory and read the last value written, etc., and it should not be possible to free memory that is already free, allocate memory that is already allocated, read values that have not been written, etc. Although each axiom only specifies the interaction between the new operation and the most recent operation performed, we can

$$\text{loc-comm}\ \frac{loc(op) \neq loc(op')}{\mathsf{can\_do}(m\ op, op') = \mathsf{can\_do}(m\ op', op)}$$

$$\text{loc-drop}\ \frac{loc(op) \neq loc(op') \quad \mathsf{can\_do}(m, op)}{\mathsf{can\_do}(m\ op, op') = \mathsf{can\_do}(m, op')}$$

$$\text{read-noop}\ \frac{\mathsf{can\_do}(m, \mathsf{read}(\ell, v))}{\mathsf{can\_do}(m\ \mathsf{read}(\ell, v), op) = \mathsf{can\_do}(m, op)}$$

$$\text{read-written}\ \frac{\mathsf{can\_do}(m, \mathsf{write}(\ell, v))}{\mathsf{can\_do}(m\ \mathsf{write}(\ell, v), \mathsf{read}(\ell, v')) = (v = v')}$$

$$\text{write-not-read}\ \frac{\mathsf{can\_do}(m, \mathsf{write}(\ell, v)) \quad \forall v'.\ op \neq \mathsf{read}(\ell, v')}{\mathsf{can\_do}(m\ \mathsf{write}(\ell, v), op) = \mathsf{can\_do}(m, op)}$$

$$\text{not-mod-write}\ \frac{\mathsf{can\_do}(m, op) \quad op \text{ does not modify } \ell}{\mathsf{can\_do}(m\ op, \mathsf{write}(\ell, v)) = \mathsf{can\_do}(m, \mathsf{write}(\ell, v))}$$

$$\text{write-any-value}\ \frac{}{\mathsf{can\_do}(m, \mathsf{write}(\ell, v)) = \mathsf{can\_do}(m, \mathsf{write}(\ell, v'))}$$

$$\text{alloc-allows}\ \frac{\mathsf{can\_do}(m, \mathsf{alloc}(\ell))}{\begin{array}{c}\mathsf{can\_do}(m\ \mathsf{alloc}(\ell), \mathsf{write}(\ell, v)) \wedge \neg\mathsf{can\_do}(m\ \mathsf{alloc}(\ell), \mathsf{alloc}(\ell)) \wedge \\ \mathsf{can\_do}(m\ \mathsf{alloc}(\ell), \mathsf{free}(\ell))\end{array}}$$

$$\text{free-allows}\ \frac{\mathsf{can\_do}(m, \mathsf{free}(\ell))}{\begin{array}{c}\neg\mathsf{can\_do}(m\ \mathsf{free}(\ell), \mathsf{read}(\ell, v)) \wedge \\ \mathsf{can\_do}(m\ \mathsf{free}(\ell), \mathsf{alloc}(\ell)) \wedge \neg\mathsf{can\_do}(m\ \mathsf{free}(\ell), \mathsf{free}(\ell))\end{array}}$$

$$\text{base-allows}\ \frac{}{\begin{array}{c}\neg\mathsf{can\_do}(\cdot, \mathsf{read}(\ell, v)) \wedge \mathsf{can\_do}(\cdot, \mathsf{alloc}(\ell)) \wedge \\ \neg\mathsf{can\_do}(\cdot, \mathsf{free}(\ell))\end{array}}$$

Fig. 1. The axioms of the memory model specification

derive rules that connect each new operation to "the last relevant operation", e.g., the last alloc or free of a location being written.

If a behavior is "implementation-dependent", or might vary across different memory models, then the axioms leave it unspecified. Two major kinds of operation are left unspecified: reads from locations that have been allocated but not written to (we call these locations "uninitialized"), and writes to locations that have not been allocated. Parameterizing by the sets $\mathcal{L}$ and $\mathcal{V}$ also implicitly leaves some aspects of the memory model unspecified. We do not constrain the kinds or sizes of data that can be stored (although we do require that any value can be stored in any location and read back unchanged), and we do not specify whether there are a finite or an infinite number of locations. If we instantiate the specification with an infinite $\mathcal{L}$, then for any m there is an ℓ such that $\mathsf{can\_do}(m, \mathsf{alloc}(\ell))$; if we choose a finite $\mathcal{L}$, then we may reach states in which there is no such ℓ. The effect of running out of memory on program executions is left to the language semantics, as we will show in Section 4.1.

