A Formalization and Logic for Java Opaque Mode

Author: Please provide author information

Abstract

Recently the Java memory model was expanded with additional "access modes" that allow fine-grained control of lock-free concurrent program behavior. The weakest of these is Opaque Mode, which aims to provide a minimum of semantic guarantees to the programmer and a maximum of optimization opportunities to the architecture and compiler. In contrast to other memory models, Java Opaque Mode forbids causal cycles and so called “thin-air” reads. Further, it lacks a total order on writes to the same location. The absence of thin-air reads is an opportunity to define a simple semantic model, while the lack of a total order on writes is a challenge for reasoning because it invalidates existing state-based approaches.

In this paper we present the first formalization of Java Opaque mode, in the form of an operational semantics. Our semantics has a modest number of rules and supports a stateless form of reasoning called write elimination. In an effort to simplify proofs, we also present a program logic that we prove sound with respect to the semantics. We have used our logic to prove correctness of a bounded memory queue that models a RingBuffer from Linux and we have machine-checked the proof in Coq.

2012 ACM Subject Classification Software and its engineering → General programming languages; Theory of computation → Semantics and reasoning; Concurrency

Keywords and phrases Concurrency, Correctness, Memory Models, Verification

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

As progress on improving single threaded performance has slowed, programmers have turned their focus toward concurrent algorithms. Writing these algorithms and proving them correct is notoriously difficult even when the semantic model is relatively simple.

For example, there is a large body of research on verifying concurrent algorithms where the model is *sequential consistency* [7, 10, 11, 12, 28, 34, 46, 51]. Informally, sequential consistency is the interleaving semantics. It’s a model where every thread has the same view of memory, one memory access from one thread is allowed to execute in program order at a time, and reads always see the last write to the same location.

In reality most modern processor architectures provide a much "weaker" model of execution. That is, they forgo the guarantees of sequential consistency in the interest of improved performance. For example, on most architectures different threads can see writes to different locations happening in different orders. This can make reasoning about algorithm behavior difficult. The problem is further exacerbated by compiler optimizations built with single threaded execution in mind [8].

Recently, Java’s memory model was updated and expanded with "access modes" [26] which can be leveraged by performing shared memory accesses through the new `VarHandle` API [1]. Access modes are intended to clarify Java’s weak memory semantics and provide guarantees to the programmer about how programs will behave that the Java compiler is expected to enforce. The access mode with the fewest guarantees is Java Opaque Mode (hereafter "the JOM"). Using Opaque Mode memory accesses results in faster execution at the cost of more difficulty in reasoning about program behavior when compared with sequential consistency.

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:35
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The few guarantees that are provided by the JOM differ from other memory models, like C/C++11 (hereafter C11), in two key ways. First, the JOM explicitly forbids a particular type of execution where a read can affect its own value through a later write and memory accesses in other threads. By forbidding these "causal cycles", the JOM also forbids an important class of unwanted behavior where reads can produce values that can never be generated by the program, often referred to as thin-air reads [4]. Second, the JOM makes no guarantee that writes to a particular memory address will be seen in a linear order by all threads.

When taken together with the broader weak behavior of the JOM, these differences have important implications for modeling and verification. Where prior models for Java and C11 have tried to work around thin-air reads using complex formal constructs [23, 33], forbidding them outright offers an opportunity to build a relatively simple semantics. On the other hand, the minimal set of guarantees provided by the JOM presents a challenge for reasoning. In particular, the absence of a consistent linear order on writes means that the JOM lacks a global notion of state. Not only does this make reasoning difficult, it also prohibits the use of prior weak-memory logics which depend on global state in their proofs of soundness [47].

Here we present the first formal model for the JOM and a logic to aid in proofs of high-level correctness theorems for lock-free concurrent algorithms. In contrast to prior work, our semantics is relatively simple and our logic is sound in the presence of the JOM's per-location strict partial order on writes. Specifically, we make the following contributions:

- A simple semantics that models Java Opaque Mode, requiring just four rules for the core of the memory model.
- A stateless logic based on equational reasoning for expression values and relational reasoning for memory accesses.
- Mechanized proofs of correctness for Dekker's mutex and a two-thread Ringbuffer found in the Linux kernel.

Our semantics is for an imperative calculus we have mechanized in Coq. It models the guarantees outlined by the JOM specification [26, 27] which we detail later in Section 2. Notably, the core of the memory model is built on a relation defined by four properties that characterizes the pairing of reads and writes. This relation, called coherence, is a cousin to the coherence rules of Batty et al.'s model of C11[3]. We also study one addition to the JOM in the form of specified orders as proposed in [9, 29]. Specified orders are a relational view of synchronization that can be compiled to existing hardware synchronization primitives as illustrated by [6, 43]. This allows our theorems to assume synchronization behavior while our logic maintains a small assertion language. The key idea is that a correct program is not just the code but should also include specified orders. Both are given to the compiler and the result is machine instructions that maintain the guarantees associated with the specified orders.

Our logic must work without any global notion of state since the JOM does not support a total order on non-atomic writes. To accomplish this we use only expression values and the relations of the memory model. In this stateless approach, a key reasoning step proceeds by showing that certain pairings of reads and writes are impossible. The proof proceeds by supposing that a pairing is possible and then deriving a contradiction by building a cycle in the coherence order. In lifting this process of write elimination to expressions and abstracting over the details of the operational semantics our logic provides a significant economy of proof.

To demonstrate the effectiveness of the logic we have mechanized, in Coq, proofs of correctness for a simplified form of Dekker's mutual exclusion algorithm and a two-thread RingBuffer taken from [20, 47]. The theorems for the RingBuffer are similar to the lineariz-

ability specifications of Herlihy and Wing [18] with additional theorems for the bounded size of the queue memory.

The rest of this paper is layed out as follows. In Section 2 we give more detail about the JOM, including example programs that illustrate its four guarantees and its relationship with C11. In Section 3 we illustrate our formalization of the JOM. In Section 4 we present the assertion language, semantics and soundness theorem of our logic. In Section 5 we illustrate how reasoning proceeds in our logic by employing it to prove correctness for Dekker's mutual exclusion algorithm. In Section 6 we examine how induction over the JOM's partial order on writes can be used to prove invariants for memory locations and add structure to proofs by sketching correctness for a RingBuffer algorithm. In Section 7 we place our work in context with prior research. Finally, we include the entirety of our semantics for the JOM in Appendix A and all the rules for our logic in the Appendix B while the full soundness proof for the logic is included in supplementary material.

2 JOM Guarantees

The ninth version of the Java Development Kit provides the `VarHandle` API [1] for shared memory multithreaded programming. Programmers using shared variables for lock-free concurrent algorithms in Java can use this API to get fine-grained semantic guarantees from the compiler. The weakest set of guarantees, otherwise referred to as access modes, is Opaque Mode. Java Opaque Mode provides four core guarantees as set forth in [27]. Here, we will give detail to these guarantees by way of the JOM's relationship with sequential consistency, then more directly by example, finishing with a comparison to C11. First, we summarize the guarantees as follows:

Bitwise Atomicity *Reads will see the value of only one write.* The JOM guarantees that reads will not see mixed bits from different writes at any one time (which could violate type safety guarantees).

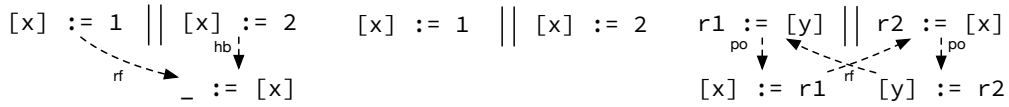
Write Availability *Writes can be read by later reads.* The intent is to avoid a situation where repeated reads (e.g. in a spin-loop) never see a write in another thread because they are optimized by the compiler to execute only one time. When an optimization like this happens the availability of the write for the read, which is intended to break the loop, depends on when the read is executed [8].

Coherence *The order of writes to a particular location should agree with the way that reads see their values.* Broadly, reads can provide information about the ordering of writes on a per location basis. For example, one guarantee (of four we will define later) is that a read should be paired with the last write to the same location that happens before it, where "happens before" might be program order sequencing.

Acyclic Causality *A read should not influence its own value.* Importantly, causal cycles result in counter-intuitive behavior like thin-air reads and reads from unexecuted branches. The JOM forbids all causal cycles excluding these behaviors and, along with them, some optimizations [36].

Recall that sequential consistency (SC) is defined by three, much stronger guarantees. First, memory accesses happen in program order. Second, a read takes the value of the last write to the same location that happens-before it. Finally, all memory accesses are ordered, one way or the other, by happens-before.

The JOM's guarantees are far weaker than those of SC. First, if two accesses in the same thread are to different locations there is no broad guarantee that they will execute in program order. Second, a read still takes the value of the last write before it to the same location but



(a) Coherence: can x = 2?

(b) Total-co: is it the case that $x = 2 \Rightarrow [x] := 1 \xrightarrow{co} [x] := 2$?

(c) Causal cycles: x = y = 42?

Figure 1 Sample Behaviors

the notion of "last" is defined by a per-location ordering on writes called the coherence order ($\xrightarrow{co}$). The definition of the coherence order is dictated by the coherence properties of the JOM. Third, there is no longer a total order on all memory accesses.

2.1 Examples

We will consider each the JOM's guarantees in more detail, with a focus on the Coherence and Acyclic Causality guarantees, by examining the three example programs in Figure 1.

Bitwise Atomicity and Write Availability. Bitwise atomicity connects one read with one write in the reads-from relation ($\xrightarrow{rf}$) to avoid type safety violations. The availability of writes is a guarantee that repeated reads *can* eventually see a write. In particular it is intended to deal with the case where aggressive compiler optimizations can lift a repeated read out of a loop, for example in a spin-lock [8], and thereby break the intended message passing idiom.

Coherence. The ordering of writes in the JOM is determined by four coherence properties. These properties take information about read-write pairings and combine them with other relationships to derive the ordering of writes.

For example, consider one such coherence property: a read must be paired with (i.e. read from, $\xrightarrow{rf}$) the last write (in $\xrightarrow{co}$) that it knows about ($\xrightarrow{hb}$). A program illustrating this property is given in Figure 1a. The question is, if $[x]$ reads 1, can we observe a final state where $x = 2$? The answer is "no". Since $[x] := 2$ happens-before $[x]$ and $[x]$ reads-from $[x] := 1$, by the property described above we can conclude that $[x] := 2$ is coherence-order before $[x] := 1$. But, if x = 2 we would conclude the opposite ordering, a contradiction.

The JOM diverges from other memory models most clearly in its lack of a total order on writes to the same location. Figure 1b gives a simple program to illustrate the difference. The question is, if we observe the state of the shared variable x is 2 at the end of execution does that imply that the write $[x] := 1$ executes before $[x] := 2$? The JOM provides no guarantee of an ordering either way. Intuitively, there is no universal view of state for a given memory location.

Acyclic Causality. A side effect of the weakening of sequential consistency is that the JOM might otherwise admit executions where writes of spurious values can affect their own execution through reads and data flow dependencies, up to and including reading any value at all. The acyclic causality requirement forbids such behaviors.

Consider the example behavior in Figure 1c. The question is, can the reads of x and y see a value not created anywhere in the program? Even though the JOM allows the reordering of accesses to different locations, the answer is "no". The reason is that the JOM forbids the cycle in the transitive closure of program order and reads-from which is illustrated in the example. The performance cost of forbidding such cycles is the subject of recent research in [36].

2.2 C11

The JOM is similar to the "fully relaxed atomics" of C11 but there are important differences between the two models. The relationship can be divided into three parts: coherence

property	JOM	C11
coherence	✓	✓
total coherence order		✓
acyclic causality	✓	

Figure 2 The JOM and C11relaxed

properties, the lack (resp. presence) of a total coherence order, and the lack (resp. presence) of causal cycles. A summary appears in the table in Figure 2.

Coherence. The coherence order of the JOM has the same purpose as the modification order of C11. We prefer “coherence order” over “modification order” since the relation in the JOM is defined entirely by the coherence properties of the model. The behavior of the program in Figure 1a is the same for both C11 and the JOM. Also the four coherence properties of the JOM are same as the "forbidden behaviors" of Batty et al. [3]. However the JOM’s coherence order is not total.

Recall the example in Figure 1b. The question is, if we observe the state of the shared variable x is 2 at the end of execution does that imply that the write $[x] := 1$ executes before $[x] := 2$? In contrast to the JOM, the answer for C11 is "yes" because the alternate order $[x] := 2 \xrightarrow{\text{co}} [x] := 1$ would suggest that the final state for x should be 1 which is a contradiction. Thus, by the total ordering of the two writes we can conclude $[x] := 1 \xrightarrow{\text{co}} [x] := 2$.

Acyclic Causality. Again, consider the example behavior in Figure 1c. For C11 the answer to the question posed there is, "yes". To achieve such a result, $[x]$ and $[y]$ must read from the corresponding writes to x and y so they must happen after those writes. If, as in SC, we assume that program order also means happens-before, then we would have a cycle in happens-before which is a contradiction. C11 and the JOM relax program order. Thus, without a restriction on causal cycles, there is nothing to forbid the cycle in program order and reads-from depicted in the example.

3 Formal Model

We model the JOM using an operational semantics, where the program generates transactions recording the execution of memory accesses and those transactions are validated by the axioms of the memory model before being recorded in a history as memory events. Importantly, our formalization of the JOM assigns the correct behavior to each of the three example programs from Section 2.

We use a core language, detailed in Figure 3, that models key features of Java bytecode. Figure 4 focuses on the most relevant portion of the semantics. The full semantics is available in Appendix A. The top level state step coordinates steps of the program with steps in a sequence of events, a history, which represents memory (rule: step). Critically, when transactions are created by the program they are certified by a step that adds an event in the history semantics.

Most of the expression semantics is standard. We focus here on the rules that most directly affect the memory model. There are two memory access expressions: reads $[s]$, writes $[s] := s$. A memory access goes through three phases: variable substitution, initialization, and execution.

Execution may proceed out of order (rule: let-right) but initialization proceeds in program order (e_1 init) and accesses cannot be initialized before all variable arguments are substituted ($x \notin FV(d)$). The initialization phase reduces any action to a unique identifier and records the form of the action in an event $i = a$ (rule: action-init). The uniqueness of the identifier which replaces the action is guaranteed by the history semantics.

