Modular Metatheory for Memory Consistency Models

Laura Effinger-Dean
University of Washington
effinger@cs.washington.edu

Dan Grossman
University of Washington
djg@cs.washington.edu

Abstract

We present a framework based on operational semantics for formalizing shared-memory multithreaded programming languages with relaxed memory consistency models. The key feature of our framework is a division of each system's semantics into two modules, the *program semantics* and the *heap semantics*. This modularization allows elegant, concise, and reusable components. It is trivial to use the same memory model with multiple languages and the same language with multiple memory models.

We demonstrate the effectiveness of our framework by proving several results. First, we give a semantics for the Partial Store Order memory model, and prove that data-race-free programs exhibit sequentially-consistent semantics using this model. Critically, we did not need to define a specific language semantics to prove this result. We assumed only that the language satisfies certain sanity checks easily met by realistic languages. Second, we generalize the notion of type safety to arbitrary memory models, and prove type safety for a multithreaded simply-typed lambda calculus for any *type-preserving* memory model. Finally, we present a model of the MSI cache coherence protocol, and prove that it is semantically equivalent to sequential consistency for any language.

We used the Coq proof assistant to carry out and mechanically verify all our work. Our Coq code is freely available and was modularly designed to support reuse for future extensions and projects.

1. Introduction

Any multithreaded system with shared memory, whether it is a programming language or a multicore processor, must specify a *memory consistency model*, or *memory model*. The memory model defines which values may be returned by a memory-read operation in the course of a given execution. The simplest and most intuitive memory model is *sequential consistency*, which guarantees a global interleaving of memory operations consistent with each thread's program semantics. Unfortunately, sequential consistency is not efficiently implementable for typical programming languages given compiler transformations and modern multiprocessor hardware.

Therefore, most languages have *relaxed memory models*, which weaken the ordering constraints of sequential consistency to allow more legal behaviors. (Worse, many languages give no definition at all, which is a glaring hole in their specification, but parallel implementations are still not sequentially consistent.) Section 2 gives additional background on the need for relaxed memory models and what properties they should have to allow for reasonable language semantics. By far, the most important property is the *data-race-free guarantee*: programs with no data races behave *as if* the memory model were sequential consistency.

1.1 The Need for Modularity

The semantics of relaxed memory models are tricky and subtle, requiring experts to devote great resources to defining them and proving properties. As in any formal endeavor, they carefully model the tricky aspects of what they are studying—possible memory reorderings—while simplifying details that are presumably less important. As such, the "programming language" for most research in this area is just threads emitting traces of memory and synchronization operations. Only recently has there been work formalizing relaxed memory models using operational semantics [8] rather than trace reorderings [2, 3]. (See Section 8 for related work.)

Programming-language semantics are also tricky and subtle, with experts developing well-suited techniques for reasoning about sophisticated language constructs. For example, we have rich type systems and metatheory that lets us prove safety properties. But such research often ignores shared memory, or at best adds threads that communicate via sequentially-consistent memory.

In short, proofs of the data-race-free guarantee and other memory model properties rarely use real language constructs, and proofs of type safety and other language properties rarely use real memory models. This is a disappointing reality for semantics and a potential practical problem given how counterintuitive concurrent interactions can be.

We argue that the solution is *not* heroic efforts to build formal semantics compounding the complexities of both worlds. Rather, we need semantics that are *modular*, in which memory models and the rest of the language can be defined independently, with a precise interface connecting them. This paper presents our *MemModel* framework, which embodies this approach and has allowed us to prove a number of reusable results.

1.2 Our Approach

The key idea of our framework is to divide the semantics of a shared-memory system into halves: the *language model*, which describes the syntax and semantics of the program, and the *memory model*, which describes the syntax and semantics of the heap. The two parts are defined by independent operational semantics that communicate over a clearly-delineated interface. Hence it is trivial to pair any language model with any memory model to produce a system. Section 3 presents this framework in detail.

This decoupling of the language model and the memory model allows much of our formal proof infrastructure to be neither memory-model- nor language-specific. For example, a proof of the data-race-free guarantee for some memory model can state minimal assumptions about the language model and then hold for any language meeting those assumptions. We are unaware of any prior work that applies such results to multiple languages. Conversely, a proof of type safety for a programming language can state minimal assumptions about the memory model. Similarly, we are unaware of prior type-safety results that work for multiple memory models, or even for memory models other than sequential consistency. In our approach, the possibly-complex memory model is held abstract, and the type-safety proof is essentially no more difficult than when assuming sequential consistency.

We call our system a "framework" because it is designed to support future model and proof developments. Our focus has been reusability and separation of concerns, enabling more sophisticated memory models or programming languages later. We have done all our work in Coq, finding Coq's dependently-typed functional kernel language, and in particular its module system, to be an ideal match for stating and proving reusable theorems. Section 7 gives an overview of our publicly-available Coq interfaces.

1.3 Contributions

Our primary contribution is a formal framework and associated Coq library that makes it easy to integrate relaxed memory models into formalisms of non-trivial programming languages and vice-versa.

We have also put our system to work, developing mechanized proofs of theorems that exploit our decoupled approach to state more general results than is possible when studying one programming language with one memory model. Section 4 highlights the benefits of our approach by presenting a proof of the data-race-free guarantee for a write-buffering memory model similar to that used by Boudol and Petri [8]. Section 5 presents a type-safety theorem that makes only very basic assumptions about the memory-model half of the system, providing a template for how to establish safety properties without picking a memory model. Finally, we use MemModel to study a problem closely related to relaxed memory models: the correctness of cache coherence protocols. Section 6 constructs a model of the MSI protocol [10] using our framework and proves its equivalence to sequential consistency while making no assumptions about the language model, essentially proving that MSI caching is correct for any language.

In short, MemModel applies the principles of modularity and abstraction to operational semantics for programming languages with relaxed memory models. We believe our approach improves the state of metatheory for an important and complex area of programming-language semantics.

2. Background: Memory Models for Languages

Too many programming-language users, designers, and implementors are unaware of the need for and subtleties of memory models. A memory model is an essential part of a language's semantics, defining when reads and writes may be reordered. In single-threaded programs, the answer is "never": semantically, each read must "see" the most recent write, but implementations can still reorder accesses to disjoint locations without violating the language's semantics. However, with multiple threads, other threads could observe these reorderings.

The simplest memory consistency model, *sequential consistency* [13], essentially disallows memory reordering: it requires a single global order of memory operations that is consistent with each thread's program execution. Unfortunately, this model is too burdensome for language implementations. First, no widely available multiprocessor guarantees it without using expensive memory fences on all accesses to shared memory. Second, and generally less appreciated, many compiler transformations—including benign concepts like common-subexpression elimination—can have the effect of reordering memory operations.

If not sequential consistency, then what? Relaxed memory models for programming languages must strike a delicate balance of giving a usable language semantics while still allowing compilers, run-time systems, and hardware sufficient latitude. To keep the complexities hidden from most programmers, modern memory models provide the *data-race-free guarantee* originally articulated by Adve for hardware [1, 2]: If every sequentially-consistent program execution is data-race free (basically, no two threads can read/write or write/write the same location simultaneously), then every execution appears to be sequentially consistent. This guarantee places the burden on programmers to avoid races by using synchronization (e.g., locks), promising in return reasoning in terms of memory interleavings. For language implementors, it allows substantial leeway, but essentially prevents reordering memory operations across synchronization operations or performing transformations that create new memory operations that introduce races.

For an unsafe language like C++, the data-race-free guarantee is almost the end of the story, although the memory model has additional features for code that needs data races to provide progress guarantees (e.g., lock-free data structures). If a C++ program has a data race, its semantics is *undefined*, much like a program that has a buffer overflow. This approach is totally unacceptable for safe languages: the semantics must ensure language and programmer abstractions are enforced even for "bad" programs that have data races. Otherwise, an execution could crash or, much worse, compromise security policies and other trusted modules.

Therefore, the data-race-free guarantee is not part of the *definition* of relaxed memory models for safe languages. Rather, it is a *theorem* one proves about a model. Because the models involve subtle concurrent interactions and sets of legal reorderings, the proofs are difficult even for trivial languages like memory traces or basic assembly languages. The Java Memory Model [14] is a notable attempt to provide a usable precise memory model for a safe, high-level language. It is referred to so often not because it is perfect or ideal but because few other language communities have even made an attempt to define a memory model for their language. It is our hope that our work in this paper will make this crucial endeavor less daunting by making memory models a more modular component of language semantics.

Research on memory models in the languages community has centered on imperative languages such as C++ [7], Java [14] and the x86 architecture [19], but functional languages are not immune to the complexities of mutable memory. First, it is common to have threads communicate via shared-memory updates even if most computation is side-effect-free. Second, *implementations* of functional languages must write to read-only objects—e.g., at initialization, to update unevaluated thunks as in Haskell [15], or during garbage collection. Ensuring such imperative updates do not violate program semantics is difficult. For example, the Java Memory Model is substantially complicated by `final` (read-only) fields.

3. The MemModel Framework

This section introduces the MemModel semantic framework. The key to our approach is to divide the semantics of a system into two components: the *heap semantics* and the *program semantics*. Modularizing our semantics this way accelerates the research process and makes theoretical results more easily generalizable. Heaps in our framework are *active*: they may take steps that the program is unaware of. Similarly, the heap is not aware of all program steps. *Mutual* steps are taken by both the heap and the program.

3.1 Preliminaries

Figure 1 presents syntactic constructs shared by all the models discussed in this paper. We assume a program state is a pool of threads, each identified by a thread ID θ. Threads allocate, read, and write heap locations r, and acquire and release mutual-exclusion locks l. For simplicity, locks are not allocated dynamically. The symbol v represents a value, which is language-specific and will be defined in Section 3.3.