In the context of concurrent memory models, it is usually assumed or proved that well-synchronized programs are sequentially consistent, regardless of the relaxations allowed by the memory model. This allows the complexities of the model to be hidden from the programmer, and means that verification of a certain (large) class of programs can be done independently of the relaxed model. Our axiomatization admits a similar property for sequential memory models. Consider a simple abstract machine that associates each memory location with one of three states: free, uninit, or stored(v), where v is a value. The machine has the expected transitions for each memory operation, as shown in the register automata of Figure 2.

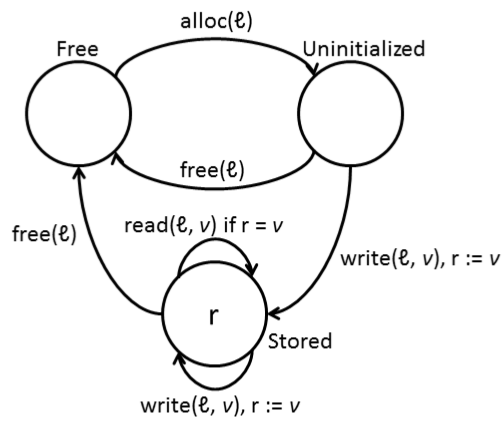

Fig. 2. The simple memory machine

Definition 2. The simple machine state corresponding to a history m, written $\mathsf{SM}(m)$, is the state reached by starting in the state in which all locations are Free and taking the transitions in m in order. The can_do predicate induced by the simple machine is the one such that $\mathsf{can\_do}_{\mathsf{SM}}(m, op)$ when there is a transition labeled op from $\mathsf{SM}(m)$.

Then we can prove the following theorems:

Theorem 1. *The simple machine satisfies the memory model axioms.*

Theorem 2. *If a program never reads an uninitialized location and never writes to a free location, then for any* can_do *predicate that satisfies the axioms and any consistent history m,* $\mathsf{can\_do}(m, op)$ *if and only if* $\mathsf{can\_do}_{\mathsf{SM}}(m, op)$.

This gives us a class of programs for which any model that satisfies the axioms is equivalent. This is the sequential counterpart to the theorem that well-synchronized programs have only sequentially consistent executions. For the (large) set of programs that take a principled approach to memory and avoid implementation-dependent behavior, we can reason using the simple machine and derive results that are applicable to any memory model that implements the specification; this has the potential to greatly simplify our proofs. On the other hand, many interesting programs may not meet the requirements of the theorem. In this case, we may still be able to reason using the specification: while we cannot turn to the fully defined simple machine, we can still use the axioms to draw conclusions about a program's memory behavior. Finally, if we expect that the correctness of our reasoning depends on a particular implementation, then we can go beneath the specification and work with the implementation directly. Having a reasoning principle for "well-behaved" programs simplifies our reasoning when it can be applied, but does not force us to give up on reasoning about programs that are not well-behaved.

3 Instantiating the Specification

The CompCert verified C compiler includes a C-like memory model [5], which is used to verify its transformations. In fact, it includes both a specification of a memory model and an implementation of that specification. Memory is modeled as a set of non-overlapping blocks, each of which behaves as an array of bytes; an *address* is a pair (b, o) of a block and an offset into the array. The specification defines four functions that can be called by programs (`alloc`, `free`, `load`, and `store`) and states properties on them. Most of these properties center around the *permissions* associated with each address, such as Readable, Writeable, and Freeable, which indicate which operations can be performed on the address. Figure 3 shows some of the properties for `store`; the other operations have similar axioms. CompCert's memory implementation manages the bounds, allocation state, and content of each block in a way that is shown to satisfy the axioms.

$$\frac{(b,o) \text{ has permission } \mathsf{Writeable} \text{ in } M_1}{\exists M_2.\ \mathtt{store}(M_1,b,o,v) = M_2}$$

$$\frac{\mathtt{store}(M_1,b,o,v) = M_2 \quad (b',o') \text{ has permission } p \text{ in } M_1}{(b',o') \text{ has permission } p \text{ in } M_2}$$

$$\frac{\mathtt{store}(M_1,b,o,v) = M_2 \quad (b',o') \text{ has permission } p \text{ in } M_2}{(b',o') \text{ has permission } p \text{ in } M_1}$$

$$\frac{\mathtt{store}(M_1,b,o,v) = M_2}{(b,o) \text{ has permission } \mathsf{Writeable} \text{ in } M_1}$$