Nat	$n := 0 \mid 1 \mid \ldots$	Action Ids	$i := \ldots$
Value	$v := n$	Thread Ids	$p := \ldots$
Labels	$l := \ldots$	Program	$P := \epsilon \mid P; p : \texttt{fork}\ e$
Simple	$s := v \mid x$	History	$H := \epsilon \mid H, h$
Action	$a := [\,s\,] \mid [\,s\,] := s$	Events	$h := \mathsf{exec}(i) \mid \mathsf{rf}(i,i)$
Exp	$e := s \mid s + s \mid s \bmod n \mid s == s$		$\mid \mathsf{init}(i,p) \mid \mathsf{is}(i,a)$
	$\mid \texttt{let}\ x := e\ \texttt{in}\ e \mid \texttt{repeat}\ e\ \texttt{end}$		$\mid \mathsf{vo}(i,i) \mid \mathsf{push}(i,i)$
	$\mid \texttt{if}\ s\ \texttt{then}\ e\ \texttt{else}\ e \mid s = n\ \texttt{in}\ e \mid l{:}e$	Label List	$L := \epsilon \mid l, L$
		Transaction	$d := \emptyset \mid i = a \mid i \mid i\ \mathsf{to}\ n$

Figure 3 Imperative Syntax

$$\frac{P \xrightarrow{d@p} P' \quad H \xrightarrow{d@p} H'}{(P,H) \to (P',H')}\ \text{step} \qquad \frac{e_2 \xrightarrow{d} e_2' \quad e_1\ \mathsf{init} \quad x \notin FV(d)}{\texttt{let}\ x := e_1\ \texttt{in}\ e_2 \xrightarrow{d} \texttt{let}\ x := e_1\ \texttt{in}\ e_2'}\ \text{let-right}$$

$$\frac{}{a \xrightarrow{i=a} i}\ \text{action-init} \qquad \frac{}{i \xrightarrow{i\ \mathsf{to}\ n} n}\ \text{exec-read} \qquad \frac{}{i \xrightarrow{i} 0}\ \text{exec-write}$$

$$\frac{}{[s/x]e \xrightarrow{\emptyset} s = n\ \texttt{in}\ [n/x]e}\ \text{spec} \qquad \frac{e \xrightarrow{d} e'}{s = n\ \texttt{in}\ e \xrightarrow{d} s = n\ \texttt{in}\ e'}\ \text{spec-under} \qquad \frac{}{n = n\ \texttt{in}\ e \xrightarrow{\emptyset} e}\ \text{spec-rm}$$

$$\frac{\mathsf{wf}(H, \mathsf{exec}(i), p) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H,\mathsf{exec}(i)})}{H \xrightarrow{i@p} H, \mathsf{exec}(i)}\ \text{hist-write} \qquad \frac{\mathsf{wf}(H, \mathsf{rf}(i_w, i), p, n) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{po} \cup \mathsf{rf}}_{H,\mathsf{rf}(i_w,i)}) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H,\mathsf{rf}(i_w,i)})}{H \xrightarrow{i\ \mathsf{to}\ n@p} H, \mathsf{rf}(i_w, i)}\ \text{hist-read}$$

$$\mathsf{wf}(H, \mathsf{exec}(i), p) \triangleq \begin{cases} H(\mathsf{init}(i,p)) \\ H(\mathsf{is}(i, [\,l\,] := \_)) \\ \neg H(\mathsf{exec}(i)) \end{cases} \qquad \mathsf{wf}(H, \mathsf{rf}(i_w, i), p, n) \triangleq \begin{cases} H(\mathsf{init}(i,p)) \\ H(\mathsf{is}(i, [\,l\,])) \\ \neg H(\mathsf{rf}(\_, i)) \\ H(\mathsf{is}(i_w, [\,l\,] := n)) \\ H(\mathsf{exec}(i_w)) \end{cases}$$

Figure 4 Semantics

$$i_1 \xrightarrow{\mathsf{po}}_H i_2 \triangleq \exists H_1\ H_2\ H_3\ p,\ H = H_1, \mathsf{init}(i_1,p), H_2, \mathsf{init}(i_2,p), H_3$$
$$i_1 \xrightarrow{\mathsf{to}}_H i_2 \triangleq \exists H_1\ H_2\ H_3,\ H = H_1, \mathsf{exec}(i_1), H_2, \mathsf{exec}(i_2), H_3$$
$$i_1 \xrightarrow{\mathsf{rf}}_H i_2 \triangleq H(\mathsf{rf}(i_1, i_2))$$
$$i_1 \xrightarrow{\mathsf{push}}_H i_2 \triangleq H(\mathsf{push}(i_1, i_2))$$
$$i_1 \xrightarrow{\mathsf{svo}}_H i_2 \triangleq H(\mathsf{vo}(i_1, i_2))$$
$$i_1 \xrightarrow{\mathsf{vo}}_H i_2 \triangleq i_1 \xrightarrow{\mathsf{rf}}_H i_2 \vee i_1 \xrightarrow{\mathsf{svo}}_H i_2 \vee i_1 \xrightarrow{\mathsf{push}}_H i_2 \vee (\exists i_3\ i_4, i_1 \xrightarrow{\mathsf{push}}_H i_3 \wedge i_4 \xrightarrow{\mathsf{push}}_H i_2 \wedge i_1 \xrightarrow{\mathsf{to}}_H i_4)$$
$$i_1 \xrightarrow{\mathsf{hb}}_H i_2 \triangleq i_1 \xrightarrow{\mathsf{vo}^+}_H i_2 \vee i_1 \xrightarrow{\mathsf{po}}_H i_2$$

Figure 5 Memory Relations

	expression	history
	$\texttt{let}\ v := [\,l\,]\ \texttt{in}\ [\,l\,] := v$	$\varnothing$
$\xrightarrow{i_r=[\,l\,]}$	$\texttt{let}\ v := i_r\ \texttt{in}\ [\,l\,] := v$	$\mathsf{init}(i_r, [\,l\,])$
$\xrightarrow{i_r\ \mathsf{to}\ 1}$	$\texttt{let}\ v := 1\ \texttt{in}\ [\,l\,] := v$	$\mathsf{init}(i_r, [\,l\,]), \mathsf{rf}(i_0, i_r)$
$\xrightarrow{\varnothing}$	$[\,l\,] := 1$	$\mathsf{init}(i_r, [\,l\,]), \mathsf{rf}(i_0, i_r)$
$\xrightarrow{i_w=([\,l\,]\,:=\,1)}$	i_w	$\mathsf{init}(i_r, [\,l\,]), \mathsf{rf}(i_0, i_r),\ \mathsf{init}(i_w, [\,l\,] := 1)$
$\xrightarrow{i_w}$	0	$\mathsf{init}(i_r, [\,l\,]), \mathsf{rf}(i_0, i_r),\ \mathsf{init}(i_w, [\,l\,] := 1), \mathsf{exec}(i_w)$

(a) Initialization

	...	
$\xrightarrow{i_r=[\,l\,]}$	$\texttt{let}\ v := i_r\ \texttt{in}\ [\,l\,] := v$	$\mathsf{init}(i_r, [\,l\,])$
$\xrightarrow{\varnothing}$	$\texttt{let}\ v := i_r\ \texttt{in}\ v = 1\ \texttt{in}\ [\,l\,] := 1$	$\mathsf{init}(i_r, [\,l\,])$
$\xrightarrow{i_w=[\,l\,]\,:=\,1}$	$\texttt{let}\ v := i_r\ \texttt{in}\ v = 1\ \texttt{in}\ i_w$	$\mathsf{init}(i_r, [\,l\,]),\ \mathsf{init}(i_w, [\,l\,] := 1)$
	...	

(b) Speculation

Figure 6 Example Execution

Memory accesses are executed, possibly out of program order, by choosing an arbitrary natural number value (rules: exec-read or exec-write). The event generated by their execution is passed down to the history semantics to ensure that the action associated with the identifier can actually produce the chosen value and event. Note that memory locations are addressed directly by natural numbers and we do not consider allocation.

Finally, the expression semantics incorporates a flexible form of speculation. Speculation works by guessing values for free variables in expressions and adding a constraint that requires later execution to confirm the guessed value (rules: spec and spec-rm). Since execution can proceed under the constraint (rule: spec-under) the semantics is able emulate weak behavior introduced by compiler optimizations.

Figure 6a shows an example execution. First, the read of the concrete location l is initialized to i_r (note that we omit $\mathsf{is}(i, a)$ events for simplicity). Then the read executes and reads from some write i_0 producing the value 1. Once the variable v is substituted into the write it can also initialize and execute.

Alternately, after the first step, before the read executes, the semantics could speculate on the value of v, as in Figure 6b. This allows the write to initialize, and possibly even execute, under the assumption that the read can later satisfy the constraint on the value of v. However the read cannot be paired with the write in this example as it would violate the coherence rules of the JOM. We discuss why this behavior is forbidden by the coherence later in this section.

3.1 History Validation

Histories are sequences of events: identification $\mathsf{is}(i, a)$, initialization $\mathsf{init}(i, p)$, write execution $\mathsf{exec}(i)$, read execution $\mathsf{rf}(i, i)$. Identification records the association between an action and an identifier. Initialization records the thread where the identifier appeared with the initialization's position in a history recording program order. Read and write executions record the execution of identifiers (the value is derived from the $\mathsf{is}(i, a)$ event).

The wf predicate ensures that the events generated by initialization and execution of

actions in the expression semantics conform to the basic semantics of those actions. For example, when an action executes as a read (rule: hist-read), the transaction i to n is checked to ensure that action a associated with i was actually a read in the thread p, that it has not yet been executed, and that it reads from some executed write. The process for a write is similar (rule: hist-write).

In Figure 6a, the program generates events which are validated and recorded in history. If the expression semantics were to generate i_r instead of i_r to 1 when executing the read then the well-formedness conditions of the history semantics would not validate the step.

Aside from these basic well-formedness conditions, the history semantics depends on two acyclicity predicates which form a simple interfaces to the rules and relations of the memory model.

The first, $\mathsf{acyclic}(\xrightarrow{\mathsf{po} \cup \mathsf{rf}}_H)$, is included to forbid causal cycles. By forbidding a cycle in the transitive closure of $\mathsf{po} \cup \mathsf{rf}$ the semantics rules out the unwanted behavior of the example program in Figure 1c. Notably, this is an over-approximation and while it does prevent thin-air reads it also rules out some standard compiler optimizations. We discuss this in more detail in Section 7.

The second, $\mathsf{acyclic}(\xrightarrow{\mathsf{co}}_H)$, is for the coherence order. This predicate along with the four coherence properties of the coherence relation defines the memory model. As we will see, forbidden behavior can be characterized as cycles in the coherence order.

3.2 Relations

The definition of the coherence order, $\xrightarrow{\mathsf{co}}$, is the core of the memory model. It determines which writes may satisfy a read. It is derived from the smaller relations over the memory events in H, including specified orders, in Figure 5.

The program order, po, orders initialization events in the same thread p according to the sequence they were added to the history. The trace order, to, is the order in which the execution of reads and writes appear in the history event list. Specified orders are always assumptions in our proofs, as a technical matter they appear as special events in history: $\mathsf{vo}(i_1, i_2)$ for a specified visibility order, and $\mathsf{push}(i_1, i_2)$ for a push order. Visibility orders, vo, derive from a reads-from relationship, rf, a specified visibility order, svo, a specified push order, push, or a cross-thread, push order pair where the heads are trace ordered. Finally, the hb relation is derived from a transitive vo relationship or a single po relationship.

Intuitively, the specified visibility order, svo, guarantees a globally visible intra-thread ordering of execution between memory accesses while the push order, push, guarantees both an intra-thread ordering of execution and one of two possible cross-thread orderings when combined with another push order.

Concretely, a specified visibility order is a guaranteed vo edge which can be implemented by lightweight fence where available. The push order's effect on the definition of vo is more complex as it models the behavior of a full fence between the head and tail of the order. Since full fences would otherwise be totally trace-ordered, two push orders $i_1 \xrightarrow{\mathsf{push}} i_3$ and $i_4 \xrightarrow{\mathsf{push}} i_2$ imply their intra-thread ordering and either $i_1 \xrightarrow{\mathsf{vo}} i_2$ or $i_4 \xrightarrow{\mathsf{vo}} i_3$. But modeling full fences more directly would require a much more complex definition of visibility.

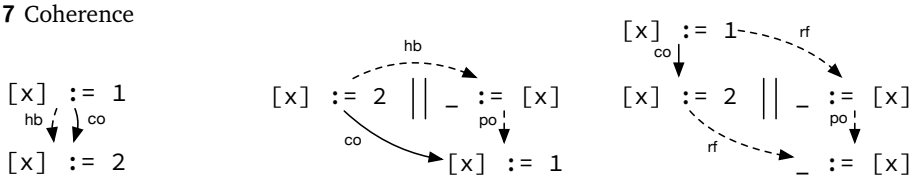
To keep the semantics and the definition of push orders simple we have instead chosen to treat the heads of push orders like SC accesses with additional intra-thread ordering between the head and the tail of each order. Thus, in the definition of vo, if the two heads of the orders, i_1 and i_4, are trace ordered then they are similarly happens-before ordered (as in SC). Coupled with the intra-thread properties of the push order this gives an ordering between the head of the first order, i_1, and the tail of the second, i_2. The effect of this approach and

$$\mathsf{writes}(H, i, l, n) \triangleq H(\mathsf{is}(i, [\,l\,] := n)) \wedge H(\mathsf{exec}(i))$$
$$\mathsf{reads}(H, i, l) \triangleq H(\mathsf{is}(i, [\,l\,])) \wedge H(\mathsf{rf}(\_, i))$$

$$\frac{\begin{array}{ll} & \mathsf{writes}(H, i_1, l, v_1) \\ i_1 \xrightarrow{\mathsf{hb}}_H i_2 & \mathsf{writes}(H, i_2, l, v_2) \end{array}}{i_1 \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-ww} \qquad \frac{\begin{array}{ll} & i_1 \neq i_2 \\ i_1 \xrightarrow{\mathsf{hb}}_H i_r & \mathsf{writes}(H, i_1, l, v) \\ i_2 \xrightarrow{\mathsf{rf}}_H i_r & \mathsf{reads}(H, i_r, l) \end{array}}{i_1 \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-wr}$$

$$\frac{\begin{array}{ll} & \mathsf{writes}(H, i_1, l, v_1) \\ i_1 \xrightarrow{\mathsf{hb}}_H i_r & \mathsf{reads}(H, i_r, l) \\ i_r \xrightarrow{\mathsf{po}}_H i_2 & \mathsf{writes}(H, i_2, l, v_2) \end{array}}{i_1 \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-rw} \qquad \frac{\begin{array}{ll} i_1 \xrightarrow{\mathsf{rf}}_H i_{r_1} & i_1 \neq i_2 \\ i_2 \xrightarrow{\mathsf{rf}}_H i_{r_2} & \mathsf{writes}(H, i_1, l, v_1) \\ i_{r_1} \xrightarrow{\mathsf{po}}_H i_{r_2} & \mathsf{writes}(H, i_2, l, v_2) \end{array}}{i_1 \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-rr}$$

Figure 7 Coherence



(a) co-ww **(b)** co-rw **(c)** co-rr

Figure 8 co-rw and co-rr Examples

modeling full fences more directly is the same: one of two possible orderings of the heads and tails of two push orders dictated by the total trace order.