Models in our framework are tied together by the notion of *actions*. An action a represents a step that is visible both to the program and to the heap, such as a heap write or a lock acquire. There are six types of actions that a thread may perform: allocating location r with initial value v; reading value v from location r; writing value v to location r; spawning a new thread with ID θ; acquiring a lock l; and releasing a lock l.

$$\begin{array}{rccl}
\text{Thread IDs} & \theta & \in & \mathit{ThreadID} \\
\text{Locations} & r & \in & \mathit{Location} \\
\text{Locks} & l & \in & \mathit{Lock} \\
\text{Mutual actions} & a & ::= & \mathsf{ref}(r,v) \mid \mathsf{rd}(r,v) \mid \mathsf{wr}(r,v) \\
 & & & \mid \mathsf{sp}(\theta) \mid \mathsf{acq}(l) \mid \mathsf{rel}(l) \\
\text{Program actions} & \alpha & ::= & \mathsf{pure} \mid a \\
\text{Effects} & \sigma & ::= & (\theta,\alpha) \mid \epsilon \\
\text{Heap effects} & h & ::= & (\theta,a) \mid \epsilon \\
\text{Program effects} & p & ::= & (\theta,\alpha) \\
\text{Traces} & t & ::= & \cdot \mid t,\sigma
\end{array}$$

Figure 1. Common syntax shared by all models in our framework.

Every step taken by a model is decorated with an *effect* σ, which describes which action the step performed. If thread θ performs action a, then the step's effect is the pair (θ, a). If a step by thread θ has no effect on the heap (e.g., a beta reduction), then the step's effect is (θ, pure). The third type of effect is the *empty effect* ϵ, which corresponds to steps taken by the heap that are invisible to the program (such as flushing a value from a cache).

We further distinguish two types of effects: *heap effects* h and *program effects* p. These effects describe actions "visible to" the memory and language models, respectively. h does not include pure effects, while p does not include the empty effect. Both h and p include mutual effects.

A *trace* t is a list of effects. $\mathsf{erase}(t)$ is the trace obtained by removing all empty effects from t.

Throughout this paper we make use of a generic map type. We write $M : k \Rightarrow v$ for a map M from keys k to values v. The empty map is $[]$. The map in which every key has a default value of v_0 is $[* \mapsto v_0]$. The value stored for key k in map M (if any) is $M(k)$. The new map created by updating map M to hold value v for key k is $M[k \mapsto v]$. The new map created by removing the mapping for key k (if any) from map M is $M|_k$. The domain of a map M is $\mathsf{dom}(M)$. We will assume that two maps M and M' are equal if $\mathsf{dom}(M) = \mathsf{dom}(M')$ and $M(k) = M'(k)$ for all $k \in \mathsf{dom}(M)$.

3.2 Memory Models

Our framework has two distinct kinds of models. The first is the *memory model*, which gives the syntax and semantics for the heap. The heap is responsible for allocating new locations and tracking the values stored at each location, as well as the state of locks.

A memory model has three components:

1. A set of heaps *Heap*. We use the symbol H to represent an element drawn from *Heap*.
2. An initial heap $H_0 \in \mathit{Heap}$.
3. A *heap semantics* with the following judgment form:

$$\boxed{H \xrightarrow{h} H'}$$

The last component, the heap semantics, is a small-step operational semantics for the memory model. Each transition in the semantics is labeled with a heap effect (either a mutual effect or, if the transition is not visible to the program, ϵ).

By design, a memory model is completely oblivious to the program semantics. Instead, the program and the heap communicate via effects. For example, the transition $H \xrightarrow{(\theta,\mathsf{wr}(r,v))} H'$ indicates that, given an initial heap H, a write of v to r by thread θ could yield new heap H'. The program may or may not be able to take such a step; the heap semantics defines all possible heap transitions.

$$\textsc{Alloc}\ \frac{r \notin \mathsf{dom}(S)}{(S;L) \xrightarrow{(\theta,\mathsf{ref}(r,v))} (S[r \mapsto v];L)} \qquad \textsc{Read}\ \frac{S(r) = v}{(S;L) \xrightarrow{(\theta,\mathsf{rd}(r,v))} (S;L)}$$

$$\textsc{Write}\ \frac{}{(S;L) \xrightarrow{(\theta,\mathsf{wr}(r,v))} (S[r \mapsto v];L)} \qquad \textsc{Spawn}\ \frac{}{H \xrightarrow{(\theta,\mathsf{sp}(\theta'))} H}$$

$$\textsc{Acquire}\ \frac{l \notin \mathsf{dom}(L)}{(S;L) \xrightarrow{(\theta,\mathsf{acq}(l))} (S;L[l \mapsto \theta])} \qquad \textsc{Release}\ \frac{L(l) = \theta}{(S;L) \xrightarrow{(\theta,\mathsf{rel}(l))} (S;L|_l)}$$

Figure 2. Sequential consistency as a MemModel memory model.

Example: sequential consistency As an essential example, we define sequential consistency as a MemModel memory model. We model the sequentially-consistent heap as having two components: a store and a lock map. A *store* is a map from locations to values and a *lock map* is a map from locks to thread IDs:

$$\begin{array}{rccc}
\text{Stores} & S & : & r \Rightarrow v \\
\text{Lock maps} & L & : & l \Rightarrow \theta
\end{array}$$

If lock map L has no mapping for lock l, then l is available. If $L(l) = \theta$, then thread θ currently holds l. Threads cannot acquire locks they already hold. Now we can give the syntax for heaps:

$$\begin{array}{rcl}
H & ::= & (S;L) \\
H_0 & = & ([];[])
\end{array}$$

Figure 2 gives the heap semantics, which is entirely standard.

Although memory models can be analyzed without any reference to program semantics (see Section 6), our framework is much more powerful when we combine the heap semantics with a program semantics, as Section 3.3 discusses next.

3.3 Language Models

A *language model* in MemModel gives the syntax and semantics for a programming language. Such a model has three components:

1. A set of values *Value*. We use the symbol v to represent an element drawn from *Value*.
2. A set of program states *ProgramState*. We use the symbol P to represent an element drawn from *ProgramState*.
3. A *program semantics* with the following judgment form:

$$\boxed{P \xrightarrow{p} P'}$$

Typically, a program state P will be a pool of threads indexed by thread ID. However, the interface is flexible enough to accommodate other programming models.

Just as the memory model is not dependent on a particular language model, the language model is oblivious to the syntax and semantics of the heap. For example, the program transition $P \xrightarrow{(\theta,\mathsf{rd}(r,v))} P'$ says that P can take a read step in which thread θ reads v from r, yielding new program state P'. Whether that value is actually readable is determined by the heap semantics; the language model simply describes all possible actions.

The language model as presented thus far does not support type systems, only syntax and dynamic semantics. Section 5 will add type systems to language models in a clean and general way.

Example: multithreaded lambda calculus As a canonical example, we construct a language model for a lambda calculus with

$$\boxed{\begin{array}{l} e \xhookrightarrow{\alpha} e'; \hat{e} \\ P \overset{p}{\rightharpoondown} P' \end{array}} \qquad \textsc{Beta}\ \frac{}{(\lambda x.e)\ v \xhookrightarrow{\mathsf{pure}} e[x/v]; \cdot} \qquad \textsc{Ref}\ \frac{}{\mathsf{ref}\ v \xhookrightarrow{\mathsf{ref}(r,v)} r; \cdot}$$

$$\textsc{Read}\ \frac{}{!r \xhookrightarrow{\mathsf{rd}(r,v)} v; \cdot} \qquad \textsc{Write}\ \frac{}{r := v \xhookrightarrow{\mathsf{wr}(r,v)} (); \cdot} \qquad \textsc{Spawn}\ \frac{}{\mathsf{spawn}\ e \xhookrightarrow{\mathsf{sp}(\theta)} (); e}$$

$$\textsc{Acquire}\ \frac{}{\mathsf{acq}\ l \xhookrightarrow{\mathsf{acq}(l)} (); \cdot} \qquad \textsc{Release}\ \frac{}{\mathsf{rel}\ l \xhookrightarrow{\mathsf{rel}(l)} (); \cdot} \qquad \textsc{App1}\ \frac{e_1 \xhookrightarrow{\alpha} e_1'; \hat{e}}{e_1\ e_1 \xhookrightarrow{\alpha} e_1'\ e_2; \hat{e}} \qquad \cdots$$

$$\textsc{PrgmStep}\ \frac{P(\theta) = e \qquad e \xhookrightarrow{\alpha} e'; \cdot}{P \xrightarrow{(\theta,\alpha)} P[\theta \mapsto e']} \qquad \textsc{PrgmSpawn}\ \frac{P(\theta) = e \qquad e \xhookrightarrow{\mathsf{sp}(\theta')} e'; e'' \qquad \theta' \notin \mathsf{dom}(P)}{P \xrightarrow{(\theta,\mathsf{sp}(\theta'))} P[\theta \mapsto e'][\theta' \mapsto e'']}$$

Figure 3. Multithreaded lambda calculus as a language model in our framework (selected rules omitted).

$$\boxed{H; P \xrightarrow{\sigma} H'; P'} \qquad \textsc{Mutual}\ \frac{H \xrightarrow{(\theta,a)} H' \qquad P \xrightarrow{(\theta,a)} P'}{H; P \xrightarrow{(\theta,a)} H'; P'}$$

$$\textsc{Heap}\ \frac{H \xrightarrow{\epsilon} H'}{H; P \xrightarrow{\epsilon} H'; P} \qquad \textsc{Program}\ \frac{P \xrightarrow{(\theta,\mathsf{pure})} P'}{H; P \xrightarrow{(\theta,\mathsf{pure})} H; P'}$$

Figure 4. MemModel's generic system model.

threads, references, and locks. We first define the syntax for the model, including values *Value* and program states *ProgramState*:

$$\begin{array}{rrcl} \text{Expressions} & e & ::= & () \mid r \mid l \mid \lambda x.e \mid x \mid e\ e \mid \mathsf{ref}\ e \mid !e \\ & & & \mid e := e \mid \mathsf{spawn}\ e \mid \mathsf{acq}\ e \mid \mathsf{rel}\ e \\ \text{Values} & v & ::= & () \mid r \mid l \mid \lambda x.e \\ \text{Optional spawn} & \hat{e} & ::= & \cdot \mid e \\ \text{Program states} & P & : & \theta \Rightarrow e \end{array}$$

The program state maps thread IDs to expressions. Next we define the language semantics in Figure 3. The semantics has a single-threaded judgment that describes legal transitions for one thread (including an option to spawn a new thread), and a top-level judgment that nondeterministically chooses the next thread to run. This semantics is mostly unsurprising, although a few rules are counterintuitive. For example, the read rule picks a value out of thin air: there is no heap to determine the location's current value. Similarly, the acquire and release rules do not enforce any mutual exclusion. These semantic issues are enforced by the heap semantics.