Fig. 3. A few of CompCert's `store` axioms

Although the specification level of the CompCert memory model abstracts away from some of the details of the implementation, it has some limitations as a generic memory model specification. It is tied to CompCert's particular definition of values and its notion of blocks. Furthermore, there is no uniformity across the different memory operations; each function takes different arguments and has a different result type. The CompCert memory model specification does not include an axiom that says "operations on different locations are independent"; indeed, it is difficult to state such an axiom, since "operations" are not quantifiable objects. Instead, we can look at the axioms stating that, e.g., a `store` to (b, o) does not change the permissions of another block and a `free` succeeds as long as the target address is $\mathsf{Freeable}$, and conclude that a `free` can occur after a `store` to a different location if and only if it could occur before the `store`.

Using this sort of reasoning, we can show that the CompCert memory model specification satisfies our specification in turn. We "implement" each one of our memory operations with a call to the corresponding CompCert function, with one allocated block for each allocated memory location. Our specification does not include details about the size of values, so we restrict ourselves to 32-bit values (which includes most CompCert values).

Definition 3. Given CompCert memory states M and M', let $M \xrightarrow{op} M'$ if the function call corresponding to op can be applied to M to yield M'. Let $\mathsf{can\_do}_{\mathsf{CC}}(m, op)$ be true when there exist CompCert memory states M_1, M_2 such that $\texttt{empty} \xrightarrow{m}{}^* M_1$ and $M_1 \xrightarrow{op} M_2$, where `empty` is the initial CompCert memory state.

Theorem 3. $\mathsf{can\_do}_{\mathsf{CC}}$ *satisfies the axioms of our specification.*

This provides some evidence for the feasibility of our specification, since the CompCert memory model (when used in this restricted way) satisfies its axioms. By Theorem 2, we also know that on programs that do not read uninitialized locations or write to free locations, the CompCert memory model has the same behavior as the simple abstract machine. (The CompCert specification requires that reads of uninitialized locations return a special `undef` value and writes to free locations fail, which is just one point in the design space of memory models allowed by our specification; reads of uninitialized locations could also fail or return arbitrary values, for instance.)

Interestingly, while we choose the set of 32-bit CompCert values as our $\mathcal{V}$ for this instance, we do not need to choose a particular $\mathcal{L}$ in order to prove the above theorem. Each allocated location is mapped to a block, but the set of locations need not be the set of blocks itself. In the CompCert memory model, an `alloc` call always succeeds, implying that memory is infinite; however, the proof of implementation still applies even if we choose a finite $\mathcal{L}$. In this case, while CompCert's memory model is always willing to allocate more blocks, programs may still run out of distinct locations to request. Our specification's view of CompCert's infinite-memory model gives us an interface that can be either infinite-memory or finite-memory.

4 Using the Specification

From the perspective of a programming language, a memory model fills in the gaps in the semantics and provides some guarantees about the observable behavior of the memory. In this section we will show how our specification can be used for these tasks, by defining the semantics of a language using the specification and verifying an optimization against it.

4.1 MiniLLVM

Our example language is MiniLLVM, a language based on the LLVM intermediate representation [3]. MiniLLVM has no indirect jumps, so we infer the targets of jumps from the control flow graph rather than making them explicit in the instructions; otherwise, the syntax of the language closely resembles LLVM.