3.3 Coherence Order

Coherence order edges can be derived using the one of the four rules in Figure 7. Each of these rules corresponds to the forbidden behaviors of Batty et al. and the modification order of the C11 specification coherence rules. We extend the example programs of Figure 1 with additional examples in Figure 8 for the other three coherence rules.

co-ww As in Figure 8a, two writes to the same location, ordered by happens-before, are similarly ordered by co.

co-wr If an action reads from some write ($i_2 \xrightarrow{\mathsf{rf}} i_1$) and if it has seen a second write ($i_1 \xrightarrow{\mathsf{hb}} i_r$) to the same location, then the second write must be ordered before the first write ($i_1 \xrightarrow{\mathsf{co}} i_2$). Intuitively, the read must take the value of the most recent write it has "seen". Recall the example from Figure 1a. If we assume identifiers for the example memory accesses, by co-wr, $[\,x\,] := 2 \xrightarrow{\mathsf{hb}} [\,x\,]$, and $[\,x\,] := 1 \xrightarrow{\mathsf{rf}} [\,x\,]$ we can conclude that $[\,x\,] := 2 \xrightarrow{\mathsf{co}} [\,x\,] := 1$ which contradicts $[\,x\,] := 1 \xrightarrow{\mathsf{co}} [\,x\,] := 2$ (implied by the final state $x = 2$) as a cycle in co.

co-rw If a read to some location has seen some write to the same location ($i_1 \xrightarrow{\mathsf{hb}} i_r$), and there is a second write program order after the read ($i_r \xrightarrow{\mathsf{po}} i_2$), then the writes are coherence ordered. Because we expect writes later in program order to execute after earlier reads to the same location, as in Figure 8b, this prevents the opposite, counter-intuitive execution, ordering $[\,x\,] := 2$ after $[\,x\,] := 1$.

co-rr Coherence order must agree with reads in program order. The co edge in Figure 8c prevents executions where the second read sees the first write because co-rr implies the opposite co edge and thus, a cycle in co.

Notably, the formal specification for C11 defines coherence using forbidden behaviors which works in conjunction with the total order on writes to determine edges in the relation. The JOM has no total order so the coherence relationships must be defined directly. Thus, the behavior in Figure 1b is correctly modeled.

Also, in contrast to the coherence rules of C11, co-rw and co-rr only need to enforce po edges and not hb. This is due to the fact that any vo edge would be pair with the rf to form a transitive vo edge and therefore a direct hb edge. Thus, where co-rw would be employed we could employ co-hb and where co-rr would be employed we could employ co-wr. This is important because, happens-before need not be transitive, which would result in a much stronger memory model.

In spite of the speculation in Figure 6b, it is not possible for the read to be paired with the write. Assume that it does read from the later write, $i_w \xrightarrow{\mathsf{rf}} i_r$. Since the read is po before the write, $i_r \xrightarrow{\mathsf{po}} i_w$, the write is is coherence ordered before itself by co-rw, a contradiction.

3.4 Acyclic Causality

The JOM diverges from C11 in its guarantee of acylic causality. This comes at the cost of disallowing some compiler optimizations [4]. Recall the example of Figure 1c. Our model can produce values that do not appear in the program by first speculating on the value of the read of y as 42 in the first thread and then executing the write to x out of order. Then $[x]$ can read the value 42 allowing the write of y to satisfy the speculation constraint on the read of y. Importantly this execution requires $[y] := x \xrightarrow{\mathsf{rf}} [y]$ and $[x] := y \xrightarrow{\mathsf{rf}} [x]$. Thus, taken together with the program order edges this behavior is contradicts the $\mathsf{acyclic}(\mathsf{po} \cup \mathsf{rf})$ predicate of hist-read.

3.5 Summary

We can now summarize how the formal semantics addresses each of the four guarantees of the JOM:

Bitwise Atomicity *Reads will see the value of only one write.* The pairing of a read with only one write in hist-read guarantees that the value generated by the expression semantics matches exactly one write.

Write Availability *Writes can be read by later reads.* The semantics does not remove memory accesses and can not optimize reads out of loops. Thus, a repeated read can see any write to the same location that has executed, subject to the coherence rules.

Coherence *The order of writes to a particular location should agree with the way that reads see their values.* The four coherence rules and $\mathsf{acyclic}(\xrightarrow{\mathsf{co}}_H)$ define the relationship between writes based on how they are associated with reads.

Acyclic Causality *A read should not influence its own value.* The acyclicity requirement $\mathsf{acyclic}(\xrightarrow{\mathsf{po} \cup \mathsf{rf}}_H)$ prevents these causal cycles as exemplified by Figure 1c.

Importantly, our semantics requires a relatively modest number of concepts and rules to accurately model the JOM. It models a broad range of hardware and compiler optimization behavior through speculation and out-of-order execution. Further, there are only four rules and two predicates to accurately model the coherence and acyclic causality guarantees of the JOM.

4 Logic

Our logic abstracts over the semantics of JOM expressions using inequalities and over memory access behavior using the relations of the memory model. In the case of expressions this

Expressions	$e :=$	$\ldots$
Naturals	$n :=$	$\ldots$
Labels	$l :=$	$\ldots$
Executions	$E :=$	$E;(P,H) \mid (P,H)$
Label Seq.	$L :=$	$L;l \mid l$
Val. Variables	$V :=$	$L \mid n$
Operators	$\circ :=$	$= \mid <$
Relations	$R :=$	$\mathsf{co} \mid \mathsf{coi} \mid \mathsf{hb} \mid \mathsf{po} \mid$ $\mathsf{push} \mid \mathsf{rf} \mid \mathsf{vo}$
Assertions	$A :=$	$\mathsf{false} \mid A \Rightarrow A \mid \forall V, A \mid$ $V \mathbin{\dot\circ} V \mid e \mathbin{@} L \mid L \xrightarrow{R} L$
Assumptions	$\Gamma :=$	$\Gamma; A \mid A$

(a) Language

$$
\begin{aligned}
E \vDash e \mathbin{@} L &\Leftrightarrow \mathsf{match}(E, L, e) \\
E \vDash L \doteq n &\Leftrightarrow E = E'; E'' \wedge E'' \vDash n \mathbin{@} L \\
E \vDash n \mathbin{\dot\circ} n &\Leftrightarrow n \circ n \\
E \vDash L \mathbin{\dot\circ} n &\Leftrightarrow \exists n', L \doteq n' \ \wedge \ n \circ n' \\
E \vDash L_1 \mathbin{\dot\circ} L_2 &\Leftrightarrow \exists n_1, E \vDash L_1 \mathbin{\dot\circ} n_1 \wedge \\
&\qquad \exists n_2, E \vDash L_2 \mathbin{\dot\circ} n_2 \wedge \\
&\qquad n_1 \circ n_2 \\
E \vDash L_1 \xrightarrow{R} L_2 &\Leftrightarrow \exists i_1, \mathsf{actionid}(E, L_1, i_1) \wedge \\
&\qquad \exists i_2, \mathsf{actionid}(E, L_2, i_2) \wedge \\
&\qquad i_1 \xrightarrow[E]{\mathsf{R}} i_2 \\
E \vDash A_1 \Rightarrow A_2 &\Leftrightarrow \text{if } E \vDash A_1 \text{ then } E \vDash A_2 \\
E \vDash \forall V, A &\Leftrightarrow \forall V', \ E \vDash [V'/V]A
\end{aligned}
$$

(b) Semantics

$$\Gamma \vDash A \ \Leftrightarrow \ \forall E, (\forall A' \in \Gamma, E \vDash A') \Rightarrow E \vDash A$$

(c) Models with Assumptions

Figure 9 Logical Assertion Language and Semantics

represents a significant reduction in proof effort. Recall that the expression semantics of our model tracks program order, executes memory accesses out-of-order, and emulates compiler optimizations with speculation, all while maintaining a loose coupling to the history semantics via memory access identifiers. As a consequence, reasoning about expressions directly with the semantics would be tedious, requiring an accounting of the many possible outcomes of the step relation for each expression. Moreover, associating memory access expressions to their identifiers and behaviors in the memory model would be difficult.

Here, we detail the assertion language and semantics of our logic. In Sections 5 and 6 we will state and prove theorems using our assertions and the rules of our logic.

$$\frac{(\texttt{if } s \texttt{ then } l_1{:}e_1 \texttt{ else } l_2{:}e_2) \mathbin{@} L}{\begin{array}{l}(L, l_{cnd} \doteq 1 \wedge L \doteq L, l_1) \vee \\ (L, l_{cnd} \doteq 0 \wedge L \doteq L, l_2)\end{array}} \text{ if-alt}$$

$$\frac{(\texttt{let } x := l_1{:}e_1 \texttt{ in } l_2{:}e_2) \mathbin{@} L}{L \doteq L, l_2} \text{ let-right}$$

$$\frac{\texttt{let } x := l_1{:}e_1 \texttt{ in } l_2{:}e_2 \mathbin{@} L}{\exists n, \ L, l_1 \doteq n} \text{ let-left}$$

$$\frac{x \mathbin{@} L, l_2, L' \quad x \in \mathsf{FV}(e_2)}{\begin{array}{c}\texttt{let } x := l_1{:}e_1 \texttt{ in } l_2{:}e_2 \mathbin{@} L \\ \hline L, l_2, L' \doteq L, l_1\end{array}} \text{ let-bind}$$

(a) Expression Rules

$$\frac{[\,x\,] \mathbin{@} L_r \quad L_r \doteq n \quad L_r, l_{loc} \doteq l}{\exists L_w, \ L_w \xrightarrow{\mathsf{rf}} L_r} \text{ reads-rf}$$

$$\frac{L_1 \xrightarrow{\mathsf{push}} L_2 \quad L_3 \xrightarrow{\mathsf{push}} L_4}{L_1 \xrightarrow{\mathsf{hb}} L_4 \ \vee \ L_3 \xrightarrow{\mathsf{hb}} L_2} \text{ hb-pushes}$$

$$\frac{\begin{array}{c}[\,s\,] := s \mathbin{@} L_1 \\ L_1 \xrightarrow{\mathsf{hb}} L_r \quad L_2 \xrightarrow{\mathsf{rf}} L_r \quad L_1 \neq L_2\end{array}}{L_1 \xrightarrow{\mathsf{co}} L_2} \text{ co-wr}$$

$$\frac{L_1 \xrightarrow{\mathsf{co}} L_2 \quad L_2 \xrightarrow{\mathsf{co}} L_1}{\mathsf{false}} \text{ co-cycl}$$

(b) Memory Rules

Figure 10 Example Rules

$$\frac{\begin{array}{c} L_1 \xrightarrow{\mathsf{co}} L_2 \qquad L_1 \xrightarrow{\mathsf{coi}} L_2 \Rightarrow P\ L_1\ L_2 \\ \forall L_3, L_1 \xrightarrow{\mathsf{co}} L_3 \Rightarrow P\ L_1\ L_3 \Rightarrow L_3 \xrightarrow{\mathsf{coi}} L_2 \Rightarrow P\ L_1\ L_2 \end{array}}{P\ L_1\ L_2}\ \text{co-ind}$$

Figure 11 Coherence Order Induction

4.1 Assertions

Our assertion language, depicted in Figure 9a, is comprised of a standard first order fragment with quantification for labels and natural numbers, and three core program assertions: expression location, memory relations, and expression relations.

To reason about expressions we need a way to identify specific expressions within the program. Further we want to refer to all "instances" of a particular expression, such as library procedures, so that we can reason about all occurrences of the expression. To that end the logic assumes a labeling discipline to locate expressions and allows for quantification over labels. Intuitively, quantification over labels is quantification over program context.

The memory relations and memory model rules of the semantics are lifted from identifiers to expressions via label sequences that point to memory accesses. The memory model rules of the logic work by establishing cycles in $\xrightarrow{\mathsf{co}}$ and thereby help to rule out certain write executions as contradictions. As a result the logic enables reasoning about the values of read expressions. Recall the co-rw rule of the semantics. In Figure 10 we have lifted the identifiers of co-rw to labels matching the appropriate expressions.

In Figure 11, the rule co-ind encodes structural induction with predicates in our assertion language over writes at the label sequences L_1 and L_2. Note that co the relation is transitive. So, to perform induction, we introduce the coi relation to signify that there are no intervening writes (i.e. $\mathsf{coi}^+ = \mathsf{co}$). This rule allows proofs to incorporate existing invariant based reasoning about single memory locations in spite of the partial ordering of writes in the JOM.

Finally, expression relations relate the eventual value of an expression during execution with the eventual value of its subexpressions. For example the if-alt rule in Figure 10. We know that when an if expression evaluates to some n then one of its sub expressions must have evaluated to n.

4.2 Semantics

The semantics of our assertion language, $E \vDash A$, is defined inductively, see Figure 9b, where an execution, E, is a finite sequence of program history pairs related by the semantics of the top level transition relation, $(P, H) \rightarrow^* (P', H')$. The match function recursively locates expressions within the initial program, P, of an execution, E, using the label sequence L (See Appendix C for the definition) . We use this to follow the evolution of the expression, located at L, across an execution to a value with $E \vDash L \circ n$. With that we can reason about binary relations over labels and natural numbers, $E \vDash L \circ n$ and $E \vDash L_1 \circ L_2$. In our proofs of Dekker and RingBuffer we use $=$ and $<$. The memory relations, $L_1 \xrightarrow{\mathsf{R}} L_2$, are also defined in terms of matching expressions. Recall that each memory operation is assigned an identifier i and the relations are defined over these identifiers. The actionid predicate ensures that the labeled expressions reach the correct identifier and that they reach the correct value (See Appendix C for the definition). Other standard connectives are defined in terms of $\Rightarrow$, false, and $\forall V, A$. Finally, we lift our assertions from executions to collections of assertions as in Figure 9c.

The careful reader will have noticed that the equational rule for if-alt has a single matching hypothesis of the form $e \ @ \ L$. Further, the semantics of $e \ @ \ L$ only match in the original

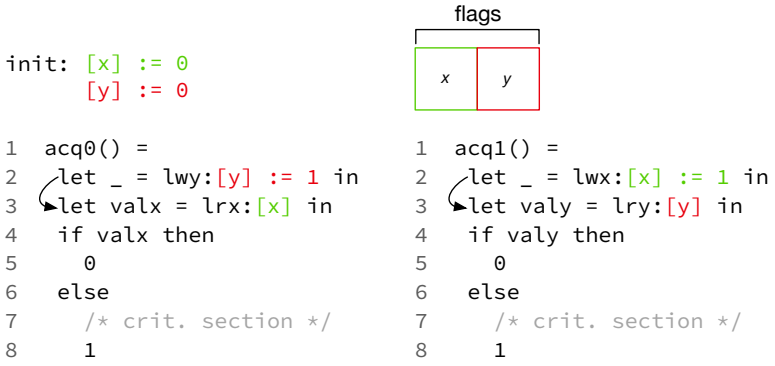


Figure 12 Dekker's Mutex

program. In practice all of the expression level rules require an additional assumption of equality, i.e. $\exists n, L \doteq n$. Since that assumption is identical for every rule we have simply added it to the matching requirement and retained the simpler assertion semantics here for clarity.