3.4 System Semantics

Figure 4 merges the language and memory models into a complete system model. The system model is parameterized: we can combine any memory model with any language model using the same three rules. This set-up is what lets us "plug" many memory models into a single program model, or vice versa. The system model allows both the heap and the program to step without the other side's cooperation (the HEAP and PROGRAM rules, respectively). However, the MUTUAL rule requires that the two sides cooperate on key transitions—allocation, reading and writing memory, acquiring and releasing locks, or spawning a new thread. A helpful analogy is synchronous message-passing: the program and the heap are independent processes that *rendezvous* to take mutually-relevant steps.

For example, the transition $H; P \xrightarrow{(\theta,\mathsf{rd}(r,v))} H'; P'$ requires the program and heap to cooperate. For typical language models, this transition means the program has a thread θ trying to read location r, and the heap allows such a read to return v. However, the system semantics merely requires that both halves agree that the effect $(\theta, \mathsf{rd}(r, v))$ occurs.

We define $\rightarrow*$ to be the reflexive transitive closure of $\rightarrow$, and concatenate the transition effects to build a trace: $H; P \xrightarrow{t}* H'; P'$.

3.5 Discussion

MemModel's design naturally supports efficient proof development. For example, to establish an equivalence between two memory models, it usually suffices to define the syntax and semantics of the memory models, and leave the language model abstract. That way, we can concentrate on the relevant aspects of the proof without getting bogged down in unnecessary details. At the same time, we avoid making unfounded assumptions about how the language behaves (e.g., which steps can commute). Later, concrete instantiations of the abstracted components may be arbitrarily complex, and the proof still applies provided the proof's explicitly stated assumptions about the abstracted components are established.

The notion of abstraction and modularity is not new, but MemModel's system model is an important instantiation of it. Just as abstract interfaces help software designers avoid dependencies on the internals of a library, abstract models in MemModel help us avoid non-generalizable assumptions about the semantics of the program or the heap, allowing more thorough and general formal results.

4. Proving the Data-Race-Free Guarantee

This section proves the fundamental property—data-race-free programs have sequentially-consistent semantics—for a specific relaxed memory model, which we call the *write-buffering* model. Many details of the model and proof were described by Boudol and Petri [8]. Our goal is not to prove a new result, but rather to show the power of our framework by generalizing an interesting prior result. Using MemModel simplified the proof in key places because we avoided using a specific language semantics. We will note where our development differs from Boudol and Petri's work.

4.1 Write Buffering

We intend to prove the equivalence between two memory models, the first of which is sequential consistency, as described in Section 3. The second (weaker) model is the write-buffering model, which implements the Partial Store Order (PSO) memory model [20]. PSO relaxes the write/read and write/write program orders; that is, writes may be delayed such that they actually commit after reads or writes to other locations by the same thread.

Here is an example of how PSO may affect program semantics, where we assume `x` and `y` are initially `0`.

Thread 1	Thread 2
`x := 1;`	`y := 1;`
`r1 := y;`	`r2 := x;`

In a sequentially-consistent execution, the outcome `r1 = r2 = 0` is impossible. The first action must be a write, so at least one of the threads will see a `1` for their read. However, in the PSO model, the outcome `r1 = r2 = 0` is possible if both threads do their reads before either thread does its write.

First, let us define the syntax for the write-buffering model. The model uses a data structure called a *buffer*. Buffers are maps from

ALLOC
$$\frac{r \notin \mathsf{dom}(S)}{(S;B;L) \xrightarrow{(\theta,\mathsf{ref}(r,v))} (S[r \mapsto v];B;L)}$$

READ1
$$\frac{S(r) = v \qquad B(\theta,r) = \cdot}{(S;B;L) \xrightarrow{(\theta,\mathsf{rd}(r,v))} (S;B;L)}$$

READ2
$$\frac{B(\theta,r) = q,v}{(S;B;L) \xrightarrow{(\theta,\mathsf{rd}(r,v))} (S;B;L)}$$

WRITE
$$\frac{B(\theta,r) = q}{(S;B;L) \xrightarrow{(\theta,\mathsf{wr}(r,v))} (S;B[(\theta,r) \mapsto q,v];L)}$$

SPAWN
$$\frac{\forall r.B(\theta,r) = \cdot}{(S;B;L) \xrightarrow{(\theta,\mathsf{sp}(\theta'))} (S;B;L)}$$

ACQUIRE
$$\frac{l \notin \mathsf{dom}(L)}{(S;B;L) \xrightarrow{(\theta,\mathsf{acq}(l))} (S;B;L[l \mapsto \theta])}$$

RELEASE
$$\frac{L(l) = \theta \qquad \forall r.B(\theta,r) = \cdot}{(S;B;L) \xrightarrow{(\theta,\mathsf{rel}(l))} (S;B;L|_l)}$$

COMMIT
$$\frac{B(\theta,r) = v,q}{(S;B;L) \xrightarrow{\epsilon} (S[r \mapsto v];B[(\theta,r) \mapsto q];L)}$$

Figure 5. Write-buffering semantics.

thread ID/location pairs to lists of values:

$$\begin{array}{llll} \text{Queues} & q & ::= & \cdot \mid q,v \\ \text{Buffers} & B & : & (\theta,r) \Rightarrow q \end{array}$$

Lists of values q are used as FIFO queues: values are enqueued on the right and dequeued from the left. If the least-recently enqueued value in a queue q is v, we can write $q = v, q'$. The empty buffer is $[* \mapsto \cdot]$ (all queues empty). The idea is that writes from each thread will be initially be buffered (delayed), not visible to other threads until they leave the buffer and commit to the global store.[1] Because the pending writes for each location are buffered separately for each thread, writes may bypass other writes. Given the definition of a buffer, we can now give the full form of the write-buffering heap:

$$\begin{array}{lll} H & ::= & (S;B;L) \\ H_0 & = & ([];[* \mapsto \cdot];[]) \end{array}$$

We have simply added a buffer to the sequentially-consistent heap.

Figure 5 gives the heap semantics for the write-buffering model. The ALLOC and ACQUIRE rules are identical to those in Figure 2. Rule READ has been split in two. Rule READ2 states that if the thread has values buffered for that location, the value returned by the read is the most recently buffered value. If the thread's buffer for that location is empty, it sees the store's current value for that location (READ1). The WRITE rule buffers the value onto the appropriate queue, rather than updating the global store. The COMMIT rule, which is the single empty transition, commits the least-recently-enqueued value for a thread ID/location pair back to the store. The RELEASE and SPAWN steps act as fences, forcing the heap to commit any buffered values for the thread before proceeding. Without this synchronization, which is standard for PSO, the data-race-free guarantee does not hold.

For the rest of this section, we will use *WB* and *SC* as abbreviations for write buffering and sequential consistency, respectively.

4.2 Data Race Freedom

First, we must define the notion of a data-race-free program. A *data race* occurs when two threads access the same location, and at least one of those accesses is a write.

Definition 4.1. *Actions α_1 and α_2 form a data race (*$\mathsf{datarace}(\alpha_1, \alpha_2)$*) if there exist r, v_1, v_2 such that one of the following holds:*

1. $\alpha_1 = \mathsf{rd}(r, v_1)$ *and* $\alpha_2 = \mathsf{wr}(r, v_2)$*;*
2. $\alpha_1 = \mathsf{wr}(r, v_1)$ *and* $\alpha_2 = \mathsf{rd}(r, v_2)$*; or*
3. $\alpha_1 = \mathsf{wr}(r, v_1)$ *and* $\alpha_2 = \mathsf{wr}(r, v_2)$*.*

A program is *data-race-free* if no SC executions of the program have concurrent (i.e., simultaneous) actions which form a data race.

[1] We differ slightly from Boudol and Petri in that we do not have a hierarchical program state; all threads and buffers are at a single top level.

Definition 4.2. *A program P_0 is* data-race-free *(*$\mathsf{DRF}(P_0)$*) if, whenever $H_0; P_0 \rightarrow* H_1; P_1 \xrightarrow{(\theta_1,\alpha_1)} H_2; P_2 \xrightarrow{(\theta_2,\alpha_2)} H_3; P_3$ in SC, it is always the case that $\theta_1 = \theta_2$ or* $\neg\mathsf{datarace}(\alpha_1, \alpha_2)$*.*

Our goal is to show that data-race-free programs have sequentially-consistent semantics under WB.

4.3 Proving WB Can Simulate SC

We must prove equivalence in both directions, one of which is much more difficult than the other. Given any SC execution of any program (not necessarily data-race-free), it is possible to simulate that execution in the WB semantics.