$$
\begin{array}{rl}
\mathit{expr} ::= \%\mathrm{x} \mid @\mathrm{x} \mid \mathrm{c} & \qquad \mathit{type} ::= \texttt{int} \mid \mathit{type}\ \texttt{pointer} \\
\mathit{instr} ::= & \%\mathrm{x} = \mathrm{op}\ \mathit{type}\ \mathit{expr},\ \mathit{expr} \mid \%\mathrm{x} = \texttt{icmp}\ \mathrm{cmp}\ \mathit{type}\ \mathit{expr}, \mathit{expr} \mid \\
& \texttt{br}\ \mathit{expr} \mid \texttt{br} \mid \texttt{alloca}\ \%\mathrm{x}\ \mathit{type} \mid \\
& \%\mathrm{x} = \texttt{load}\ \mathit{type}^*\ \mathit{expr} \mid \texttt{store}\ \mathit{type}\ \mathit{expr},\ \mathit{type}^*\ \mathit{expr} \mid \\
& \%\mathrm{x} = \texttt{cmpxchg}\ \mathit{type}^*\ \mathit{expr},\ \mathit{type}\ \mathit{expr},\ \mathit{type}\ \mathit{expr} \mid \\
& \%\mathrm{x} = \texttt{phi}\ [\mathit{node}_1, \mathit{expr}_1], \ldots, [\mathit{node}_k, \mathit{expr}_k] \mid \\
& \%\mathrm{x} = \texttt{call}\ \mathit{type}\ \mathit{expr}(\mathit{expr}, \ldots, \mathit{expr}) \mid \texttt{return}\ \mathit{expr} \mid \texttt{output}\ \mathit{expr}
\end{array}
$$

A MiniLLVM program P is a list of function definitions $(f, \ell, \mathit{params}, G)$, where f is the name of the function, ℓ is its location in memory, *params* is the list of the function's formal parameters, and G is the function's control-flow graph (CFG). (For simplicity, we assume that each node in a CFG contains exactly one instruction.) A *configuration* is either an error state Error or a tuple $(f, p_0, p, \mathit{env}, \mathit{st}, \mathit{al})$, where f is the name of the currently executing function, p_0 is the previously executed program point, p is the current program point, *env* is the environment giving values for thread-local variables, *st* is the call stack, and *al* is a record of the memory locations allocated by the currently executing function. The semantics of MiniLLVM are given by a transition relation $P \vdash c \xrightarrow{a} c'$, where a is either a list of memory operations performed in the step or a value output by the `output` instruction. A few of the semantic rules for MiniLLVM instructions are shown in Figure 4, where P_f is the CFG for the function f in P and $\mathsf{succ}(p)$ is the successor node of p in its CFG. We make a point of allowing the `store` instruction to fail into an Error state so that in our example optimization—a dead store elimination—we can safely remove ill-formed stores.

Note that the interaction between the semantics of MiniLLVM and the memory model is restricted to the transition labels. We complete the semantics by combining the transitions of the language with an instance of the memory model specification, passing the memory operations to the $\mathsf{can\_do}$ predicate and retaining the output values, if any:

$$
\frac{P \vdash c \xrightarrow{op_1, \ldots, op_n, v_1, \ldots, v_k} c' \quad \mathsf{can\_do}(m, op_1 \ldots op_n)}{P \vdash (c, m) \xrightarrow{v_1, \ldots, v_k} (c', m\ op_1\ \ldots\ op_n)}
$$

$$\frac{\mathsf{Label}\ P_f\ p = (\%x = \text{op}\ ty\ e_1, e_2) \quad (e_1\ \text{op}\ e_2, env) \Downarrow v}{P \vdash (f, p_0, p, env, st, al) \to (f, p, \mathsf{succ}(p), env(x \mapsto v), st, al)}$$

$$\frac{\mathsf{Label}\ P_f\ p = (\mathtt{alloca}\ \%x\ ty)}{P \vdash (f, p_0, p, env, st, al) \xrightarrow{\mathsf{alloc}(\ell)} (f, p, \mathsf{succ}(p), env(x \mapsto \ell), st, al \cup \{\ell\})}$$

$$\frac{\mathsf{Label}\ P_f\ p = (\%x = \mathtt{load}\ ty^*\ e) \quad (e, env) \Downarrow \ell}{P \vdash (f, p_0, p, env, st, al) \xrightarrow{\mathsf{read}(\ell, v)} (f, p, \mathsf{succ}(p), env(x \mapsto v), st, al)}$$

$$\frac{\mathsf{Label}\ P_f\ p = (\mathtt{store}\ ty_1\ e_1, ty_2{}^*\ e_2) \quad (e_1, env) \Downarrow v \quad (e_2, env) \Downarrow \ell}{P \vdash (f, p_0, p, env, st, al) \xrightarrow{\mathsf{write}(\ell, v)} (f, p, \mathsf{succ}(p), env, st, al)}$$