4.3 Soundness

Where $\Gamma \vdash A$ is defined by the rules of our logic, we can now state our soundness theorem.

▶ **Theorem 1** (Soundness). *if* $\Gamma \vdash A$ *then* $\Gamma \vDash A$

The proof proceeds by induction on $\Gamma \vdash A$. Thus, we must establish $\Gamma \vDash A$ from the semantic definitions of the assumptions in each rule of the logic.

A full listing the rules of our logic appears in Appendix B. Proof sketches for the soundness of a few expression and memory ordering rules are included in in Appendix D. Formal proofs of the expression and memory rules appear in the supplementary material.

5 Proof by Write Elimination: Dekker

In the absence of a tradition notion of state, our logic facilitates reasoning about the JOM through a process of *write elimination*. Write elimination removes unwanted writes that a read can be paired with in $\xrightarrow{\mathsf{rf}}$. By eliminating such writes we can prove important properties of the value returned by the read without a notion of state.

Our approach to write elimination is based on proof by contradiction. Proofs in this style have three main ingredients: assumptions, derived relations, and derived contradictions. We begin the process of eliminating a write by collecting some basic assumptions including the the $\xrightarrow{\mathsf{rf}}$ relationship we wish to eliminate. We then add *derived relationships* implied by the memory model and the assumptions. With these relationships we show a contradiction in the form of a cycle in the coherence order.

5.1 Dekker

As an example of write elimination we consider Dekker's mutual exclusion algorithm in Figure 12. The memory operations use bracket syntax, for example `[x]` and `[y] := 1`, and some of the expressions are labeled, `lrx:[x]`. Other syntatic elements are standard. The procedures of the algorithm, executing concurrently, work by communicating their intent to enter the critical section through the flags `x` and `y`. We wish to show that it is impossible for both procedures to enter the critical section, see Theorem 2.

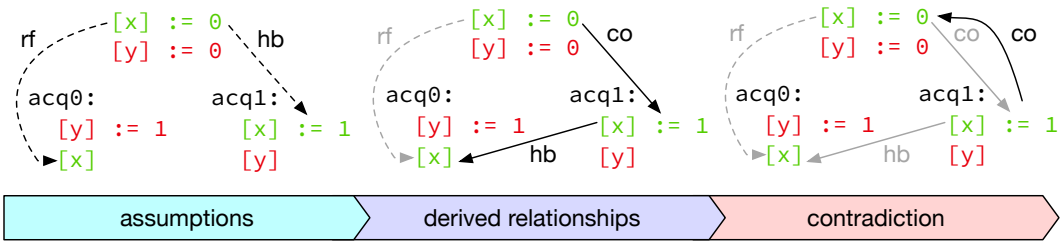


Figure 13 First Write Elimination, Dekker, SC

To begin, we assume the first thread's read, `[x]`, reads from the initialization and thereby enters the critical section. We wish to eliminate the case where the second thread's read, `[y]`, also reads from the initialization. That is, we want to eliminate the case where both thread's reads see the value 0 for `x` and `y` and thereby eliminate the possibility that both threads enter the critical section. If we can show that `[y]` has "seen" the write, `[y] := 1` in `acq0` then it could not have read from the initialization because, recalling co-wr, a read should always be connected with the latest write it is aware of.

We will first focus on write elimination by examining the proof under sequential consistency and then carry that intuition forward to work with the proof under the JOM.

5.2 Write Elimination with Sequential Consistency and the JOM

Sequential consistency allows us to derive many relationships between memory accesses. Standard proofs of Dekker (or Dekker like mutex algorithms [17]) rely on small subset of these relationships. In particular they use the happens-before relation inferred from the program order between the read and write of each thread and a total happens-before relation across the threads between `[x]` in `acq0` and `[x] := 1` in `acq1`.

With this small set of relationships in hand we can complete the proof with two write eliminations. We will focus in on the first to detail the three step approach to write elimination as set out in Figure 13.

First, as stated, we assume that `[x]` read from the initialization. We also assume that initialization always happens before any other memory access to the same location. These are the dashed lines in Figure 13, labeled *assumptions*.

Second, from sequential consistency, we have a total happens-before relation across the threads between `[x]` in `acq0` and `[x] := 1` in `acq1`. One side of the disjunction is shown as the solid line in Figure 13, labeled very generally as *derived relations*. Further, by co-ww, if two writes to the same location are happens-before ordered then they are similarly coherence ordered, so we know that the initialization, `[x] := 1`, must be coherence ordered before the write in `acq1`.

Finally, we construct a contradiction in the form of a cycle. Recall that, by co-wr, a read should be paired with the latest write it knows about (rule: co-wr). Thus, if `[x] := 1` happens before `[x]` then `[x]` is aware of `[x] := 1` and the write must happen before the initialization. This provides the dark solid edge in Figure 13 labeled below with *contradiction*. Intuitively, a write cannot happen before itself. This contradictory cycle eliminates the initialization write from being paired with `[x]` which contradicts this side of the total order between `[x]` and `[x] := 1`.

The same process applies to the second side of the total order derived from sequential consistency (and in all other write eliminations). We assume that `[y]` reads from the initialization, See diagram 1 of Figure 14. From the right side of the total order we have that `[x]` happens before `[x] := 1`. From program order we have that `[y] := 1` happens before `[x]` and `[x] := 1` happens before `[y]`. Then by transitivity we know that `[y]` must have

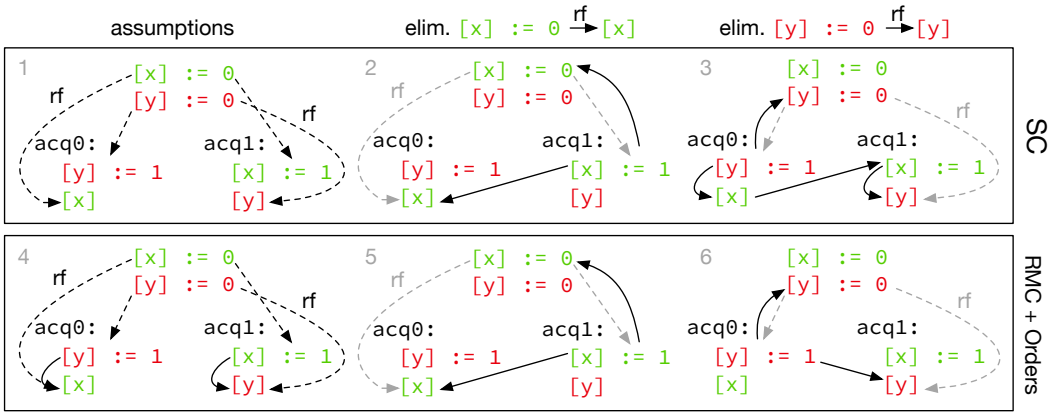


Figure 14 All Write Eliminations, Dekker, SC and RMC

seen `[y] := 1` and we get the contradictory cycle illustrated in diagram 3 of Figure 14. As a result we know that `[y]` could not have read from the initialization, as required.

Once we have that `[y]` read the value 1 we can conclude that the `if` statement must follow the `then` branch. As a result, the procedure would evaluate to 0 and avoid the critical section. Thus, combining the relations provided by sequential consistency and the stated memory model axiom allows us to eliminate the unwanted write from the initialization and demonstrate mutual exclusion.

The JOM is far weaker than sequential consistency and it does not provide a way to derive the happens-before relationships we needed in the previous proof. For the purposes of Dekker, the only thing that the JOM retains is the axiom that a read should be paired with the latest write it knows about.

Thus, to perform the two write eliminations we must recover the relationships derived from SC in the write elimination in diagram's 2 and 3 of Figure 14. In practice, we do this by specifying a *push order* between the read and write in both `acq0` and `acq1`.

This manifests in the algorithm as the orders of Figure 12. which appear as the assumptions of diagram 4 of Figure 14. A pair of push orders like this provides a cross thread guarantee that either the head of the first push order happens before the tail of the second push order or vice versa (see hb-pushes in Figue 10). The result is the derived inter-thread orders in Figure 14, diagrams 5 and 6.

We can now consider both cases of the disjunction implied by the push orders. In each case we will derive the same contradiction from the proof under sequential consistency. We assume the `[x]` read from the initialization and the additional push orders. From the push orders we derive that `[x] := 1` happens before `[x]`. Then `[x]` must have seen `[x] := 1` and, as with sequential consistency, it must be before the initialization which is a contradiction, see diagram 5 in Figure 14. Similarly if `[y] := 1` happens before `[y]` then `[y]` must have seen `[y] := 1`. Assuming that it read from the initialization, we derive a cycle and a contradiction, see diagram 6 in Figure 14.

5.3 Expression Equalities

With the basic outline of the proof in place we state our correctness specification for mutual exclusion using expression equalities. Intuitively, since the value of the expression signals whether the method entered the critical section, we can show that if one enters the critical section the other should not.

▶ **Theorem 2** (Mutual Exclusion). *If* `acq0` $\doteq 1$ *then* `acq1` $\doteq 0$.

$1 \doteq \texttt{acq0}$	by assumption and symmetry	
$\doteq \texttt{acq0}, l_1$	by let-right, $\texttt{let y} := 1 \texttt{ in } l_1 : \ldots \texttt{ @ acq0}$	
$\doteq \texttt{acq0}, l_1, l_2$	by let-right, $\texttt{let valx} := \texttt{lrx} : [\texttt{x}] \texttt{ in } l_2 : \ldots \texttt{ @ acq0}, l_1$	(1)
$1 \doteq \texttt{acq0}, l_1, l_2$	by (1)	
$\doteq \texttt{acq0}, l_1, l_2, l_3$	by the right side of the left disjunct of if-alt	
$\doteq 0$	contradiction	(2)
$0 \doteq \texttt{acq0}, l_1, l_2, l_{cnd}$	by the left side of the right disjunct of if-alt	
$\doteq \texttt{acq0}, l_1, \texttt{lrx}$	by let-bind, $\texttt{let valx} := \texttt{lrx} : [\texttt{x}] \texttt{ in } l_2 : \ldots \texttt{ @ acq0}, l_1$	(3)

Figure 15 Equational Reasoning

We begin our proof sketch by recalling that, in Section 5, we assumed the read `[x]` in `acq0` read from the initialization. Here we will derive that fact from the assumption $\texttt{acq0} \doteq 1$.

▶ **Lemma 3** (Entered, Not Flagged). *If* $\texttt{acq0} \doteq 1$ *then* $\texttt{acq0}, \texttt{lrx} \doteq 0$.

The derivation of Lemma 3 appears in Figure 15. Note, that some of the labels do not appear in Figure 12. In particular we have been referencing the memory accesses informally with sequences like `acq0,lrx`, but in the derivation we use the more precise $\texttt{acq0}, l_1, \texttt{lrx}$. Referenced rules appear in Figure 10 in Section 4.

In Figure 15 equation (1), the two let binding expressions and the assumed equality ($\texttt{acq0} \doteq 1$) mean that the `if` expression must evaluate to 1. Then by if-alt, it is either the case that the `then` branch is equal to 1 and the condition expression (`valx`) is equal to 1 or the `else` branch is equal to 1 and the condition is equal to 0. The `then` branch results in a contradiction (2), so we have that $\texttt{valx} \doteq 0$. Then it must be that the bound expression `lrx` is also equal to 0 by let-bind, in (3) as required. With this equality we can show that the read of `x` in `acq0` read from the initialization by using the value and write elimination to rule out the write `[x] := 1` in `acq1`.

With the read `[x]` connected to the initialization, we can use use the match `[y] @ acq1,lry` and the memory model level reasoning from Section 5 to derive the corresponding lemma for `acq1`.

▶ **Lemma 4** (Flagged, Failed to Enter). *If* $\texttt{acq1}, \texttt{lry} \doteq 1$ *then* $\texttt{acq1} \doteq 0$.

Using similar reasoning to (3) we can show that the `if` condition must be 1 and that the `if` must be 0. By similar reasoning to (1) and (2) we can show that $\texttt{acq1} \doteq 0$, as required.

5.4 Reads to Memory

We will now formalize the reasoning of Section 5.2 and connect it to the expression level derivations of Section 5.3.

If we know that a read, $[\texttt{x}] \texttt{ @ } L$ results in some value ($L_r \doteq n$) and read from a particular location ($L_r, l_{loc} \doteq l$), we also know that it read from some write ($\exists L_w, L_w \xrightarrow{\text{rf}} L_r$). The write is unknown but we consult a disjunction over a finite set of writes which could have satisfied the read for the location equal to L, l_{loc}. In the case of Dekker, we have two locations, x and y.

$$[x] := s \texttt{ @ } L_w \Rightarrow L_w = \texttt{init} \vee L_w = \texttt{acq1}, \texttt{lwx} \tag{4}$$

$$[y] := s \texttt{ @ } L_w \Rightarrow L_w = \texttt{init} \vee L_w = \texttt{acq0}, \texttt{lwy} \tag{5}$$

For our Dekker example, the writes in equations (4) and (5) are fixed because we are considering the whole program, so they do not require quantification over the label sequences

$\texttt{init} \xrightarrow{\mathsf{rf}} \texttt{acq1}, \texttt{lry}$ (6)

$\texttt{init} \xrightarrow{\mathsf{co}} \texttt{acq1}, \texttt{lwy}$ (7)

$\texttt{acq0}, \texttt{lwy} \xrightarrow{\mathsf{push}} \texttt{acq0}, \texttt{lrx}$ (8)

$\texttt{acq1}, \texttt{lwx} \xrightarrow{\mathsf{push}} \texttt{acq1}, \texttt{lry}$ (9)

(a) Assumptions

$\texttt{acq1}, \texttt{lwx} \xrightarrow{\mathsf{hb}} \texttt{acq0}, \texttt{lrx} \vee \texttt{acq0}, \texttt{lwy} \xrightarrow{\mathsf{hb}} \texttt{acq1}, \texttt{lry}$	by hb-pushes, (8), and (9)	(10)
$\texttt{acq0}, \texttt{lwy} \xrightarrow{\mathsf{hb}} \texttt{acq1}, \texttt{lry}$	by the right side of (10)	(11)
$\texttt{acq0}, \texttt{lwy} \xrightarrow{\mathsf{co}} \texttt{init}$	by co-read, (6), and (11)	(12)
false	by co-cycle, (7), and (12)	

(b) Proof by Contradiction

Figure 16 Write Elimination Proof

to locate the writes. More often we will quantify over label sequences in the write-set definition (e.g. RingBuffer in Section 6).