Theorem 4.1. *If $H_0; P_0 \xrightarrow{t}* H; P$ in the SC semantics, then there exist H', t' such that $H_0; P_0 \xrightarrow{t'}* H'; P$ in the WB semantics.*

Proof. The proof is by induction on the length of t. At each step, we map the SC heap to the WB heap in which the store and lock map are the same but the buffer is empty. If the SC semantics takes an ALLOC, READ, SPAWN, ACQUIRE, or RELEASE step, we take the corresponding step in the WB semantics (using READ1 for READ steps). If the SC semantics takes a WRITE step, we take a WRITE step followed by a COMMIT step (emptying the buffer) in the WB semantics. Finally, if the program takes a pure step, it can take the same pure step in the WB semantics. □

Note the simple definition of equivalence: both systems must be able to reach the same program state. The exact trace or WB heap is irrelevant, as long as the program observes the same behavior. Note, too, that we did not have to define a language model for this proof; leaving the language abstract was sufficient for our purposes.

4.4 Proving SC Can Simulate WB: The DRF Guarantee

Now we will prove the other half of the equivalence. This half holds only for data-race-free programs—as mentioned in Section 4.1, racy programs may observe non-sequentially-consistent behavior. The proof is very technical and involved, so we omit many intermediate results for brevity.

An advantage of our modular framework is that it is possible to extract large portions of the proof and generalize them to models other than write-buffering. In particular, a key result (Theorem 4.2) uses only the SC model, and thus could be reused for other proofs.

4.4.1 Happens-Before Relation

We first establish a result for SC executions. We will use the following convenient notation: if a trace has the form $\sigma_1, \sigma_2, \ldots, \sigma_n$, then σ_1 has *index* 1, σ_2 has index 2, and so on. We use the notation σ^i to indicate that effect σ has index i (e.g., $t = t_1, \sigma^i, t_2$).

The *happens-before* relation is a strict partial order (i.e., irreflexive, asymmetric and transitive) on indexes in a trace [12]. Happens-before combines program order with synchronization order and thread spawn order.[2]

Definition 4.3. *For all t, if $t = t_1, (\theta_i, \alpha_i)^i, t_2, (\theta_j, \alpha_j)^j, t_3$, then we have $i \ \mathsf{hb}_t \ j$ if one of the following holds:*

1. *$\theta_i = \theta_j$;*
2. *there exists l such that $\alpha_i = \mathsf{rel}(l)$ and $\alpha_j = \mathsf{acq}(l)$; or*
3. *$\alpha_i = \mathsf{sp}(\theta_j)$.*

We further define hb_t^+ to be the transitive closure of hb_t, and say that i happens-before *j in t if $i \ \mathsf{hb}_t^+ \ j$.*

A trace is *well-synchronized* if all data races in the trace are ordered by happens-before.

Definition 4.4. *Trace t is* well-synchronized *if, for all i, j such that $t = t_1, (\theta_i, \alpha_i)^i, t_2, (\theta_j, \alpha_j)^j, t_3$ and $\mathsf{datarace}(\alpha_i, \alpha_j)$, $i \ \mathsf{hb}_t^+ \ j$.*

Given Definition 4.4, we prove the following key result. The judgment $\mathcal{R} \vDash P_0$ is language-specific and will be discussed in more detail in Sections 4.4.3 and 5.3.

Theorem 4.2. *If $\mathsf{DRF}(P_0)$, $\emptyset \vDash P_0$ and $H_0; P_0 \xrightarrow{t}* H; P$ in the SC semantics, then t is well-synchronized.*

The proof is similar to that of Proposition 3.11 in [8]. To summarize, we assume that the trace is not well-synchronized, and therefore there exist two effects in t which (1) form a data race and (2) are not ordered by happens-before. We proceed by induction on the distance between these two effects. If the two actions are adjacent, this clearly violates the DRF assumption. Else, we construct an *equivalent* trace t' (a permutation of t which satisfies $H_0; P_0 \xrightarrow{t'}* H; P$) in which the distance between the two racy effects is strictly smaller, and use the inductive hypothesis to conclude.

4.4.2 DRF Safety Theorem

Given Theorem 4.2, we can proceed with the proof of the main equivalence theorem.

Theorem 4.3. *If $\mathsf{DRF}(P_0)$, $\emptyset \vDash P_0$ and $H_0; P_0 \xrightarrow{t}* H; P$ in the WB semantics, then there exist t', H' such that $H_0; P_0 \xrightarrow{t'}* H'; P$ in the SC semantics.*

The proof of the safety theorem mostly follows the outline given in [8]. At each step of the WB semantics, Theorem 4.2 implies that at most one thread has values buffered for each location (because any writes to the same location by different threads must be separated by a lock release or a thread spawn). Therefore, the WB heap "reduces" to a unique SC heap when the values in the buffer are committed. Moreover, if one thread has values buffered for a location, no other thread may read that location until the thread's buffer has been cleared by a lock release or a thread spawn, so reads always return the correct value for that location.

Our proof most noticeably departs from Boudol/Petri in its assumptions about the language model.

4.4.3 Language Model Assumptions

The proof assumes that the language model satisfies four properties. We have verified that these requirements are correct for a simple language (the lambda calculus given in Figure 3), and we expect them to apply to most "realistic" languages. The set of requirements is revealing in itself as a snapshot of what memory model designers can reasonably expect from a programming language.

[2] Our definition of well-synchronized is more general and closer to the conventional definitions than Boudol and Petri's.

Value-independent reads First, we require that the language not place any restrictions on which values may be returned by a read operation. This requirement is used in the proof of Theorem 4.3.

LR 4.1. *If $P_1 \xrightarrow{(\theta, \mathsf{rd}(r,v))} P_2$, then for all v', there exists P_2' such that $P_1 \xrightarrow{(\theta, \mathsf{rd}(r,v'))} P_2'$.*

This property eliminates some pathological cases, such as a program semantics that checks all reads against a private copy of a sequentially-consistent heap. Note that this rule specifically allows reads to return ill-typed values or values that were never written to the location. The type safety results in Section 5 assuage this concern by proving that a type-preserving memory model (such as WB) never introduces type errors into well-typed programs.

Also note this requirement does not prevent the program semantics from reading the value and then rejecting it, e.g., by transitioning to a dynamic error state. What it prevents is using the identity of the value to decide not to read the value in the first place, like a guarded receive in synchronous message passing.

Program reordering We next require that non-conflicting operations in the program trace must commute. This requirement is crucial in the proof of Theorem 4.2. First, we must define the notion of conflicting program effects:

Definition 4.5. *Two effects p_1 and p_2 conflict ($\mathsf{conflict}(p_1, p_2)$) if one of the following holds:*

1. *$p_1 = (\theta, \alpha_1)$ and $p_2 = (\theta, \alpha_2)$;*
2. *$p_1 = (\theta_1, \mathsf{sp}(\theta_2))$ and $p_2 = (\theta_2, \alpha_2)$; or*
3. *$p_1 = (\theta_1, \alpha_1)$ and $p_2 = (\theta_2, \mathsf{sp}(\theta_1))$.*

LR 4.2. *If $P_1 \xrightarrow{p_1} P_2 \xrightarrow{p_2} P_3$, and $\neg\mathsf{conflict}(p_1, p_2)$, then there exists P_2' such that $P_1 \xrightarrow{p_2} P_2' \xrightarrow{p_1} P_3$.*

The definition of $\mathsf{conflict}$ does not include reads and writes to the same location by different threads (nor acquires/releases of the same lock). Unintuitively, language models can let such operations commute. For instance, suppose $P_1 \xrightarrow{(\theta_1, \mathsf{rd}(r,v_1))} P_2 \xrightarrow{(\theta_2, \mathsf{wr}(r,v_2))} P_3$, where $\theta_1 \neq \theta_2$. It is legal to commute these two operations, even though they access the same location, because the value returned by the read is embedded in the action $\mathsf{rd}(r, v_1)$. Of course, the two operations do not commute in the heap semantics, but this observation reduces the proof burden for the language model.

Fresh thread IDs Next, we require that spawned threads get fresh thread IDs. This property is also needed to prove Theorem 4.2.

LR 4.3. *If $P_0 \xrightarrow{t}* P_1 \xrightarrow{(\theta_1, \mathsf{sp}(\theta_2))} P_2$, then θ_2 does not occur in t.*

Well-formed program states The final requirement states that the program cannot read or write unallocated locations. Here $\mathcal{R}$ is a set of locations, and the judgment $\mathcal{R} \vDash P$ guarantees that the program state P does not reference any locations that are not in $\mathcal{R}$.

LR 4.4. *If $\mathcal{R} \vDash P_1$ and $P_1 \xrightarrow{(\theta, \mathsf{rd}(r,v))} P_2$ (or $P_1 \xrightarrow{(\theta, \mathsf{wr}(r,v))} P_2$), then $r \in \mathcal{R}$.*

Because P is an abstract type, we cannot define $\mathcal{R} \vDash P$ directly. We must instead define this relation for a specific language model and show that it is preserved at each step. We will return to this requirement in Section 5.3.