$$\frac{\begin{array}{c}\mathsf{Label}\ P_f\ p = (\mathtt{store}\ ty_1\ e_1, ty_2{}^*\ e_2) \\ e_1 \text{ fails to evaluate in } env \text{ or } e_2 \text{ fails to evaluate to a pointer in } env\end{array}}{P \vdash (f, p_0, p, env, st, al) \to \mathsf{Error}}$$

$$\frac{\mathsf{Label}\ P_f\ p = (\mathtt{output}(e)) \quad (e, env) \Downarrow v}{P \vdash (f, p_0, p, env, st, al) \xrightarrow{v} (f, p, \mathsf{succ}(p), env, st, al)}$$

$$\frac{}{P \vdash \mathsf{Error} \xrightarrow{a} \mathsf{Error}}$$

Fig. 4. Part of the transition semantics of MiniLLVM

So while, e.g., a `load` operation may produce $\mathsf{read}(\ell, v)$ for any v, the only v that will be allowed by the $\mathsf{can\_do}$ predicate is the one stored at ℓ. We can see that this definition of the language's semantics works equally well for any instance of the memory model specification.

Finite Memory Semantics In Section 1, we noted that our specification encompasses both infinite-memory and finite-memory models, and indeed our semantics for MiniLLVM works in either case. However, it is interesting to consider the way that finite memory is reflected in the semantics. If the set of locations is finite, then we may reach a state (c, m) in which $\mathsf{can\_do}(m, \mathsf{alloc}(\ell))$ does not hold for any ℓ. In this case, the combining rule cannot be applied, and (c, m) is stuck. In terms of optimizations, this means that `alloca` instructions may not be removed from programs, since this may enable behaviors that were previously impossible due to the out-of-memory condition.

An alternative approach is to treat out-of-memory as an error state. We can obtain this semantics by adding one more rule:

$$\frac{P \vdash c \xrightarrow{\mathsf{alloc}(\ell)} c' \quad \forall \ell.\ \neg \mathsf{can\_do}(m, \mathsf{alloc}(\ell))}{P \vdash (c, m) \to (\mathsf{Error}, m)}$$

Now the language semantics catches the out-of-memory condition and transitions to an error state rather than getting stuck. This new semantics allows `alloca` instructions to be removed but not inserted, since optimizations should not introduce new errors. (With a more sophisticated treatment of $\mathcal{L}$ in our proofs, we may be able to state a semantics that allows both adding and removing `alloca`.) We can choose whichever semantics is appropriate to the language or the application at hand; our specification implicitly makes the behavior of out-of-memory programs a question of language design rather than a feature of the memory model itself.

4.2 Verifying an Optimization

A good specification should allow us to abstract away from unnecessary details, so that we can separate reasoning about programs from reasoning about memory models. In this section, we will use the semantics of MiniLLVM with the memory model specification to prove the correctness of a dead store elimination optimization (under any memory model that satisfies the specification). We will assume that we have some analysis for finding dead stores, and prove that removing dead stores does not change the possible behaviors of a MiniLLVM program.

To begin, we need to state our notion of correctness. A correct optimization should *refine* the behaviors of a program; it may remove some behaviors (e.g. by collapsing nondeterminism), but it should never introduce new behaviors.

Definition 4. A configuration is *initial* if it is a tuple (f, p_0, p, env, st, al) such that st and al are empty and p is the start node of P_f. A *trace* of a program P is a sequence of values $v_1, \dots v_n$ for which there is some initial configuration c_0 and some final state (c', m') such that $(c_0, \cdot) \xrightarrow{v_1,\dots,v_n}{}^* (c', m')$. A program P *refines* a program Q if every trace of P is a trace of Q.

We can prove refinement through the well-established technique of *simulation*. In particular, since dead store elimination removes an instruction from the program, we will use *right-option simulation*, in which the original program may take some externally unobservable steps that the transformed program omits.

Definition 5. A relation R on states is a *right-option simulation* between programs P and Q if for any states C_P, C_Q in P and Q respectively, if $R(C_P, C_Q)$ and $P \vdash C_P \xrightarrow{k} C'_P$, then there is a state C'_Q such that $R(C'_P, C'_Q)$ and either

- $Q \vdash C_Q \xrightarrow{k} C'_Q$, or
- $\exists C''_Q.\ Q \vdash C_Q \rightarrow C''_Q$ and $Q \vdash C''_Q \xrightarrow{k} C'_Q$.