We then perform write elimination by cases as illustrated previously to show that only the desired writes will satisfy the read. Once we have information about which writes are possible we can match the write value with the read value and continue our reasoning.

For each write we wish to eliminate, we use specified orders (hb-pushes) and axioms of the memory model (co-read) to prove a contradiction (co-cycle). The rule hb-pushes gives us a disjunction over visibility of the heads and tails of any two push orders. Recall that in Dekker we have push orders from the write to the read in each thread.

The rule co-read draws on the intuition we described earlier: Of all the writes that a read (L_r) is aware of ($L_1 \xrightarrow{\mathsf{hb}} L_r$), the write that it reads from ($L_2 \xrightarrow{\mathsf{rf}} L_r$) should happen last ($L_1 \xrightarrow{\mathsf{co}} L_2$).

The rule co-cycl says that a write can't happen before itself. This is the contradiction we use to eliminate a write.

Recall diagrams 5 and 6 from Figure 14. Our goal is to show that a read of the initialization by, `[y]` in `acq1` results in a contradiction. Thus it could not have read 0. We will focus on the elimination in diagram 6.

▶ **Lemma 5** (Flag Read). *if* $\texttt{init} \xrightarrow{\mathsf{rf}} \texttt{acq0}, \texttt{lrx}$ *then* $\texttt{acq0}, \texttt{lwx} \xrightarrow{\mathsf{rf}} \texttt{acq1}, \texttt{lry}$

By matching we have $[\mathrm{y}] \mathbin{@} \texttt{acq1}, \texttt{lry}$ and by let-left we have that $\exists n, \texttt{acq1}, \texttt{lry} \doteq n$. Then by reads-rf and the write set, we have a disjunction of two writes. We wish to eliminate the initialization. Thus we assume $\texttt{init} \xrightarrow{\mathsf{rf}} \texttt{acq1}, \texttt{lry}$ and derive a contradiction.

The assumptions we require from Figure 14 correspond with the equations in Figure 16a. With them we can derive a cycle using the steps detailed in Figure 16b.

Note that the two sides of the disjunction in (10) are diagrams 5 and 6 in Figure 14 and the derivation is for the right side which corresponds with diagram 6. Thus, having considered both sides of the push disjunction we concluded that `acq1,lry` must have read from `acq0,lwy` by eliminating the initialization, as required.

5.5 Theorem 2

We can now complete the proof of Theorem 2. Assume $\texttt{acq0} \doteq 1$, then by Lemma 3 and the fact that only initialization writes the value 0 to `x`, we have that $\texttt{init} \xrightarrow{\mathsf{rf}} \texttt{acq0}, \texttt{lry}$.

Then, by Lemma 5 we have that `acq1, lrx` must have read from `acq0, lwy`. Since the value of written by `acq0, lwy` is 1, `acq1, lrx` must evaluate to 1, that is `acq1, lrx` $\doteq 1$. Finally, since we have `acq1, lrx` $\doteq 1$, we employ Lemma 4 to show `acq1` $\doteq 0$. Thus, we have proven mutual exclusion for Dekker without using state. Our mechanized proof of Dekker is 260 lines in Coq without whitespace.

6 Induction on the Coherence Order: RingBuffer

In our logic, induction over the coherence order allows proofs to incorporate existing invariant based reasoning about single memory locations using the partial ordering of writes in the JOM. It also allows proofs to build on a structure for the surrounding program without explicitly defining it. When used in conjunction with quantification over label sequences, our logic can prove important properties of library algorithms like RingBuffer.

6.1 RingBuffer Algorithm and Specification

Our implementation, detailed in Figure 17, is a simplified form of the two-thread ring buffer that appears in the Linux kernel documentation [20]. It closely follows that of [47], though we do not consider allocation and we use monotonic read and write indices [1]. It implements a queue using a fixed number of memory locations, which means that when the buffer is full `tryProd` will fail and when it is empty `tryCons` will fail.

We treat `wi`, `ri`, `b`, and `N` as fixed constants representing the write index offset (0), read index offset (1), buffer offset (2), and buffer size respectively. We also include the `succeed` and `fail` result constants for clarity.

Each procedure returns a value when the corresponding expression evaluates to a natural number. When the `tryProd` procedure successfully enqueues an element `x` in `q` it evaluates to `succeed`, otherwise it evaluates to `fail`. When the `tryCons` procedure successfully dequeues from `q`, it evaluates to the dequeued value, otherwise it evaluates to `fail`.

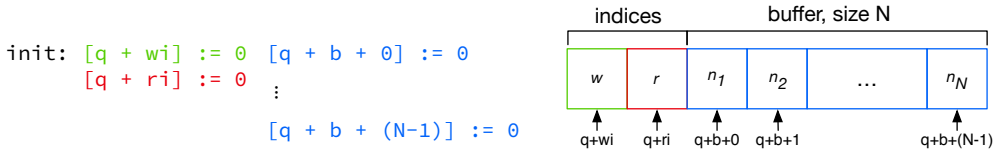
The writer index is managed by `tryProd` write `[wic] := w'` (green code/cell). It represents the tail of the queue. The reader index is managed by the `tryCons` write `[ric] := r + 1` (red code/cell). It represents the head of the queue. Both the reader and writer indices are allowed to increase indefinitely. When the buffer (blue cells) is empty the head and tail are equal (modulo `N`). When the buffer is full the writer index is one fewer than the reader index (both modulo `N`). The buffer index is the position at which a `tryProd` or `tryCons` enqueues or respectively dequeues an `x`. The buffer index is always calculated modulo the length of the buffer (`N`).

The correctness of the RingBuffer algorithm hinges on how the state of the writer index and reader index evolve over time. Specifically, we must show how the state of each evolves individually and then we must use that information to show how they evolve together.

As examples, the individual indices must progress by one index at a time. Otherwise, a `tryCons` invocation may skip an element in the queue or a `tryProd` invocation may leave an element that has already been dequeued for a `tryCons` to dequeue again. Together, the indices must stay within the bounds of the buffer size with respect to one another or the algorithm will exhibit similar problems.

We define the three theorems in our specification for RingBuffer. In our theorems we use the following conventions. We use $L_{\texttt{tryProd}}$ and $L_{\texttt{tryCons}}$ to represent arbitrary label sequences

[1] This is worthy of special note since the logic of [47] employs ghost state with overflowing indices which is a non-trivial addition to the logic.




```
1  tryCons(q) =
2   let wic = q + wi in
3   let ric = q + ri in
4   let w = [wic] in
5   let r = [ric] in
6   let limited = w mod N == r mod N in
7   if limited then
8     fail
9   else
10    let index = q + b + (r mod N) in
11    let x = [index] in
12    let _ = [ric] := r + 1 in
13    x
14
```

```
1  tryProd(q, x) =
2    let wic = q + wi in
3    let ric = q + ri in
4    let w = [wic] in
5    let r = [ric] in
6    let w' = w + 1 in
7    let limited = w' mod N == r mod N in
8    if limited then
9      fail
10   else
11     let index = q + b + (w mod N) in
12     let _ = [index] := x in
13     let _ = [wic] := w' in
14     succeed
```

Figure 17 RingBuffer

locating instances of their respective procedures. We use `ri`$*$, `wi`$*$, and `buff`$*$ to represent label sequences locating the read index, write index, and buffer location accesses within the procedures where $* \in \{\texttt{rd}, \texttt{wr}\}$ for reads and writes. Finally we use $L_1, \texttt{seq}$ to represent sequence concatenation.

▶ **Lemma 6** (Paired). *If we have* $L_{\texttt{tryCons}} \dot{\neq} \texttt{fail}$ *and* $L_{\texttt{tryProd}} \doteq \texttt{succeed}$ *then* $L_{\texttt{tryProd}}, \texttt{buffwr} \xrightarrow{\text{rf}} L_{\texttt{tryCons}}, \texttt{buffrd}$ *iff* $L_{\texttt{tryProd}}, \texttt{wird} \doteq L_{\texttt{tryCons}}, \texttt{rird}$

Informally, if a `tryCons` expression and `tryProd` expression both succeed, then the `tryCons` reads from the write to the buffer in the `tryProd` if and only if the reader and writer indices (resp.) used to calculate the buffer index are the same.

Lemma 6 is a natural correctness criteria for an unbounded linear queue. It guarantees that the n^{th} dequeue is paired with the n^{th} enqueue, where n^{th} is defined here by the monotonic reader and writer indices. As evidence, it's possible to use Lemma 6 to prove the key theorems for the concurrent queue of Herlihy and Wing [18] (see Appendix E).

Theorems 7 and 8 focus on the bounded nature of the queue. Intuitively, each dequeue (`tryCons`) should be paired with a newly enqueued value (`tryProd`) and not an old one, preventing stale values, and a dequeue (`tryCons`) musth have executed when more than one enqueue (`tryProd`) is attempted at a particular position in the buffer, preventing overwrites.

▶ **Theorem 7** (Produce). *If we have* $L_{\texttt{tryCons1}} \neq L_{\texttt{tryCons2}}$, $L_{\texttt{tryCons1}} \dot{\neq} \texttt{fail}$, $L_{\texttt{tryCons2}} \dot{\neq} \texttt{fail}$, $L_{\texttt{tryProd1}}, \texttt{buffwr} \xrightarrow{\text{rf}} L_{\texttt{tryCons1}}, \texttt{buffrd}$ *and* $L_{\texttt{tryProd2}}, \texttt{buffwr} \xrightarrow{\text{rf}} L_{\texttt{tryCons2}}, \texttt{buffrd}$ *then* $L_{\texttt{tryProd1}} \neq L_{\texttt{tryProd2}}$

Informally, If two distinct `tryCons` invocations succeed, then their corresponding `tryProd` invocations are distinct. That is, no two `tryCons` invocations can read from the same `tryProd` and thus no `tryCons` can read a stale value.

▶ **Theorem 8** (Consume). *If we have* $L_{\texttt{tryProd1}} \neq L_{\texttt{tryProd2}}$, $L_{\texttt{tryProd1}} \doteq \texttt{succeed}$, $L_{\texttt{tryProd2}} \doteq \texttt{succeed}$ *and* $L_{\texttt{tryProd1}}, \texttt{buffwr}, l_{loc} \doteq L_{\texttt{tryProd2}}, \texttt{buffwr}, l_{loc}$ *then there exists a* $L_{\texttt{tryCons}}$ *such that, if* $L_{\texttt{tryCons}} \dot{\neq} \texttt{fail}$ *then* $L_{\texttt{tryProd1}}, \texttt{buffwr} \xrightarrow{\text{rf}} L_{\texttt{tryCons}}, \texttt{buffrd}$

Informally, If two distinct `tryProd` invocations succeed and write to the same place in the buffer, then there must be a `tryCons` invocation that reads from the write of the first

tryProd invocation. That is, values in the buffer can't be overwritten without having been read. Note that we must assume the successful execution of the tryCons ($L_{\texttt{tryCons1}} \neq \texttt{fail}$) in our conclusion. This is a weakness of our abstraction over the expression semantics in dealing with the situation where the tryCons has not fully evaluated but the increment of the reader index inside the procedure executed. In the language of Herlihy and Wing, the tryCons is concurrent with the second tryProd but the update to the index has taken place allowing the second tryProd to proceed and reuse the buffer index.

Our mechanized the proofs of these theorems is approximately 3000 lines in Coq without whitespace. Notably, there is a large amount of duplicate proof code shared between tryCons and tryProd invariants. Here, we give a proof sketch that focuses on how our logic supports complex algorithms using induction on the partial coherence order. We will work through a series of lemmas building up to the proof of Theorems 7 and 8. We first give a simple example to show how the coherence order supports proofs of invariants and then examine a more complex example where it forms a scaffolding for write elimination.

6.2 Individual Invariants

In the process of proving Theorems 7 and 8 we will need to prove a series of lemmas. We begin with two invariants, one for the reader index and one for the writer index.

▶ **Lemma 9** (Monotonic Writer). *If we have* $\mathsf{writes}(L, q + wi, n_1)$, $L_{\texttt{tryProd}}, \texttt{wiwr}, l_{val} \doteq n_2$ *and* $L \xrightarrow{\mathsf{co}} L_{\texttt{tryProd}}, \texttt{wiwr}$ *then* $n_1 < n_2$

Informally, if one write to the writer index is earlier than another then the first write's value is smaller than that of the second write. Here, $\mathsf{writes}(L, l, n)$ is used to encapsulate the equality of the location of the write expression, the equality of the value of the write expression, and the execution of the write expression itself. We use an abstract L because the first write may be the initialization for the writer index location, $q + wi$. The lemma for the reader index is similar.

The proofs for these lemmas proceed by induction on the coherence order of the writes to their respective indices. We will focus on tryProd and Lemma 9. The proof for tryCons is similar.

We will rely on the fact that every tryProd write to the writer index is an increment of the coi previous write. Intuitively, each successful addition to the buffer should increment the previous (by coi) writer index by 1.

We can perform induction on $L \xrightarrow{\mathsf{co}} L_{\texttt{tryProd}}, \texttt{wiwr}$ (first assumption of the rule co-ind in Figure 11). Where $q + wi$ is the write index memory location, we wish to show:

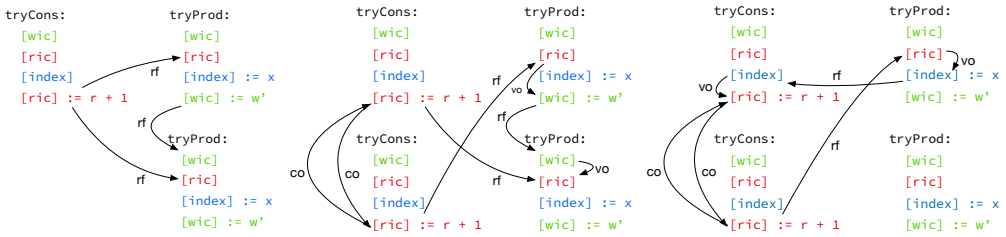
$$P\ L\ L_{\texttt{tryProd}}, \texttt{wiwr} \triangleq \mathsf{writes}(L, q + wi, n_1) \Rightarrow L_{\texttt{tryProd}}, \texttt{wiwr}, l_{val} \doteq n_2 \Rightarrow n_1 < n_2$$

By co-ind, in the base case, we must show that $L \xrightarrow{\mathsf{coi}} L_{\texttt{tryProd}}, \texttt{wiwr}$ (second assumption of co-ind). By the assumed lemma above we know the read in $L_{\texttt{tryProd}}$ will be the value written by L. Thus the tryProd will add one to the read value for its write, which is greater than the value written by L.