4.5 Read Speculation Analogy

We conclude this section with an intriguing observation about the relationship between write buffering and another optimization, *read speculation*. Read speculation is an optimization that allows a system to "guess" a value for a location, verifying it after the program

$$\textsc{Alloc}\quad \frac{r \notin \mathsf{dom}(S)}{(S;B;L) \xrightarrow{(\theta,\mathsf{ref}(r,v))} (S[r \mapsto v];B;L)} \qquad \textsc{Read}\quad \frac{B(\theta,r) = q}{(S;B;L) \xrightarrow{(\theta,\mathsf{rd}(r,v))} (S;B[(\theta,r) \mapsto q,v];L)} \qquad \textsc{Write}\quad \frac{B(\theta,r) = \cdot}{(S;B;L) \xrightarrow{(\theta,\mathsf{wr}(r,v))} (S[r \mapsto v];B;L)}$$

$$\textsc{Spawn}\quad \frac{\forall r.B(\theta,r) = \cdot}{(S;B;L) \xrightarrow{(\theta,\mathsf{sp}(\theta'))} (S;B;L)} \qquad \textsc{Acquire}\quad \frac{l \notin \mathsf{dom}(L)}{(S;B;L) \xrightarrow{(\theta,\mathsf{acq}(l))} (S;B;L[l \mapsto \theta])} \qquad \textsc{Release}\quad \frac{L(l) = \theta \qquad \forall r.B(\theta,r) = \cdot}{(S;B;L) \xrightarrow{(\theta,\mathsf{rel}(l))} (S;B;L|_l)} \qquad \textsc{Validate}\quad \frac{S(r) = v \qquad B(\theta,r) = v,q}{(S;B;L) \xrightarrow{\epsilon} (S;B[(\theta,r) \mapsto q];L)}$$

Figure 6. Read speculation semantics.

has already continued execution with that value. Although practical implementations of write buffering and read speculation are very different, we noticed interesting parallels between the two memory models when we defined their semantics in MemModel.

Figure 6 gives the semantics for read speculation (RS). Its syntax is identical to WB, but the buffer queues pending *reads*, instead of pending *writes*. The READ rule pulls a value out of thin air and enqueues it for later validation (cf. rule WRITE in WB). As in WB, an empty step (COMMIT/VALIDATE) dequeues buffer items nondeterministically. Like READ1 in WB, the WRITE rule in RS requires the buffer for the thread and location to be empty. In fact, if we remove from WB rule READ2, which lets threads see their own writes early, the parallels become even more striking. (Deleting READ2 effectively disallows non-atomic writes in PSO, but preserves PSO's instruction reordering.)

Despite the deep similarity between these two models, it is well known that RS does not satisfy the data-race-free guarantee because race-free programs can observe out-of-thin-air values due to self-validating speculation [14].

5. Type Safety with Relaxed Memory Models

In the previous section, we gave an example of how our framework could be used to prove a theorem about a specific memory model that can be generalized to many different language models. This section does the opposite: we prove a theorem—type safety—for a specific language model and generalize it to many different memory models. Furthermore, we provide a blueprint for future type safety proofs by proving a general theorem of type preservation for type systems that satisfy certain properties.

The standard way to prove type safety for a language in the presence of a shared heap is to define a *heap typing* Σ that maps locations to types, then typecheck the program state with respect to this Σ. When a new location is allocated, its type is added to Σ; allocated locations keep the same type for the entire program execution. An important part of the proof is showing that the heap is *well-typed*, and therefore any values read from the heap have the type given by Σ for that location. Typically, this is done by assuming that the heap is fairly simple (i.e., a basic map from locations to values), and proving that each value stored in the heap is well-typed with respect to the current heap typing Σ. (It is necessary to use Σ when typechecking the heap because values stored in the heap may have locations embedded in them.) However, this approach ignores the effect of relaxed memory models. If the heap is more complex than a simple map from locations to values, then the proof of well-typedness for the heap is correspondingly more difficult.

5.1 Type Models

A *type model* specifies the type syntax and typechecking judgments for a type system. Each type model must be defined with respect to a specific language model. A type model has three components:

$$\boxed{\Sigma \rhd a} \qquad \frac{r \notin \mathsf{dom}(\Sigma)}{\Sigma \rhd \mathsf{ref}(r,v)} \qquad \frac{\Sigma(r) = \tau \qquad \Sigma \vdash v : \tau}{\Sigma \rhd \mathsf{rd}(r,v)}$$

$$\frac{}{\Sigma \rhd \mathsf{wr}(r,v)} \qquad \frac{}{\Sigma \rhd \mathsf{acq}(l)} \qquad \frac{}{\Sigma \rhd \mathsf{rel}(l)} \qquad \frac{}{\Sigma \rhd \mathsf{sp}(\theta)}$$

Figure 7. Heap-to-program action typing judgment.

$$\boxed{\Sigma \lhd a : \Sigma'} \qquad \frac{\Sigma \vdash v : \tau}{\Sigma \lhd \mathsf{ref}(r,v) : \Sigma[r \mapsto \tau]} \qquad \frac{\Sigma(r) = \tau \qquad \Sigma \vdash v : \tau}{\Sigma \lhd \mathsf{wr}(r,v) : \Sigma}$$

$$\frac{}{\Sigma \lhd \mathsf{rd}(r,v) : \Sigma} \qquad \frac{}{\Sigma \lhd \mathsf{acq}(l) : \Sigma} \qquad \frac{}{\Sigma \lhd \mathsf{rel}(l) : \Sigma} \qquad \frac{}{\Sigma \lhd \mathsf{sp}(\theta) : \Sigma}$$

Figure 8. Program-to-heap action typing judgment.

1. A set of types *Type*. Elements drawn from *Type* are denoted by τ. Given the set *Type*, we define a *heap typing* $\Sigma : r \Rightarrow \tau$ to be a map from locations to types. Σ gives the types for any locations embedded in a value (e.g., if the value is a lambda abstraction) or in a program state.
2. A *value typing judgment* of the form:
$$\boxed{\Sigma \vdash v : \tau}$$
This typechecks a value to a type under a given heap typing.
3. A *program state typing judgment* of the form:
$$\boxed{\Sigma \models P}$$
This judgment checks that the program state is well-typed under a given heap typing.

Given a typed language model, we define two judgments for actions. The *heap-to-program action judgment* (Figure 7) states that any values read from the heap are well-typed under a given heap typing, and that any locations allocated are fresh. The *program-to-heap action judgment* (Figure 8) states that any values written or allocated by the program are well-typed under a given heap typing. This judgment also yields a new heap typing, which includes the type of the newly allocated location for allocation actions. These two judgments precisely capture the separate responsibilities of the heap and the program with respect to type preservation.

Whenever the program allocates a new location, the heap typing grows. However, the types of existing locations never change, so we say that the new heap typing *extends* the old heap typing.

Definition 5.1. *A heap typing* Σ' extends *another heap typing* Σ ($\Sigma \subseteq \Sigma'$) *if, for all* r, τ *such that* $\Sigma(r) = \tau$, $\Sigma'(r) = \tau$.

The following lemma is easy to prove:

Lemma 5.1. *If $\Sigma \rhd a$ and $\Sigma \lhd a : \Sigma'$, then $\Sigma \subseteq \Sigma'$.*

Example: simply-typed lambda calculus Consider the language model for the lambda calculus previously discussed in Section 3. We can construct a type model for the simply-typed lambda calculus. First, we define *Type*, including types for unit, locks and references in addition to the standard function types:

$$\tau \quad ::= \quad () \mid \mathsf{lock} \mid \mathsf{ref}\ \tau \mid \tau \to \tau$$

Next, we define an expression typing judgment of the form $\Sigma; \Gamma \vdash e : \tau$, where Γ is a map from variables x to types τ. The rules for this judgment are completely standard; here are two examples:

$$\textsc{TAbs}\ \frac{\Sigma; \Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Sigma; \Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \textsc{TLoc}\ \frac{\Sigma(r) = \tau}{\Sigma; \Gamma \vdash r : \mathsf{ref}\ \tau}$$

The value typing judgment is the expression typing judgment with Γ set to the empty map, and the program state typing judgment checks that each expression in the thread pool is well-typed:

$$\frac{\Sigma; [] \vdash v : \tau}{\Sigma \vdash v : \tau} \qquad \frac{\forall \theta \in \mathsf{dom}(P). \Sigma; [] \vdash P(\theta) : \tau}{\Sigma \models P}$$

5.2 Type Preservation

We will now define several requirements for a type model and a memory model that together imply type preservation for the system model as a whole. We do not have a progress result, because the use of locks means that the system could deadlock. First, we define the separate notions of *type-preserving* type and memory models; we then show that the combination of any two such supports type preservation. This result means that proving preservation for a type system is as simple as proving three straightforward lemmas.

Type-preserving language models A typed language model is *type-preserving* if it satisfies the following requirements:

LR 5.1 (Type stability)**.** *If $\Sigma \vdash v : \tau$ and $\Sigma \subseteq \Sigma'$, then $\Sigma' \vdash v : \tau$.*

LR 5.2 (Type preservation for mutual steps)**.** *If $\Sigma \models P$, $P \xrightarrow{(\theta,a)} P'$ and $\Sigma \rhd a$, then there exists Σ' such that $\Sigma \lhd a : \Sigma'$ and $\Sigma' \models P'$.*

LR 5.3 (Type preservation for pure steps)**.** *If $\Sigma \models P$ and $P \xrightarrow{(\theta,\mathsf{pure})} P'$, then $\Sigma \models P'$.*

We omit proofs of these requirements for STLC, but we were able to convert an existing Coq proof of type safety for STLC [6] to this format without much difficulty.