Theorem 4. *If there is a right-option simulation between P and Q, then P refines Q.*

Now we can prove that dead store elimination refines program behaviors. We conservatively approximate dead stores by defining them as stores to locations that will never be read again.

Definition 6. An instruction `store` $ty_1\ e_1,\ ty_2{}^*\ e_2$ in a program P is *dead* if in all executions of P, if e_2 is evaluated to a location ℓ when the instruction is executed, then ℓ will not be read for the remainder of the execution.

The optimization itself, given a dead store, is simple: we remove the node containing the dead store from its CFG. The simulation relation relates a state (c', m') in the transformed program to a state (c, m) in the original program if m' can be obtained from m by dropping dead writes and c' can be obtained from c by skipping the removed node n.

Definition 7. Let P be a graph in which the function f contains a node n whose successor is n'. The predicate $\mathsf{skip\_node}$ holds on a pair of configurations (c, c') if either both c and c' are Error, or c' can be obtained from c by replacing all occurrences of n in the program point and the stack with n'. Let R_{dse} be the relation such that $R_{dse}((c', m'), (c, m))$ when either

- $c = \mathsf{Error}$, or
- m' can be obtained from m by removing writes to locations that will not be read from the remainder of the execution, and $\mathsf{skip\_node}(c, c')$ holds.

The proof proceeds as follows. First, we show that any step in the transformed graph can occur in the original graph.

Lemma 1. *Let P' be the program obtained from P by removing a node n from a function f, and n' be the successor of n. If $P' \vdash (c', m) \xrightarrow{k} (c'_2, m_2)$, $\mathsf{skip\_node}(c, c')$, and c is not at n, then there exists c_2 such that $P \vdash (c, m) \xrightarrow{k} (c_2, m_2)$ and $\mathsf{skip\_node}(c_2, c'_2)$.*

Next, we use the memory model axioms to prove that dropping writes to unread locations from a history does not change the operations it allows.

Lemma 2. *Let m and m' be consistent histories such that m is produced by a partial execution of a program P and m' can be obtained from m by removing writes to locations that are not read for the remainder of the execution. If P never reads uninitialized locations or writes to free locations, then $\mathsf{can\_do}(m, op)$ if and only if $\mathsf{can\_do}(m', op)$.*

We can use this lemma to show that the relationship between memories is preserved by program steps.

Lemma 3. *Let m and m' be consistent histories such that m is produced by a partial execution of a program P and m' can be obtained from m by removing writes to locations that are not read for the remainder of the execution. If P never reads uninitialized locations or writes to free locations and $P \vdash (c, m') \xrightarrow{k} (c_2, m'_2)$, then there exists m_2 such that $P \vdash (c, m) \xrightarrow{k} (c_2, m_2)$ and m'_2 can be obtained from m_2 by removing writes to locations that are not read for the remainder of the execution.*

Lemmas 1 and 3 taken together, with a little reasoning about the effects of the dead store, allow us to conclude that R_{dse} is a simulation relation.

Theorem 5. *Let P' be the program obtained from P by removing a dead store, and suppose that P' never reads an uninitialized location and P never writes to a free location. Then R_{dse} is a right-option simulation between P' and P, and so P' refines P.*

Note that since P' has fewer writes than P, it may have more uninitialized locations, and so the condition on reads must be checked on P' and the condition on writes must be checked on P. We can conclude that, for this class of well-behaved programs, the dead store elimination optimization is correct under any memory model that meets the specification.

Comparison with Vellvm Using a more abstract specification should lead to simpler proofs, giving us a more concise formulation of the properties of the memory model and allowing us to avoid reasoning about details of the memory model. The Vellvm project [11] also included a dead store elimination for an LLVM-based language verified in Coq, using a variant of the CompCert memory model, and so provides us a standard with which to compare our proofs. While it is difficult to compare different proof efforts based on different formalizations, several metrics suggest that our specification did indeed lead to significantly simpler proofs. Vellvm's DSE verification consists of about 1860 lines (65k characters) of definitions and proof scripts, while our verification is 890 lines (44k characters). A separate section of Vellvm's code is devoted to lifting CompCert's memory axioms for use in the proofs—essentially the memory model specification for Vellvm—and this section is 1200 lines (38k characters), while our memory model specification is 420 lines (17k characters). To correct for the effects of different proof styles on line and character counts, we also compared the gzipped sizes of the developments; Vellvm's proof is 12.4 kilobytes, our proof is 8.3 kilobytes, Vellvm's specification is 6.7 kilobytes, and our specification is 3.3 kilobytes.