In the inductive case, we have some $L'_{\texttt{tryProd}}$ and we have that $L \xrightarrow{\mathsf{co}} L_{\texttt{tryProd}}, \texttt{wiwr}$ implies that the value written by $L'_{\texttt{tryProd}}, \texttt{wiwr}$ is greater than the value written by L (third assumption of co-ind). As before, each tryProd reads from the the coi previous tryProd write and increments it. Thus the write in $L_{\texttt{tryProd}}$ is greater.

6.3 Collective Invariants

Next we will establish bounds on each index relative to its counterpart in the opposite procedure. The writer index must not "wrap around" and pass the reader index otherwise



(a) Same `tryCons` **(b)** Backward read **(c)** Read from future write

Figure 18 Two-thread Invariant Cycles

previously enqueued items will be lost. Similarly the reader index must not pass the writer index otherwise already dequeued items will be dequeued again.

We write these bounds as invariants, here for `tryProd` in Lemmas 10. Notably, these are the same core invariants proved for RingBuffer by Turon et al in [47].

▶ **Lemma 10** (Writer Invariant). *If we have* $L_{\texttt{tryProd}} \doteq \texttt{success}$, $L_{\texttt{tryProd}}, \texttt{wiwr}, l_{val} \doteq w$ *and* $L_{tryProd}, \texttt{rird} \doteq r$ *then* $w < r + N$

Informally, if a `tryProd` succeeds and writes to the writer index, then the value was smaller than the reader index it saw plus the size of the buffer, N. The lemma for the reader invariant is similar but shows that $r \leq w$.

We will again focus on `tryProd` and Lemma 10. The proof for `tryCons` is similar. We will show $w < r + N$. The proof proceeds by induction on $\texttt{init} \xrightarrow{\text{co}} L_{\texttt{tryProd}}, \texttt{wiwr}$.

In the base case, we know that $\texttt{init} \xrightarrow{\text{coi}} L_{\texttt{tryProd}}, \texttt{wiwr}$ and we know that the `tryProd` at $L_{\texttt{tryProd}}$ will read from the initialization so the invariant will be satisfied no matter which write the value for r came from.

In the inductive case, we know that the writer index value the `tryProd` reads has the desired property by coi. We also know that it must have come from the most recent previous `tryProd` and is consequently one fewer than w. That is, we can show $w - 1 < r' + N$, where r' represents the reader index value seen by the previous `tryProd`. From this we derive, $w \leq r' + N$. Then since we wish to show $w < r + N$, it's enough to show that the reader index value of the later `tryProd` must be larger than the value seen by the coi previous `tryProd`. That is, we will show $r' < r$. With the scaffolding from induction in place we can perform write elimination using the ordered `tryProd`s.

If both `tryProd` invocations read from the same write, see Figure 18a, then $r' = r$, and we consider cases for, $w \leq r + N$, with r substituted for r'. If $w < r + N$, then we are done. Otherwise $w = r + N$ and $w \bmod N = (r + N) \bmod N$ which means the buffer is full and we can show that the second `tryProd` would have taken the first branch of the `if` on line 8 and could not have written to the writer index, a contradiction.

That leaves distinct writes. We consider cases of the total coherence ordering of the two `tryCons` writes to the reader index that were read by the `tryProd`s. If the write of r' is coherence order earlier than the write of r we apply reader index monotonicity to show $r' < r$ and we are done. If the second `tryProd` invocation read from an earlier `tryCons`, depicted in Figure 18b as a red dashed rf edge, it would imply that $r < r'$. We will show a contradiction.

Recall the visibility orders of the algorithm definition in Figure 17. Also recall that the two `tryProd`s are related by a read of the writer index. Intuitively, we use the visibility orders to ensure that the second `tryProd` is aware of the second `tryCons` through the first `tryProd`'s read. As a result the first `tryProd` cannot ignore the second `tryCons` in favor of a coherence order earlier write. Said another way, the second `tryProd` would have to read

back into the past to see a state where r' is less than r. This results in the cyclic coherence order edges between the two writes to the reader index as shown in Figure 18b and in turn, a contradiction. Thus we have shown that $r' < r$.

6.4 Lemma 6 and Theorems 7 and 8

Now we can complete the proof of Lemma 6 by unifying the invariants for `tryProd` and `tryCons`, We will show that when a `tryCons` invocation reads the buffer at a particular index written by a companion `tryProd` invocation, the reader and writer indices used to calculate the index are the same. This proof brings together the invariants of Lemmas 10 and its `tryCons` counterpart.

By the semantics of reads, if a write and read of a buffer memory location are associated then the calculated index used to determine the memory location in the associated `tryProd` and `tryCons` invocations must be the same. Let the reader index value used in the `tryCons` calculation be r and the writer index value used in the `tryProd` calculation be w. If the reader and writer indices are not the same, then by the calculations on lines 10 of `tryCons` and 11 of `tryProd` we know that, $r = w + x \times N$, for some integer $x \neq 0$.

We consider the cases for x, first $x < 0$. Then $r = w + x \times N$ implies $r \leq w - N$. We know that if the `tryProd` containing the write to the buffer executed completely then `w` and its own r' are related according to Lemma 10 with $w < r' + N$. Then we have $w - N < r'$, and by assumption we have $r \leq w - N$, which gives us $r \leq w - N < r'$ and $r < r'$.

Figure 18c illustrates the contradiction in these two reads. By reader index monotonicity and because $r < r'$, wherever the `tryProd` got its reader index r' is coherence order after the `tryCons` that computed its index location using r. This is the co edge pointed downward. But, we can establish, through the read of the buffer and the visibility orders, that the coherence order later write happened-before the coherence order earlier write and thereby derive a contradiction. Intuitively, we ensure the visibility of the "future" write through the specified visibility orders. That is, through the visibility orders, the read learns about a write that has yet to take place.

The case for $x > 0$ is similar but uses the `tryCons` invariant to prove that the `tryCons` must have seen a writer index coherence order after another `tryProd` write to the same buffer index. In turn, this implies that it would have ignored the new state of that index to read into the past.

With Lemma 6 in hand we can prove the two main theorems. Theorem 7 follows from the fact that the two executed `tryCons` procedures must have read different reader index values. Then by Lemma 6 their paired `tryProd`s must also write distinct writer indices and thus be in coherence order. Theorem 8 follows from two facts. First, the two `tryProd`s must have used writer indices that are equal modulo N. Second, the later `tryProd` must have seen a reader index that was greater than its writer index less the buffer length, $w - N < r$. Since r is larger than $w - N$ there must be some `tryCons` for any $r \leq w - N$ and by Lemma 6 it must have read from the earlier `tryProd`'s index write.

7 Related Work

Though our semantics and logic are the first to address Java Opaque Mode, our work follows in a long line of related research on logics for concurrent program semantics.

7.1 Semantics

The story of weak memory model research is a story of tradeoffs. All commodity hardware is weaker than sequential consistency (x86 [37], POWER [41]) and compilers introduce

additional weak behaviors through aggressive optimizations (C++ [3, 25], Java [33]). To find general semantic models that accomodate both, researchers have been forced to either, admit bizarre behaviors that make reasoning difficult or impossible (e.g. thin air reads, read from untaken branch), sacrifice some optimizations (e.g. rule out load-store ordering as in the JOM), or appeal to complex constructs (e.g. promises [23], event structures [39]).

Our semantics is inspired by the Relaxed Memory Calculus (RMC) of Crary and Sullivan [9]. We do not employ the higher order constructs or execution orders of RMC and our coherence definition is more compact. We have mechanized a proof of the equivalence of co-ww, co-wr, and co-rw and the original formulation of RMC without execution orders [2]. We have also added the co-rr rule to track with C11's coherence definition. Thus, without a total order, the ordering of writes in our model of the JOM is a subset of the modification order definition in the model of C11 from [3]. Additionally, the model of Batty et al. admits thin-air behavior which is forbidden by the JOM.

The C11 semantics presented along with the GPS logic in [47] is stronger than the JOM in that it maintains a total order on writes and uses the stronger release-acquire atomics which provides a larger happens-before relationship.

The original Java Memory Model [33] attempted to admit standard compiler optimizations while ruling out thin-air reads. The "causality" mechanisms by which the model prevents thin-air reads while admitting many optimizations made it complex and later work [42] showed that it forbids some standard compiler optimizations.

More recently, progress has been made on new memory models that more successfully support standard compiler optimizations without the problem of thin-air reads [21, 39, 23]. However, without $\mathsf{acyclic}(\mathsf{po} \cup \mathsf{rf})$, all of these models rely on complex formal constructs, while our semantics remains relatively simple.

The semantics of Alglave and Cousot [2] is flexible in that it can be used to specify many models on a continuum from the "anarchic semantics", where virtually anything is possible, up to sequential consistency. Critically our focus is not on generality but on the JOM.

7.2 Logics

In the presence of a very weak-memory model like the JOM, a key problem for logics that reason with state is ensuring a consistent ordering of writes in memory, across threads. The best work on capturing write orderings using state is the work of Turon et al. [47]. They combine separation logic, ghost variables and protocols in a powerful logic for verifying algorithms running on the release/acquire fragment of C11. GPS has been used to verify many algorithms including the RingBuffer that we have presented and the linux RCU algorithm [45]. Our specification for Ring Buffer is stronger, in that Lemma 6 can be used to show that reordering within the buffer is impossible. On the other hand, GPS leverages ghost state to do the proof with integer values that wrap (overflow) which our logic cannot do. We also do not handle allocation.

The protocols of GPS, which are, in the words of authors, the lifeblood of prior concurrency logics require a total order on writes to the same location in their proofs of soundness. Moreover, even assuming a total order on writes, the protocols of GPS cannot be used to prove correctness of some algorithms. As an example consider Peterson's lock.

In Peteron's lock, when a thread examines the state of the victim field, the information it learns is not enough to tell whether the other thread's ordering for the two writes to the victim field is the same. This is important because the ordering of the victim writes

[2] See the Coq source file `coq/Related/Gps.v` included as part of the supplementary material.

determines which thread will enter the critical section when they compete. For example, if the first thread sees the second thread's victim write after its own it must have some way to know that the other thread sees the same ordering of writes. Otherwise, the the second thread could see the reverse ordering, read the first threads victim value, and also enter the critical section.

The protocols of GPS are based on partial orders but the victim state can't be encoded as a partial order since it could progress from 0 to 1 or vice versa as both orderings of the writes are possible in an execution.

By contrast, in Figure 19 we have a JOM execution with two competing lock procedures of Peterson. The fact that `pete0` reads from the victim write of `pete1` and knows about its own victim write (gray dashed edges) means that the two are coherence-ordered in memory by co-read (black edge). If `pete1` were to mirror `pete0` and read from `[vic] := 0` it would result in the an opposing coherence order between the two writes, which would be a cycle and a contradiction.

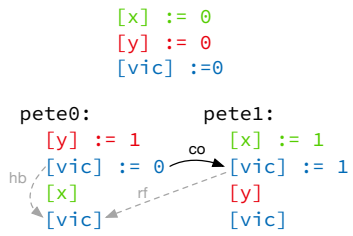


Figure 19 Peterson Victim

The work on GPS is one part of a large body of impressive work on concurrent separation logics for weak memory that includes it's forebearers [50], [49], and [48]. These logics are based on the foundational work of [19] (TTPP), [38] (interference), [22] (rely/guarantee) and [35] (CSL).

More recently [44] have constructed a logic for the promising semantics of [23] which features a version of the C11 semantics with fully relaxed memory accesses and without thin-air reads. Though, in that work, they do not prove correctness of any algorithms. Also of note is the recent work on the Fenced Separation Logic of [14] which is an extension of RSL. The soundness of FSL requires a restriction of the C11 memory model which prevents read-write reorderings by a compiler which is one possibly way to satisfy $\mathsf{acyclic}(\mathsf{po} \cup \mathsf{rf})$.

The work of Alglave and Cousot [2] aims to enable proofs for many memory models. In their proof method they specify a set of "bad reads" that will cause the algorithm to fail and prove correctness under the absence of those reads. Then a target memory model must rule out those reads. In the presence of a very weak memory model that set of bad reads must be small or empty, placing the burden on the algorithm to forbid such reads by using fences or other synchronization methods. The result would be either, a performance penalty when executing the algorithm on strong memory models that already forbid those reads or the work of extra proofs and implementations for those stronger memory models. By contrast, our goal is to construct proofs for the JOM. However, it is worth noting that the addition of specified orders means that a compiler like that of [43] can take our specified orders and provide fast, correct executable code for any "stronger" target memory model. That is, we do one proof for many memory models which is critical for Java, where the intent is to "write once, run anywhere".

Outside the context of weak memory, researchers leveraged the early work of [35] to great effect. The works of [15] and [16] supported dynamically allocated and re-entrant locks respectively. Message passing has been considered in [52] and [5]. Fork and join support appeared in [13]. Finally, in [31] they construct a logic to prove liveness properties for concurrent objects building on their work in [30].

7.3 Thin-air reads

Thin-air reads invalidate basic thread-local state based reasoning as noted in [50]. The semantics of the JOM does not permit thin-air reads by requiring that all executions satisfy

acyclic(po ∪ rf), but our approach to reasoning does not rely on that guarantee and our logic is sound in the presence of thin-air reads.

Instead, write elimination starts from an over-approximation of possible writes to a memory location. In our proofs we make no attempt to rule out writes based on data or control dependencies and so we would include the, intuitively impossible, writes that appear in the classic thin-air read examples, e.g. Figure 1c.

To address such cycles we would require specified visibility orders (vo) to augment the happens-before relationship to rule out such writes by deriving cycles in the coherence order. As a result, we may need extraneous visibility orders to rules out writes that, in practice, can never satisfy a read. Thus, our reasoning principles remain sound but in some cases performance may suffer when the orders are compiled to synchronization.

The effect of the extra visibility orders would certainly be obviated by the Java compiler's own implementation of acyclic(po ∪ rf), but the specific approach would affect the semantics. In recent work Ou et al. [36] showed that a compiler targeted at annotated accesses, like the JOM's `VarHandle` API, can eliminate thin-air reads with a worst case overhead of 6.3% for many data structures by forbidding read-write reordering. Adopting this approach would mean extending the definition of vo in our semantics to include all read-write pairs in program order.

Importantly, the ability to reason in the presence of thin-air reads differentiates our logic from recent work by Raad et al. which is also focused on concurrent libraries for weak memory models [40]. On the other hand, their framework is more general with respect to the memory model as they can simply specify it as another library. Further, their framework can handle libraries without obvious linearization points which we do not address.