Type-preserving memory models A memory model is *type-enabled* if it provides a *heap typing judgment* of the form:

$$\boxed{\Sigma \Vdash H}$$

For example, we can make sequential consistency type-enabled by defining the heap typing judgment as follows:

$$\frac{\mathsf{dom}(\Sigma) = \mathsf{dom}(S) \qquad \forall r \in \mathsf{dom}(\Sigma). \Sigma \vdash S(r) : \Sigma(r)}{\Sigma \Vdash (S; L)}$$

A type-enabled memory model is *type-preserving* if it satisfies the following requirements:

MR 5.1 (Well-typed initial heap)**.** *The initial heap is well-typed under the empty heap typing: $[] \Vdash H_0$.*

MR 5.2 (Well-typed heap-to-program actions)**.** *If $\Sigma \Vdash H$ and $H \xrightarrow{(\theta,a)} H'$, then $\Sigma \rhd a$.*

MR 5.3 (Type preservation for mutual steps)**.** *If $\Sigma \Vdash H$, $H \xrightarrow{(\theta,a)} H'$ and $\Sigma \lhd a : \Sigma'$, then $\Sigma' \Vdash H'$.*

$$\textsc{Alloc}\ \frac{r \notin \mathsf{dom}(S)}{S \xrightarrow{(\theta,\mathsf{ref}(r,v))} S[r \mapsto v]} \qquad \textsc{Read}\ \frac{S(r) = v}{S \xrightarrow{(\theta,\mathsf{rd}(r,v))} S}$$

$$\textsc{Write}\ \frac{r \in \mathsf{dom}(S)}{S \xrightarrow{(\theta,\mathsf{wr}(r,v))} S[r \mapsto v]} \qquad \textsc{Spawn}\ \frac{}{S \xrightarrow{(\theta,\mathsf{sp}(\theta'))} S}$$

Figure 9. Sequential consistency (locks omitted).

MR 5.4 (Type preservation for empty steps)**.** *If $\Sigma \Vdash H$ and $H \xrightarrow{\epsilon} H'$, then $\Sigma \Vdash H'$.*

Proving these requirements for SC is straightforward. We have also proved that the WB model from Section 4 is type-preserving, which was only slightly more difficult.

Preservation theorem We can now prove type preservation for any combination of type-preserving language and memory models.

Theorem 5.2. *If $\Sigma \Vdash H$, $\Sigma \models P$ and $H; P \xrightarrow{\sigma} H'; P'$, then there exists Σ' such that $\Sigma \subseteq \Sigma'$, $\Sigma' \Vdash H'$ and $\Sigma' \models P'$.*

Proof. We proceed by case analysis on σ.

- $\sigma = (\theta, a)$. Then $H \xrightarrow{(\theta,a)} H'$ and $P \xrightarrow{(\theta,a)} P'$. By MR 5.2, we have $\Sigma \rhd a$. By LR 5.2, there exists some Σ' (our witness) such that $\Sigma \lhd a : \Sigma'$ and $\Sigma' \models P'$. By Lemma 5.1, we have that $\Sigma \subseteq \Sigma'$. Finally, by MR 5.3, we have that $\Sigma' \Vdash H'$.
- $\sigma = (\theta, \mathsf{pure})$. Then $H = H'$ and $P \xrightarrow{(\theta,\mathsf{pure})} P'$. Let $\Sigma' = \Sigma$ (note that $\Sigma \subseteq \Sigma'$ and $\Sigma' \Vdash H'$ trivially). By LR 5.3, $\Sigma' \models P'$.
- $\sigma = \epsilon$. Then $P = P'$ and $H \xrightarrow{\epsilon} H'$. Let $\Sigma' = \Sigma$ (note that $\Sigma \subseteq \Sigma'$ and $\Sigma' \models P'$ trivially). By MR 5.4, $\Sigma' \Vdash H'$. □

Finally, we show that type preservation holds over many steps.

Corollary 5.3. *If $[] \models P_0$ and $H_0; P_0 \to\!* H; P$, then there exists Σ such that $\Sigma \Vdash H$ and $\Sigma \models P$.*

Proof. Follows from MR 5.1 and Theorem 5.2. □

This result demonstrates the true power of having a parameterizable system model. We have given a set of simple requirements which together imply type preservation for a system—and the proof of the type preservation is *oblivious* to the syntax and semantics of the language, type, and memory models of the system.

5.3 Well-Formed Program States

Recall that the DRF proof in Section 4 assumed that the language satisfied a *well-formedness* requirement (LR 4.4). Specifically, the language model defines a judgment $\mathcal{R} \models P$, where $\mathcal{R} \subseteq$ *Location*. When introducing this requirement, we alluded to the idea that well-formedness needed to be preserved at each step of the program. Now that we have defined type-preserving language and memory models, it is clear that *well-formedness is a type system* with exactly one type, and $\mathcal{R}$ is simply the domain of the heap typing Σ. Therefore, in order for the DRF proof to be valid for a language model, there must be some type model for that language such that the model is type-preserving (as defined in Section 5.2). For convenience, we allow the type model to be arbitrarily complex, although a simple type system that checks program states for unknown locations is sufficient for the purposes of the DRF proof. Moreover, we require that the initial program P_0 be well-typed with respect to the empty heap typing: $[] \models P_0$.

$$\textsc{Alloc}\ \frac{r \notin \mathsf{dom}(S)}{(S;C) \xrightarrow{(\theta,\mathsf{ref}(r,v))} (S[r \mapsto v];C)} \qquad \textsc{Read}\ \frac{C(\theta,r) = (c,v)}{(S;C) \xrightarrow{(\theta,\mathsf{rd}(r,v))} (S;C)} \qquad \textsc{Write}\ \frac{C(\theta,r) = (\mathsf{M},v)}{(S;C) \xrightarrow{(\theta,\mathsf{wr}(r,v'))} (S;C[(\theta,r) \mapsto (\mathsf{M},v')])} \qquad \textsc{Spawn}\ \frac{}{(S;C) \xrightarrow{(\theta,\mathsf{sp}(\theta'))} (S;C)}$$

$$\textsc{InvalidToShared}\ \frac{S(r) = v \qquad (\theta,r) \notin \mathsf{dom}(C) \qquad \forall\theta',v'.C(\theta',r) \neq (\mathsf{M},v')}{(S;C) \xrightarrow{\epsilon} (S;C[(\theta,r) \mapsto (\mathsf{Sh},v)])} \qquad \textsc{InvalidToModified}\ \frac{S(r) = v \qquad \forall\theta'.(\theta',r) \notin \mathsf{dom}(C)}{(S;C) \xrightarrow{\epsilon} (S;C[(\theta,r) \mapsto (\mathsf{M},v)])} \qquad \textsc{SharedToModified}\ \frac{C(\theta,r) = (\mathsf{Sh},v) \qquad \forall\theta'.\theta \neq \theta' \rightarrow (\theta',r) \notin \mathsf{dom}(C)}{(S;C) \xrightarrow{\epsilon} (S;C[(\theta,r) \mapsto (\mathsf{M},v)])}$$

$$\textsc{SharedToInvalid}\ \frac{C(\theta,r) = (\mathsf{Sh},v)}{(S;C) \xrightarrow{\epsilon} (S;C|_{(\theta,r)})} \qquad \textsc{ModifiedToInvalid}\ \frac{C(\theta,r) = (\mathsf{M},v)}{(S;C) \xrightarrow{\epsilon} (S[r \mapsto v];C|_{(\theta,r)})} \qquad \textsc{ModifiedToShared}\ \frac{C(\theta,r) = (\mathsf{M},v)}{(S;C) \xrightarrow{\epsilon} (S[r \mapsto v];C[(\theta,r) \mapsto (\mathsf{Sh},v)])}$$

Figure 10. Modified-Shared-Invalid (MSI) cache coherence protocol.

$$\textsc{ReadShared}\ \frac{\mathbb{S}(r) = ((\mathsf{Sh},\Theta),v) \qquad \theta \in \Theta}{\mathbb{S} \xrightarrow{(\theta,\mathsf{rd}(r,v))} \mathbb{S}} \qquad \textsc{ReadModified}\ \frac{\mathbb{S}(r) = ((\mathsf{M},\theta),v)}{\mathbb{S} \xrightarrow{(\theta,\mathsf{rd}(r,v))} \mathbb{S}} \qquad \textsc{WriteModified}\ \frac{\mathbb{S}(r) = ((\mathsf{M},\theta),v)}{\mathbb{S} \xrightarrow{(\theta,\mathsf{rd}(r,v'))} \mathbb{S}[r \mapsto ((\mathsf{M},\theta),v')]} \qquad \textsc{InvalidToModified}\ \frac{\mathbb{S}(r) = ((\mathsf{Sh},\emptyset),v)}{\mathbb{S} \xrightarrow{\epsilon} \mathbb{S}[r \mapsto ((\mathsf{M},\theta),v)]} \qquad \ldots$$

Figure 11. A representative subset of the inverted MSI (I-MSI) semantics.

6. Cache Coherence Protocol Verification

We now explore an alternate use of MemModel: verifying cache coherence protocols. Although we designed our framework with relaxed memory models in mind, it is equally suitable for verifying the correctness of optimizing implementations of sequential consistency. To that end, we define a semantics for MSI [10], a well-known cache coherence protocol, and prove its equivalence to SC.

For the purposes of this section, we have omitted locks from our semantics. Figure 9 gives the semantics for sequential consistency without locks. We do include thread spawn, so that threads have a means of communicating when starting with an empty heap. For simplicity, we add the hypothesis $r \in \mathsf{dom}(S)$ to the WRITE rule; this is a small cheat that allows us to avoid having to define a well-formedness type system as in Section 4.

6.1 MSI Protocol

The Modified-Shared-Invalid (MSI) protocol is a cache coherence protocol with three possible states for each entry in a cache:

- Modified: This cache's copy of this location is the only copy. Updates are allowed (they will be written back to the global store before any other thread is allowed to access the location).
- Shared: This cache has one of possibly many read-only copies of this location.
- Invalid: This cache does not have a copy of this location.

In our model, each thread has its own cache, and each cache can hold an arbitrary number of values. We model the cache as a single global structure that maps TID/location pairs to values. Each value in the cache is tagged with a cache state M (modified) or Sh (shared). If a location is not cached for a given thread, then it is implicitly invalid. The heap consists of a store and a cache.

$$\begin{array}{rcl} c & ::= & \mathsf{M} \mid \mathsf{Sh} \\ C & : & (\theta,r) \Rightarrow (c,v) \\ H & ::= & (S;C) \\ H_0 & = & ([];[]) \end{array}$$

Figure 10 gives the heap semantics for the MSI protocol. All cache state changes happen nondeterministically. For example, the INVALIDTOSHARED rule states that a read-only copy of a value may be stored in the cache at any time, given that no other threads have modified copies of that value.