Although Vellvm's language is more featureful than MiniLLVM, this appears to account for very little of the difference in the proofs themselves. Roughly speaking, our proof of correctness is 2/3 the size of Vellvm's and our specification is half the size, supporting the assertion that our specification lends itself to simpler proofs. Furthermore, our results hold not just for one model but for any instance of the specification.

5 Related Work

There have been many efforts to generically specify concurrent and relaxed memory models. The work of Higham et al. [1] is an early example of formalizing memory models in terms of sequences of read and write events; this approach is used to formalize models ranging from linearizability to TSO and PSO. Yang et al. [10] gave axiomatic specifications of six memory models, including some non-operational models, and used constraint logic programming and SMT solving to check whether specific executions adhered to the models. Saraswat et al. [7] gave a simple specification for concurrent memory models in terms of the "well-synchronized programs are sequentially consistent" property, and demonstrated that their specification could be instantiated with both models that prohibited thin-air reads and those that allowed them. In all these works, reads, writes, and synchronizations were assumed to be the only memory operations, and thus "sequential consistency" was taken to uniquely define the single-threaded memory model.

Owens et al. [6] defined the x86-TSO memory model, and showed that their axiomatic definition was equivalent to an abstract-machine model. This model formed the basis for the memory model of CompCertTSO [9], the main inspiration for our work. CompCertTSO's model includes `alloc` and `free` operations, and we follow its approach in giving semantics to our language by combining language steps and memory steps. CompCertTSO does not seek to give a general specification of a category of memory models, but rather a single instance with TSO concurrency and CompCert-specific allocation and free behavior. We know of no other work that attempts to give a generic, language-independent specification of memory models with operations beyond read and write.

6 Conclusions and Future Work

While much work has gone into formalizing the range of possibilities for concurrent memory models, less attention has been devoted to a truly generic description of sequential memory models. Our specification is a first step towards such an account, and we have highlighted the properties of generality, feasibility, and usability that make it a reasonable specification for sequential memory models with allocation and free operations. We have characterized the set of programs for which all such models are equivalent, proved that CompCert's memory model is an instance of our specification, and used it to verify an optimization with proofs demonstrably simpler than those written without such a specification.

The natural next step is to integrate our specification into a framework for concurrent memory models, allowing us to instantiate it with realistic models (such as CompCertTSO) that include allocation and free operations and verify optimizations with respect to those models. We also intend to integrate more memory features into our specification, including structured data and casts between locations and values. Ultimately, we aim to construct a unified specification for operational memory models that can be used to support and simplify any compiler verification effort.

References

1. Higham, L., Kawash, J., Verwaal, N.: Defining and comparing memory consistency models. In: In Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems. pp. 349–356 (1997)
2. Kang, J., Hur, C., Mansky, W., Garbuzov, D., Zdancewic, S., Vafeiadis, V.: A formal C memory model supporting integer-pointer casts. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '15 (to appear)
3. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the international symposium on

Code generation and optimization: feedback-directed and runtime optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004), http://dl.acm.org/citation.cfm?id=977395.977673

4. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. 43(4), 363–446 (Dec 2009), http://dx.doi.org/10.1007/s10817-009-9155-4
5. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. J. Autom. Reason. 41, 1–31 (July 2008), http://dl.acm.org/citation.cfm?id=1388522.1388533
6. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics. pp. 391–407. TPHOLs '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-03359-9_27
7. Saraswat, V.A., Jagadeesan, R., Michael, M., von Praun, C.: A theory of memory models. In: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 161–172. PPoPP '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1229428.1229469
8. Stewart, G., Beringer, L., Cuellar, S., Appel, A.W.: Compositional compcert. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 275–287. POPL '15, ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2676726.2676985
9. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: CompCertTSO: A verified compiler for relaxed-memory concurrency. J. ACM 60(3), 22:1–22:50 (Jun 2013), http://doi.acm.org/10.1145/2487241.2487248
10. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Nemos: a framework for axiomatic and executable specifications of memory consistency models. In: Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International. pp. 31– (April 2004)
11. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. SIGPLAN Not. 47(1), 427–440 (Jan 2012), http://doi.acm.org/10.1145/2103621.2103709