7.4 Specified Orders

For a long time, researchers have known that correctness can depend critically on the execution order of two instructions. Fences are a crude way of ensuring that two instructions execute in order. Kuperstein, Vechev, and Yahav [24] used a notion of execution orders as the output of a synthesis algorithm. Like us, they see these orders as part of a correct program, but inferred from a correctness property, rather than specified. The idea of specified orders appeared for first time in publications in 2014–2015, namely in the 2014 PhD dissertation of [29], in the POPL 2015 paper by [9], and in the OOPSLA 2015 paper [6] Crary and Sullivan's POPL 2015 paper introduced the RMC memory model together with a semantic foundation that includes specified orders. More recently a "Placed Before" intra-thread ordering relation was proposed for the C++ concurrency standard [32]. It captures the key idea of the visibility ordering (specifying ordering dependencies explicitly) but with a focus on ruling out thin-air reads.

8 Conclusion

Here, we have presented the first formal model of the JOM. We have also presented a sound, stateless logic for reasoning about the correctness of lock free concurrent algorithms executing on the JOM. As examples, we proved the correctness of Dekker and RingBuffer.

Java also supports other access modes through the `VarHandle` API, and we expect that our semantics is a good basis from which to model them. Further, we think that the simplicity of the judgements in our logic and the nature of our proof method make automation an attractive possibility. We plan to explore these ideas in future research.

References

1. Varhandle java se 9 & jdk 9), 2018. [Online, accessed December 2018]. URL: https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/VarHandle.html.
2. Jade Alglave and Patrick Cousot. Ogre and pythia: An invariance proof method for weak consistency models. *SIGPLAN Not.*, 52(1):3–18, January 2017. URL: http://doi.acm.org/10.1145/3093333.3009883, doi:10.1145/3093333.3009883.
3. Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. *SIGPLAN Not.*, 46(1):55–66, January 2011. URL: http://doi.acm.org/10.1145/1925844.1926394, doi:10.1145/1925844.1926394.
4. Mark Batty and Peter Sewell. The thin-air problem, 2014. [Online, accessed Dec 2018]. URL: http://www.cl.cam.ac.uk/~pes20/cpp/notes42.html.
5. Christian J. Bell, Andrew W. Appel, and David Walker. Concurrent separation logic for pipelined parallelization. In *Proceedings of the 17th International Conference on Static Analysis*, SAS'10, pages 151–166, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1882094.1882104.
6. John Bender, Mohsen Lesani, and Jens Palsberg. Declarative fence insertion. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 367–385, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2814270.2814318, doi:10.1145/2814270.2814318.
7. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 23–42, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
8. Jonathan Corbet. Access_once(), 2012. [Online, accessed December 2018]. URL: https://lwn.net/Articles/508991/.
9. Karl Crary and Michael Sullivan. A calculus for relaxed memory. In *Proceedings of POPL'15, ACM Symposium on Principles of Programming Languages*, 2015.
10. Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. Tada: A logic for time and data abstraction. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 207–231, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
11. Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1883978.1884012.
12. Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 363–377, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
13. Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP'09)*, pages 363–377, Berlin, Heidelberg, 2009. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-00590-9_26, doi:10.1007/978-3-642-00590-9_26.
14. Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with fsl++. In Hongseok Yang, editor, *Programming Languages and Systems: 26th European Symposium on Programming, ESOP 2017*, pages 448–475. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017. URL: https://doi.org/10.1007/978-3-662-54434-1_17, doi:10.1007/978-3-662-54434-1_17.
15. Alexey Gotsman, Josh Berdine, and Byron Cook. Precision and the conjunction rule in concurrent separation logic. *Electron. Notes Theor. Comput. Sci.*, 276:171–190, September

2011. URL: http://dx.doi.org/10.1016/j.entcs.2011.09.021, doi:10.1016/j.entcs.2011.09.021.

16 Christian Haack, Marieke Huisman, and Clément Hurlin. Reasoning about java's reentrant locks. In G. Ramalingam, editor, *Programming Languages and Systems*, pages 171–187, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

17 Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

18 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. URL: http://doi.acm.org/10.1145/78969.78972, doi:10.1145/78969.78972.

19 C. A. R. Hoare. Towards a theory of parallel programming. In Per Brinch Hansen, editor, *The Origin of Concurrent Programming*, pages 231–244. Springer-Verlag New York, Inc., New York, NY, USA, 1972. URL: http://dl.acm.org/citation.cfm?id=762971.762978.

20 David Howells and Paul E. McKenney. Circular buffers, 2017. [Online, accessed July 2017]. URL: https://www.kernel.org/doc/Documentation/circular-buffers.txt.

21 Alan Jeffrey and James Riely. On thin air reads towards an event structures model of relaxed memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 759–767, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2933575.2934536, doi:10.1145/2933575.2934536.

22 C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983. URL: http://doi.acm.org/10.1145/69575.69577, doi:10.1145/69575.69577.

23 Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of POPL'17, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, 2017.

24 Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 111–120, Austin, TX, 2010. URL: http://dl.acm.org/citation.cfm?id=1998496.1998518.

25 Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++11. *SIGPLAN Not.*, 52(6):618–632, June 2017. URL: http://doi.acm.org/10.1145/3140587.3062352, doi:10.1145/3140587.3062352.

26 Doug Lea. Jep 193: Variable handles, 2017. [Online, accessed December 2018]. URL: http://openjdk.java.net/jeps/193.

27 Doug Lea. Using jdk 9 memory order modes, 2018. [Online, accessed December 2018]. URL: http://gee.cs.oswego.edu/dl/html/j9mm.html.

28 K. Rustan Leino, Peter Müller, and Jan Smans. Foundations of security analysis and design v. chapter Verification of Concurrent Programs with Chalice, pages 195–222. Springer-Verlag, Berlin, Heidelberg, 2009. URL: http://dx.doi.org/10.1007/978-3-642-03829-7_7, doi:10.1007/978-3-642-03829-7_7.

29 Mohsen Lesani. *On the Correctness of Transactional Memory Algorithms*. PhD thesis, UCLA, 2014.

30 Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 459–470, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2491956.2462189, doi:10.1145/2491956.2462189.

31 Hongjin Liang and Xinyu Feng. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL '16, pages 385–399, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2837614.2837635, doi:10.1145/2837614.2837635.

32 Daniel Lustig. P1239r0 - placed before.

33 Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM. URL: http://doi.acm.org/10.1145/1040305.1040336, doi:10.1145/1040305.1040336.

34 Aleksandar Nanevski, Ruy Ley-Wild, Ilya Sergey, and Germán Andrés Delbianco. Communicating state transition systems for fine-grained concurrent resources. In Zhong Shao, editor, *Programming Languages and Systems*, pages 290–310, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

35 Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007. URL: http://dx.doi.org/10.1016/j.tcs.2006.12.035, doi:10.1016/j.tcs.2006.12.035.

36 Peizhao Ou and Brian Demsky. Towards understanding the costs of avoiding out-of-thin-air results. *Proc. ACM Program. Lang.*, 2(OOPSLA):136:1–136:29, October 2018. URL: http://doi.acm.org/10.1145/3276506, doi:10.1145/3276506.

37 Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: X86-tso. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-03359-9_27, doi:10.1007/978-3-642-03359-9_27.

38 Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6(4):319–340, December 1976. URL: http://dx.doi.org/10.1007/BF00268134, doi:10.1007/BF00268134.

39 Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. *SIGPLAN Not.*, 51(1):622–633, January 2016. URL: http://doi.acm.org/10.1145/2914770.2837616, doi:10.1145/2914770.2837616.

40 Azalea Raad, Marko Doko, Lovro Rožić, Ori Lahav, and Viktor Vafeiadis. On library correctness under weak memory consistency: Specifying and verifying concurrent libraries under declarative consistency models. *Proc. ACM Program. Lang.*, 3(POPL):68:1–68:31, January 2019. URL: http://doi.acm.org/10.1145/3290381, doi:10.1145/3290381.

41 Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. *SIGPLAN Not.*, 46(6):175–186, June 2011. URL: http://doi.acm.org/10.1145/1993316.1993520, doi:10.1145/1993316.1993520.

42 Jaroslav Ševčík and David Aspinall. On validity of program transformations in the Java memory model. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*, pages 27–51. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. URL: https://doi.org/10.1007/978-3-540-70592-5_3, doi:10.1007/978-3-540-70592-5_3.

43 Michael J. Sullivan. Low-level concurrent programming using the relaxed memory calculus, 2015. [Online, accessed July 2017]. URL: https://www.msully.net/stuff/thesprop.pdf.

44 Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. A separation logic for a promising semantics. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 357–384, Cham, 2018. Springer International Publishing.

45 Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In *PLDI*, 2015.

46 Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. *SIGPLAN Not.*, 48(9):377–390, September

2013. URL: http://doi.acm.org/10.1145/2544174.2500600, doi:10.1145/2544174.2500600.

47 Aaron Turon, Viktor Vafeiadis, and Derek Dreyer. Gps: Navigating weak memory with ghosts, protocols, and separation. In *Proceedings of OOPSLA'14, Object-Oriented Programming Systems, Languages and Applications*, 2014.

48 Viktor Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-726.pdf.

49 Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electron. Notes Theor. Comput. Sci.*, 276:335–351, September 2011. URL: http://dx.doi.org/10.1016/j.entcs.2011.09.029, doi:10.1016/j.entcs.2011.09.029.

50 Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for c11 concurrency. In *Proceedings of OOPSLA'13, Object-Oriented Programming Systems, Languages and Applications*, 2013.

51 Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *Proceedings of the 18th International Conference on Concurrency Theory*, CONCUR'07, pages 256–271, Berlin, Heidelberg, 2007. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=2392200.2392220.

52 Jules Villard, Étienne Lozes, and Cristiano Calcagno. Proving copyless message passing. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 194–209, Berlin, Heidelberg, 2009. Springer-Verlag. URL: http://dx.doi.org/10.1007/978-3-642-10672-9_15, doi:10.1007/978-3-642-10672-9_15.

A Semantics

The following constitutes the full semantics of our model of the JOM.

A.1 Expression Rules

$$\frac{k = n + m}{n + m \xrightarrow{\emptyset} k}\ \text{add} \qquad \frac{k = n \bmod m}{n \bmod m \xrightarrow{\emptyset} k}\ \text{mod} \qquad \frac{n = m}{n == m \xrightarrow{\emptyset} 1}\ \text{eq} \qquad \frac{n \neq m}{n == m \xrightarrow{\emptyset} 0}\ \text{neq}$$

$$\frac{e_1 \xrightarrow{d} e_1'}{\texttt{let } x := e_1 \texttt{ in } e_2 \xrightarrow{d} \texttt{let } x := e_1' \texttt{ in } e_2}\ \text{let-left} \qquad \frac{e_2 \xrightarrow{d} e_2' \quad e_1 \texttt{ init} \quad x \notin FV(d)}{\texttt{let } x := e_1 \texttt{ in } e_2 \xrightarrow{d} \texttt{let } x := e_1 \texttt{ in } e_2'}\ \text{let-right}$$

$$\frac{}{\texttt{let } x := n \texttt{ in } e_2 \xrightarrow{\emptyset} [n/x]e_2}\ \text{let-subst} \qquad \frac{}{\texttt{repeat } e \texttt{ end} \xrightarrow{\emptyset} \texttt{let } x := e \texttt{ in if } x \texttt{ then } x \texttt{ else repeat } e \texttt{ end}}\ \text{repeat}$$

$$\frac{n \neq 0}{\texttt{if } n \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\emptyset} e_1}\ \text{if-right} \qquad \frac{}{\texttt{if } 0 \texttt{ then } e_1 \texttt{ else } e_2 \xrightarrow{\emptyset} e_2}\ \text{if-left}$$

$$\frac{}{l{:}n \xrightarrow{\emptyset} n}\ \text{label-rm} \qquad \frac{e \xrightarrow{d} e'}{l{:}e \xrightarrow{d'} l{:}e'}\ \text{label-under}$$

$$\frac{}{[s/x]e \xrightarrow{\emptyset} s = n \texttt{ in } [n/x]e}\ \text{spec} \qquad \frac{e \xrightarrow{d} e'}{s = n \texttt{ in } e \xrightarrow{d} s = n \texttt{ in } e'}\ \text{spec-under} \qquad \frac{}{n = n \texttt{ in } e \xrightarrow{\emptyset} e}\ \text{spec-rm}$$

$$\frac{}{a \xrightarrow{\epsilon:i=a} i}\ \text{action-init} \qquad \frac{}{i \xrightarrow{i \texttt{ to } n} n}\ \text{exec-read} \qquad \frac{}{i \xrightarrow{i} 0}\ \text{exec-write}$$

A.2 State and Thread Rules

$$\frac{P \xrightarrow{d@p} P' \quad H \xrightarrow{d@p} H'}{(P, H) \to (P', H')}\ \text{step}$$

$$\frac{e \xrightarrow{d} e'}{P; p : \texttt{fork } e \xrightarrow{d@p} P; p : \texttt{fork } e'}\ \text{P-step} \qquad \frac{P \xrightarrow{d@p_1} P'}{P; p_2 : \texttt{fork } e \xrightarrow{d@p_1} P; p_2 : \texttt{fork } e}\ \text{P-find}$$

A.3 Init Rules

The intialization rules and the use of `init` in let-right rule forces initialization occur in program order.

$$\frac{}{n\ \mathsf{init}}\ \text{init-nat} \qquad \frac{e_1\ \mathsf{init} \quad e_2\ \mathsf{init}}{\texttt{let } x := e_1 \texttt{ in } e_2\ \mathsf{init}}\ \text{init-let} \qquad \frac{}{i\ \mathsf{init}}\ \text{init-id}$$

$$\frac{e\ \mathsf{init}}{s = n \texttt{ in } e\ \mathsf{init}}\ \text{init-spec} \qquad \frac{e\ \mathsf{init}}{s = n \texttt{ in } e\ \mathsf{init}}\ \text{init-spec} \qquad \frac{e\ \mathsf{init}}{l : e\ \mathsf{init}}\ \text{init-label}$$

A.4 History Rules

$$H(h) \triangleq h \in H$$

$$\frac{}{H \xrightarrow{\emptyset@p} H}\ \text{hist-empty} \qquad \frac{\neg H(\mathsf{init}(i,p))}{H \xrightarrow{i=a@p} H, \mathsf{init}(i,p), \mathsf{is}(i,a)}\ \text{hist-init}$$

$$\frac{\mathsf{wf}(H,\mathsf{exec}(i),p) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H,\mathsf{exec}(i)})}{H \xrightarrow{i@p} H,\mathsf{exec}(i)}\ \text{hist-write} \qquad \frac{\mathsf{wf}(H,\mathsf{rf}(i_w,i),p,n) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{po}\cup\mathsf{rf}}_{H,\mathsf{rf}(i_w,i)}) \quad \mathsf{acyclic}(\xrightarrow{\mathsf{co}}_{H,\mathsf{rf}(i_w,i)})}{H \xrightarrow{i\ \mathsf{to}\ n@p} H,\mathsf{rf}(i_w,i)}\ \text{hist-read}$$

$$\mathsf{wf}(H,\mathsf{exec}(i),p) \triangleq \begin{cases} H(\mathsf{init}(i,p)) \\ H(\mathsf{is}(i,[l] := \_)) \\ \neg H(\mathsf{exec}(i)) \end{cases} \qquad \mathsf{wf}(H,\mathsf{rf}(i_w,i),p,n) \triangleq \begin{cases} H(\mathsf{init}(i,p)) \\ H(\mathsf{is}(i,[l])) \\ \neg H(\mathsf{rf}(\_,i)) \\ H(\mathsf{is}(i_w,[l] := n)) \\ H(\mathsf{exec}(i_w)) \end{cases}$$