6.2 Proving MSI Can Simulate SC

First, we prove that MSI can simulate SC. The key is showing that any SC step can be simulated in the MSI semantics such that the cache starts and ends empty.

Lemma 6.1. *If $S \xrightarrow{(\theta,a)} S'$ in the SC semantics, then there exists t such that $(S;[]) \xrightarrow{t}* (S';[])$ and $\mathsf{erase}(t) = (\theta,a)$.*

Proof. If the step was a READ or WRITE, we load the location into the cache, do the operation, then flush the value from the cache. For ALLOC or SPAWN, we take the corresponding MSI step. ☐

6.3 Proving SC Can Simulate MSI

Although it is possible to show directly that SC can simulate MSI, we found it much cleaner to go through an intermediate semantics. This third semantics, which we call *inverted MSI* or *I-MSI*, inverts the structure of the MSI heap by replacing caches with centralized information for each location, including which threads own "copies." Global ownership information considerably simplifies the heap invariant needed to push the proof through (Definition 6.1).

An *ownership store* $\mathbb{S}$ is a store in which the value for each location is tagged with an *ownership state* o. For the MSI protocol, this ownership state is either (Sh,Θ) (shared by a set of threads Θ) or (M,θ) (modified by a thread θ).

$$\begin{array}{rccl} \text{TID sets} & \Theta & \subseteq & \textit{ThreadID} \\ \text{Ownership states} & o & ::= & (\mathsf{M},\theta) \mid (\mathsf{Sh},\Theta) \\ \text{Ownership stores} & \mathbb{S} & : & r \Rightarrow (o,v) \end{array}$$

A location that is invalid for all threads has ownership state $(\mathsf{Sh},\emptyset)$. The heap for I-MSI consists of an ownership store $\mathbb{S}$. Figure 11 gives a representative subset of the I-MSI semantics. This semantics is very similar to the MSI semantics, except that updates are made directly to the global store, and the empty transitions simply update the ownership state of the location appropriately, rather than loading values into and flushing values out of the cache.

I-MSI can simulate MSI The key to the proof is showing that, at each step, the MSI heap is *coherent*—at most one thread holds a modified copy of each location, read-only copies have the correct value, etc. The following definition formalizes heap coherency.

Definition 6.1. *A MSI heap $(S; C)$ is* coherent *with respect to an I-MSI heap $\mathbb{S}$ ($\mathsf{coherent}((S;C),\mathbb{S})$) if, for all r, the following properties hold:*

1. *If $r \notin \mathsf{dom}(\mathbb{S})$, then $r \notin \mathsf{dom}(S)$ and $\forall\theta.(\theta, r) \notin \mathsf{dom}(C)$.*
2. *If $\mathbb{S}(r) = ((\mathsf{Sh}, \Theta), v)$ then $S(r) = v$ and for all θ, either $\theta \in \Theta$ and $C(\theta, r) = (\mathsf{Sh}, v)$ or $\theta \notin \Theta$ and $(\theta, r) \notin \mathsf{dom}(C)$.*
3. *If $\mathbb{S}(r) = ((\mathsf{M}, \theta), v)$, then $r \in \mathsf{dom}(S)$, $C(\theta, r) = (\mathsf{M}, v)$, and $\forall\theta' \neq \theta.(\theta', r) \notin \mathsf{dom}(C)$.*

The next lemma establishes that any MSI step that starts with a coherent heap may be simulated in the I-MSI semantics.

Lemma 6.2. *If $\mathsf{coherent}((S;C),\mathbb{S})$ and $(S;C) \xrightarrow{h} (S';C')$, then there exists $\mathbb{S}'$ such that $\mathbb{S} \xrightarrow{h} \mathbb{S}'$ and $\mathsf{coherent}((S';C'),\mathbb{S}')$.*

The proof (omitted) is by cases on the rule used for $(S;C) \xrightarrow{h} (S';C')$. We always take the corresponding step in the I-MSI semantics (choosing READSHARED or READMODIFIED for reads, as appropriate). The difficulties are (1) satisfying the hypotheses for each step and (2) showing that heap coherency is preserved.

SC can simulate I-MSI The final step is showing that I-MSI executions can be simulated in SC. We translate an I-MSI heap to a SC heap by erasing the ownership state for each location.

Definition 6.2. *For all $\mathbb{S}$, we define $\mathsf{ToStore}(\mathbb{S})$ to be the store S such that $\mathsf{dom}(\mathbb{S}) = \mathsf{dom}(S)$ and if $\mathbb{S}(r) = (o, v)$, then $S(r) = v$.*

The next lemma shows that I-MSI steps can be simulated in SC.

Lemma 6.3. *Any I-MSI step may be simulated in SC:*

1. *If $\mathbb{S} \xrightarrow{(\theta,a)} \mathbb{S}'$, then $\mathsf{ToStore}(\mathbb{S}) \xrightarrow{(\theta,a)} \mathsf{ToStore}(\mathbb{S}')$.*
2. *If $\mathbb{S} \xrightarrow{\epsilon} \mathbb{S}'$, then $\mathsf{ToStore}(\mathbb{S}) = \mathsf{ToStore}(\mathbb{S}')$.*

Proof. For mutual steps, we always take the corresponding SC step (using READ for both shared and modified read steps). Empty steps in I-MSI modify only ownership states, not values, so the second half of the lemma is trivial. □

6.4 Equivalence Theorem

The equivalence theorem follows from Lemmas 6.1–6.3.

Theorem 6.4. *The MSI, I-MSI and SC semantics are equivalent:*

1. *If $H_0; P_0 \rightarrow\!* H; P$ in the SC semantics, then there exists H' such that $H_0; P_0 \rightarrow\!* H'; P$ in the MSI semantics.*
2. *If $H_0; P_0 \rightarrow\!* H; P$ in the MSI semantics, then there exists H' such that $H_0; P_0 \rightarrow\!* H'; P$ in the I-MSI semantics.*
3. *If $H_0; P_0 \rightarrow\!* H; P$ in the I-MSI semantics, then there exists H' such that $H_0; P_0 \rightarrow\!* H'; P$ in the SC semantics.*

From Theorem 6.4 we can conclude that SC and MSI are equivalent—i.e., the MSI protocol is correct. As in Section 4, this result is very general: *any* language, if implemented using a MSI-style cache coherence scheme, supports sequential consistency.

7. Coq Mechanization

All the results in this paper have been fully mechanized using the Coq proof assistant [16], for which the specification language is the dependently-typed functional language Gallina. Gallina's module system was ideal for implementing parameterizable language and memory models. We found that the mechanization process was invaluable in ironing out details of the models and eliminating subtle problems. Figure 12 shows the basic Coq interface for MemModel, which is very similar to the interface described in Section 3.

Var types We use a single datatype, `var`, to represent thread IDs, locations, and locks. This datatype is defined in the Metatheory library [6]. In addition to the `var` type, we used many other features of this library for our proofs, including the implementation of the simply-typed lambda calculus and various convenient proof tactics.

Action types The type `action` in Figure 12 corresponds to the α type in our formalism. `action` is parameterized over some value type `val` (this type will be defined later by the language model).

Models as module types Coq was especially well-matched to this problem because "language models" and "memory models" are naturally implemented as Coq modules. The Coq module types `LANGUAGE_MODEL` and `MEMORY_MODEL` represent the interfaces for language and memory models, respectively. Notice `MEMORY_MODEL` takes a `LANGUAGE_MODEL` as a parameter; this is because the language model defines a `value` type (*Value* in the formalism) which is used in `MEMORY_MODEL`. Because the language model is passed as a parameter, it is an abstract type; that is, a memory model cannot access the internal implementation of a language. One subtle point is that we must forbid a language model from taking a heap-only step and vice-versa. The type definitions for language and memory models accomplish this via axioms. The module `SystemModel` implements the system semantics given in Figure 4.

Defining a model To implement a specific language or memory model, we simply declare it as a subtype of the appropriate module type and define all the required interface components. For example,

```
Module SC (L : LANGUAGE_MODEL) <: MEMORY_MODEL L.
  Definition heap := VarMap.t L.val * VarMap.t tid.
  ...
```

Notice that `SC` is parameterized by a language model. Coq automatically checks that `SC` matches the `MEMORY_MODEL` interface. Here we use the `VarMap` type for maps, which instantiates the finite-map type in Coq's standard library. We use these maps to represent stores, lock maps, buffers, caches, etc.

Instantiating the models To use a memory model, we need to instantiate it with a language model. To develop a proof that does not assume a particular language, we simply declare that a language model exists. For example, for the proof in Section 4, we declare:

```
Declare Module L : LANGUAGE_MODEL.
Declare Module LangReqs :
  LANGUAGE_REQUIREMENTS with Module L := L.
Module SC := SC L.
Module WB := WB L.
```

The two memory models of interest use the same language model, but we are not required to actually define a language model (although it was useful to define one for testing purposes). The `LANGUAGE_REQUIREMENTS` module type defines the four language requirements described in Section 4.4.3.

Type models Figure 13 gives another example of our Coq code: the interface for the type models from Section 5. Again, we use module types for key interfaces, in this case type models (`TYPE_MODEL`) and type-enabled memory models (`TYPED_MEMORY_MODEL`). Notice `TYPE_MODEL` declares a language model `L` as part of its interface; unlike the language parameter for module types like `MEMORY_MODEL`, `L` is concrete and not abstract (i.e., `TYPE_MODEL` has access to the internal implementation of `L`).

```
Definition loc := var.
Definition lock := var.
Definition tid := var.

Section Effects.
  Variable val : Set.

  Inductive action : Set :=
  | pure : action
  | ref  : loc  -> val -> action
  | rd   : loc  -> val -> action
  | wr   : loc  -> val -> action
  | acq  : lock -> action
  | rel  : lock -> action
  | sp   : tid  -> action.

  Definition effect := option (tid * action).
End Effects.

Module Type LANGUAGE_MODEL.
  Parameter val : Set.
  Parameter program_state : Set.
  Parameter program_step : program_state
    -> effect val -> program_state -> Prop.
  Axiom no_empty_steps : forall P P',
    ~ program_step P None P'.
  ...
```

```
Module Type MEMORY_MODEL (L : LANGUAGE_MODEL).
  Parameter heap : Set.
  Parameter empty_heap : heap.
  Parameter heap_step : heap -> effect L.val
    -> heap -> Prop.
  Axiom no_pure_steps : forall H th H',
    ~ heap_step H (Some (th, pure)) H'.
  ...

Module SystemModel (L : LANGUAGE_MODEL)
                   (M : MEMORY_MODEL L).
  Inductive system_step : M.heap
    -> L.program_state -> effect L.val
    -> M.heap -> L.program_state -> Prop :=
  | system_step_mutual : forall H P th a H' P',
    M.heap_step H (Some (th, a)) H' ->
    L.program_step P (Some (th, a)) P' ->
    system_step H P (Some (th, a)) H' P'
  | system_step_heap : forall H H' P,
    M.heap_step H None H' ->
    system_step H P None H' P
  | system_step_program : forall H P P' th,
    L.program_step P (Some (th, pure)) P' ->
    system_step H P (Some (th, pure)) H P'.
  ...
```

Figure 12. Coq implementation of the semantic framework discussed in Section 3 (truncated for space).

```
Module Type TYPE_MODEL.
  Declare Module L : LANGUAGE_MODEL.
  Parameter typ : Set.
  Definition heap_typing := Map.t typ.
  Parameter typing_val : heap_typing -> val
    -> typ -> Prop.
  Parameter typing : heap_typing -> program_state
    -> Prop.
  ...

Module Type TYPED_MEMORY_MODEL
  (L : LANGUAGE_MODEL)
  (T : TYPE_MODEL with Module L := L).
  Declare Module M : MEMORY_MODEL L.
  Parameter well_typed : T.heap_typing -> M.heap
    -> Prop.
  ...
```

Figure 13. Typed language and memory models (Section 5).

TYPED_MEMORY_MODEL takes a type model T as a parameter (meaning that the type model is abstract) and declares a concrete memory model M. Therefore, typed memory models are specific to a single memory model, but parameterizable by any type model.

8. Related Work

There have been several complementary efforts to develop formal semantics for relaxed memory models.

A prominent example is the Java Memory Model, the original presentation of which included a detailed formalism and a paper proof of the data-race-free guarantee [14]. Subsequent efforts have used proof mechanization to identify semantic problems [4, 5]. We also advocate using a proof assistant, as memory models can be difficult to reason about formally, but we have focused on putting the pieces in place for a reusable and modular library.

Boudol and Petri [8] approached relaxed memory models from the perspective of operational semantics. They modeled the relaxation of write/write and write/read ordering by adding write buffers to an operational semantics for multithreaded lambda calculus. The operational approach has the advantage of having a full program semantics, which makes it easier to reason about whether a program is data-race-free. We use several ideas from this paper; in particular, the case study in Section 4 follows the paper's proof closely. Our work is mechanically verified, which we found invaluable, and separates the language and memory models.

Sarkar et al. [19] give an exhaustive formalization of the x86 memory model, which had previously been described only informally. The model and several key proofs were formalized in HOL. The main formalization is in terms of axiomatic constraints over an event structure. A second paper [17] proposes an alternate model, x86-TSO. In addition to the axiomatic model for x86-TSO, they provide an equivalent "abstract machine" (i.e., operational) semantics, which, like memory models in our framework, specifies labeled transitions between machine states (registers, buffers, and memory). They too observed that this operational formulation could be applied to many different memory models, and that the labels (like our actions) could be used to communicate with the other "half" of the system. We consider it a good sign this project, which is state of the art for formal hardware memory models, also concluded that it was useful to define the semantics of the memory in a modular, operational way. Our work, while similarly structured on the memory-model side, differs in key ways. First, we also generalize the notion of the program semantics, while the x86-TSO model is defined only for a concrete event structure. Moreover, this event structure does not have pure steps, so even processor-local data like registers must be stored in the machine state in order to track data dependencies. Second, our focus was on building a generalizable framework to study problems like type safety from a memory model perspective, while theirs was on defining a specific model in a precise and tractable form. The statement of our theorem in Section 4 also differs: we proved the data-race-free guarantee for an operational model not unlike x86-TSO, while their proof is only for the axiomatic version of the similar x86-CC model, and only with respect to the event structure (i.e., trace) of a program.

Saraswat et al [18] tackle the problem of how to show compiler optimizations are sound in the context of shared-memory multi-threading. Their RAO framework models programs as DAGs of atomic steps. Given this DAG formulation, it is possible to define the notion of a safe program transformation in a very general way. Their framework is parameterizable in the sense that a language can specify its own set of primitive atomic steps, and the memory model can specify custom code transformations for a given language.

The Concurrent C Minor project [11] advocates the use of Coq for verifying concurrent operational semantics. In our terms, their work is entirely within the language model, using separation logic techniques to modularize the sequential and concurrent features of a language. They require programs to be data-race-free, leaving the semantics of racy programs undefined. This approach *assumes* that the system's memory model has the data-race-free guarantee.

Frigo and Luchangco [9] formalized memory models as mappings from write nodes to read nodes in a graph of the program trace. The program model supports fork-join parallelism, so it has no explicit thread IDs, instead modeling computations as graphs where edges represent happens-before dependencies. They modeled a memory model as an online process that assigns sources to read nodes as an adversary reveals the program one node at a time. This concept is similar to our model in that the program semantics are abstract and the memory model must fulfill memory requests as the program executes, rather than seeing a full trace.

9. Future Work

There are several interesting directions in which to expand this work. We hope to create more MemModel models for various relaxed memory models and languages, including more complex memory models (e.g., release consistency or the Java Memory Model). As demonstrated in Section 6, the framework is also useful for non-memory model problems such as cache coherence. We have already begun work on an extension of MemModel that supports primitives for software transactional memory.

As presented, MemModel is somewhat rigid in the set of actions it supports, particularly that the only synchronization is via mutual-exclusion locks. We believe this rigidity can be avoided by creating an *action model* shared by the memory and language models of a system. This change would let model-builders specify their own sets of actions, divided into three categories: mutual actions, heap actions, and program actions.

10. Conclusion

The purpose of MemModel is to provide a framework that makes it easier to formally model and reason about programming languages with relaxed memory models. We have shown how to prove results about memory models while making minimal assumptions about program semantics and prove results about program semantics while making minimal assumptions about memory models.

A practical advantage of our approach is that it quite literally takes up less space, whether in a research paper or a Coq development, because the parameterized system model lets us define a program or heap semantics once, then plug it into our proofs where needed. This is very much an example of "code reuse," with all its inherent benefits—faster development, fewer errors, and more easily-understandable code. Our experience indicates that MemModel facilitates rapid prototyping. The clear interface between the program and heap makes it easy to develop the more important half of the semantics before making less critical decisions about, say, the syntax of the other half. The modular framework lets experts focus on the side of importance to them.

The parameterization in our framework makes it easier to generalize results. The equivalence results in Sections 4 and 6 do not assume a particular language model. In fact, abstracting the language model in Section 4 let us highlight the exact parts of the proof that rely on some aspect of the program semantics. Instead of burying these properties in a lemma, we were able to enumerate them in Section 4.4.3, thus characterizing minimal standards a language must meet to support write-buffering. Similarly, the type-safety results in Section 5 demonstrate how we can enumerate exactly the assumptions that a type-safety proof makes about a mutable heap.

In conclusion, MemModel uses modularity to improve the status quo of formal semantics for memory models. Models in the framework are concise yet general. We believe MemModel is a novel platform for developing realistic operational semantics.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[2] S. V. Adve and M. Hill. Weak ordering—a new definition. In *ACM IEEE International Symposium on Computer Architecture*, 1990.

[3] Arvind and J.-W. Maessen. Memory model = instruction reordering + store atomicity. In *ACM IEEE International Symposium on Computer Architecture*, 2006.

[4] D. Aspinall and J. Šěvčík. Formalising Java's data race free guarantee. In *International Conference on Theorem Proving in Higher Order Logics*, 2007.

[5] D. Aspinall and J. Šěvčík. Java memory model examples: Good, bad and ugly. In *International Workshop on Verification and Analysis of Multi-threaded Java-like Programs*, 2007.

[6] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM Symposium on Principles of Programming Languages*, 2008.

[7] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[8] G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *ACM Symposium on Principles of Programming Languages*, 2009.

[9] M. Frigo and V. Luchangco. Computation-centric memory models. In *ACM Symposium on Parallellism in Algorithms and Architectures*, 1998.

[10] J. Handy. *The Cache Memory Book*. Academic Press Inc., 1998.

[11] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. 2008.

[12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[14] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *ACM Symposium on Principles of Programming Languages*, 2005.

[15] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *ACM SIGPLAN International Conference on Functional Programming*, 2009.

[16] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2009. `http://coq.inria.fr`, Version 8.2.

[17] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *International Conference on Theorem Proving in Higher Order Logics*, 2009.

[18] V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[19] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreeen, and J. Algave. The semantics of x86-CC multiprocessor machine code. In *ACM Symposium on Principles of Programming Languages*, 2009.

[20] SPARC International, Inc. The SPARC architecture manual, v. 8, 1992. Revision SAV080SI9308.