A.5 History Relations

$$i_1 \xrightarrow{\mathsf{po}}_H i_2 \triangleq \exists H_1\ H_2\ H_3\ p,\ H = H_1, \mathsf{init}(i_1,p), H_2, \mathsf{init}(i_2,p), H_3$$

$$i_1 \xrightarrow{\mathsf{to}}_H i_2 \triangleq \exists H_1\ H_2\ H_3,\ H = H_1, \mathsf{exec}(i_1), H_2, \mathsf{exec}(i_2), H_3$$

$$i_1 \xrightarrow{\mathsf{rf}}_H i_2 \triangleq H(\mathsf{rf}(i_1,i_2))$$

$$i_1 \xrightarrow{\mathsf{svo}}_H i_2 \triangleq H(\mathsf{vo}(i_1,i_2))$$

$$i_1 \xrightarrow{\mathsf{push}}_H i_2 \triangleq H(\mathsf{push}(i_1,i_2))$$

$$i_1 \xrightarrow{\mathsf{vo}}_H i_2 \triangleq i_1 \xrightarrow{\mathsf{rf}}_H i_2 \vee i_1 \xrightarrow{\mathsf{svo}}_H i_2 \vee i_1 \xrightarrow{\mathsf{push}}_H i_2 \vee (\exists i_3\ i_4, i_1 \xrightarrow{\mathsf{push}}_H i_3 \wedge i_4 \xrightarrow{\mathsf{push}}_H i_2 \wedge i_1 \xrightarrow{\mathsf{to}}_H i_4)$$

$$\frac{i_1 \xrightarrow{\mathsf{vo}}_H i_2}{i_1 \xrightarrow{\mathsf{vo}^+}_H i_2}\ \text{vo-step}$$

$$\frac{i_1 \xrightarrow{\mathsf{vo}}_H i_3 \quad i_3 \xrightarrow{\mathsf{vo}^+}_H i_2}{i_1 \xrightarrow{\mathsf{vo}^+}_H i_2}\ \text{vo-trans}$$

$$i_1 \xrightarrow{\mathsf{hb}}_H i_2 \triangleq i_1 \xrightarrow{\mathsf{vo}^+}_H i_2 \vee i_1 \xrightarrow{\mathsf{po}}_H i_2$$

A.6 Coherence Rules

$$\mathsf{writes}(H,i,l,n) \triangleq H(\mathsf{is}(i,[l] := n)) \wedge H(\mathsf{exec}(i))$$

$$\mathsf{reads}(H,i,l) \triangleq H(\mathsf{is}(i,[l])) \wedge H(\mathsf{rf}(\_,i))$$

$$\frac{\begin{array}{ll} & \mathsf{writes}(H,i_1,l,v_1) \\ i_1 \xrightarrow{\mathsf{hb}}_H i_2 & \mathsf{writes}(H,i_2,l,v_2) \end{array}}{i_1 \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-ww} \qquad \frac{\begin{array}{ll} & i_1 \neq i_2 \\ i_1 \xrightarrow{\mathsf{hb}}_H i_r & \mathsf{writes}(H,i_1,l,v) \\ i_2 \xrightarrow{\mathsf{rf}}_H i_r & \mathsf{reads}(H,i_r,l) \end{array}}{i_1 \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-wr}$$

$$\frac{\begin{array}{ll} & \mathsf{writes}(H,i_1,l,v_1) \\ i_1 \xrightarrow{\mathsf{hb}}_H i_r & \mathsf{reads}(H,i_r,l) \\ i_r \xrightarrow{\mathsf{po}}_H i_2 & \mathsf{writes}(H,i_2,l,v_2) \end{array}}{i_1 \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-rw} \qquad \frac{\begin{array}{lll} i_1 \xrightarrow{\mathsf{rf}}_H i_{r_1} & & i_1 \neq i_2 \\ i_2 \xrightarrow{\mathsf{rf}}_H i_{r_2} & & \mathsf{writes}(H,i_1,l,v_1) \\ i_{r_1} \xrightarrow{\mathsf{po}}_H i_{r_2} & & \mathsf{writes}(H,i_2,l,v_2) \end{array}}{i_1 \xrightarrow{\mathsf{co}}_H i_2}\ \text{co-rr}$$

B Logic

$$\frac{\Gamma \vdash a_1 \vee a_2 \quad \Gamma \vdash a_1 \Rightarrow a_3 \quad \Gamma \vdash a_2 \Rightarrow a_3}{\Gamma \vdash a_3}\text{rml-disj-elim} \qquad \frac{\Gamma \vdash a_1}{\Gamma \vdash a_1 \vee a_2}\text{rml-disj-lintro} \qquad \frac{\Gamma \vdash a_2}{\Gamma \vdash a_1 \vee a_2}\text{rml-disj-rintro}$$

$$\frac{\Gamma \vdash a_1 \vee a_2 \quad \Gamma \vdash \neg a_1}{\Gamma \vdash a_2}\text{rml-disj-syl} \qquad \frac{\Gamma \vdash \mathsf{false}}{\Gamma \vdash a}\text{rml-exfalso} \qquad \frac{(\Gamma; a_1 \vdash a_2)}{\Gamma \vdash a_1 \Rightarrow a_2}\text{rml-impl-intro} \qquad \frac{\Gamma \vdash a_1 \Rightarrow a_2 \quad \Gamma \vdash a_1}{\Gamma \vdash a_2}\text{rml-mp}$$

$$\frac{\Gamma \vdash a_1 \quad \Gamma \vdash a_2}{\Gamma \vdash a_1 \wedge a_2}\text{rml-conj-intro} \qquad \frac{\Gamma \vdash a_1 \wedge a_2}{(\Gamma \vdash a_1)}\text{rml-conj-lelim} \qquad \frac{\Gamma \vdash a_1 \wedge a_2}{\Gamma \vdash a_2}\text{rml-conj-relim}$$

$$\frac{\Gamma \vdash V_1 \doteq V_2}{\Gamma \vdash V_2 \doteq V_1}\text{rml-geq-sym} \qquad \frac{\Gamma \vdash V_1 \doteq V_2 \quad \Gamma \vdash V_2 \doteq V_3}{\Gamma \vdash V_1 \doteq V_3}\text{rml-geq-trans} \qquad \frac{\Gamma \vdash L \doteq n_1 \quad \Gamma \vdash L \doteq n_2 \quad n_1 \neq n_2}{\Gamma \vdash \mathsf{false}}\text{rml-nope}$$

$$\frac{}{\Gamma \vdash \neg V \dot{<} V}\text{rml-glt-irr} \qquad \frac{\Gamma \vdash V_1 \dot{<} V_3 \quad \Gamma \vdash V_3 \dot{<} V_2}{\Gamma \vdash V_1 \dot{<} V_2}\text{rml-glt-trans} \qquad \frac{\Gamma \vdash V_1 \dot{<} V_2}{\Gamma \vdash \neg V_2 \dot{<} V_1}\text{rml-glt-asym}$$

$$\frac{\Gamma \vdash n \mathbin{@} L}{\Gamma \vdash L \doteq n}\text{rml-const} \qquad \frac{\Gamma \vdash s_1 + s_2 \mathbin{@} L \quad \Gamma \vdash (L; l_{opl}) \doteq n_1 \quad \Gamma \vdash (L; l_{opr}) \doteq n_2}{\Gamma \vdash L \doteq (n_1 [+] n_2)}\text{rml-add} \qquad \frac{\Gamma \vdash s_1 == s_2 \mathbin{@} L \quad \Gamma \vdash (L; l_{opl}) \doteq n_1 \quad \Gamma \vdash (L; l_{opr}) \doteq n_2 \quad \Gamma \vdash n_1 = n_2}{\Gamma \vdash L \doteq 1}\text{rml-eq}$$

$$\frac{\Gamma \vdash s_1 \,\mathtt{mod}\, s_2 \mathbin{@} L \quad \Gamma \vdash (L; l_{opl}) \doteq n_1 \quad \Gamma \vdash (L; l_{opr}) \doteq n_2}{\Gamma \vdash L \doteq (n_1 \ [\mathsf{mod}]\ n_2)}\text{rml-mod} \qquad \frac{\Gamma \vdash (\mathtt{if}\ x\ \mathtt{then}\ l_1 : e_1\ \mathtt{else}\ l_2 : e_2 \mathbin{@} L}{\Gamma \vdash ((\neg(L; l_{cnd}) \doteq 0) \wedge (L; l_1) \doteq L) \vee (L; l_{cnd}) \doteq 0 \wedge (L; l_2) \doteq L}\text{rml-if}$$

$$\frac{\Gamma \vdash \mathtt{let}\ x := l_1 : e_1\ \mathtt{in}\ l_2 : e_2 \mathbin{@} L \quad \Gamma \vdash x \mathbin{@} (L; l_2; ; L') \quad x \in \mathsf{FV}(e_2)}{\Gamma \vdash (L; l_2; ; L') \doteq (L; l_1)}\text{rml-let-bind} \qquad \frac{\Gamma \vdash \mathtt{let}\ x := l_1 : e_1\ \mathtt{in}\ l_2 : e_2 \mathbin{@} L}{\Gamma \vdash \exists n, (L; l_1) \doteq n}\text{rml-let-left}$$

$$\frac{\Gamma \vdash \mathtt{let}\ x := l_1 : e_1\ \mathtt{in}\ l_2 : e_2 \mathbin{@} L}{\Gamma \vdash (L; l_2) \doteq L}\text{rml-let-right} \qquad \frac{\Gamma \vdash \mathtt{repeat}\ e\ \mathtt{end} \mathbin{@} L}{\Gamma \vdash \exists l, e \mathbin{@} (L; l)}\text{rml-repeat-match}$$

$$\frac{\Gamma \vdash \mathtt{repeat}\ e\ \mathtt{end} \mathbin{@} L}{\Gamma \vdash \exists l, L \doteq (L; l) \wedge (\neg(L; l) \doteq 0)}\text{rml-repeat-break} \qquad \frac{\Gamma \vdash (\exists n, L \doteq n)}{\Gamma \vdash \exists e, e \mathbin{@} L}\text{rml-geq-match}$$

$$\frac{\Gamma \vdash [\,s_1\,] \mathbin{@} L \quad \Gamma \vdash (L; l_{loc}) \doteq l \quad \Gamma \vdash L \doteq n}{\Gamma \vdash \mathsf{reads}(L, l, n)}\text{rml-reads} \qquad \frac{\Gamma \vdash [\,s_1\,] := s_2 \mathbin{@} L \quad \Gamma \vdash (L; l_{loc}) \doteq l \quad \Gamma \vdash (L; l_{val}) \doteq n}{\Gamma \vdash \mathsf{writes}(L, l, n)}\text{rml-writes}$$

$$\frac{\Gamma \vdash \texttt{let}\ x := l_1 : e_1\ \texttt{in}\ l_2 : e_2\ @\ L}{\Gamma \vdash (L; l_1) \xrightarrow{\mathsf{po}} (L; l_2)}\text{rml-po} \qquad \frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{po}} L_2}{\Gamma \vdash (L_1;;L_3) \xrightarrow{\mathsf{po}} (L_2;;L_4)}\text{rml-po-sub}$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{rf}} L_2}{\Gamma \vdash L_1 \xrightarrow{\mathsf{vo}} L_2}\text{rml-vo-rf} \qquad \frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{vo}} L_3 \quad \Gamma \vdash L_3 \xrightarrow{\mathsf{vo}} L_2}{\Gamma \vdash L_1 \xrightarrow{\mathsf{vo}} L_2}\text{rml-vo-trans}$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{po}} L_2}{\Gamma \vdash L_1 \xrightarrow{\mathsf{hb}} L_2}\text{rml-hb-po} \qquad \frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{vo}} L_2}{\Gamma \vdash L_1 \xrightarrow{\mathsf{hb}} L_2}\text{rml-hb-vo} \qquad \frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{push}} L_2 \quad \Gamma \vdash L_3 \xrightarrow{\mathsf{push}} L_4}{\Gamma \vdash L_1 \xrightarrow{\mathsf{hb}} L_4 \vee L_3 \xrightarrow{\mathsf{hb}} L_2}\text{rml-vo-pushes}$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{hb}} L_2 \quad \Gamma \vdash \mathsf{writes}(L_1, l, n_1) \quad \Gamma \vdash \mathsf{reads}(L_r, l, n_2) \quad \Gamma \vdash \mathsf{writes}(L_2, l, n_3)}{\Gamma \vdash L_1 \xrightarrow{\mathsf{co}} L_2}\text{rml-co-ww}$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{hb}} L_r \quad \Gamma \vdash \mathsf{writes}(L_1, l, n_1) \quad \Gamma \vdash L_r \xrightarrow{\mathsf{po}} L_2 \quad \Gamma \vdash \mathsf{reads}(L_r, l, n_2) \quad \Gamma \vdash L_1 \neq L_2}{\Gamma \vdash L_1 \xrightarrow{\mathsf{co}} L_2}\text{rml-co-wr}$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{hb}} L_r \quad \Gamma \vdash \mathsf{writes}(L_1, l, n_1) \quad \Gamma \vdash L_r \xrightarrow{\mathsf{po}} L_2 \quad \Gamma \vdash \mathsf{writes}(L_2, l, n_2)}{\Gamma \vdash L_1 \xrightarrow{\mathsf{co}} L_2}\text{rml-co-rw}$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{rf}} L_{r_1} \quad \Gamma \vdash L_1 \neq L_2 \quad \Gamma \vdash L_2 \xrightarrow{\mathsf{rf}} L_{r_2} \quad \Gamma \vdash \mathsf{writes}(L_1, l, n_1) \quad \Gamma \vdash L_{r_1} \xrightarrow{\mathsf{po}} L_{r_2} \quad \Gamma \vdash \mathsf{writes}(L_2, l, n_2)}{\Gamma \vdash L_1 \xrightarrow{\mathsf{co}} L_2}\text{rml-co-rr}$$

$$\frac{\Gamma \vdash \mathsf{reads}(L_r, l, n)}{\Gamma \vdash \exists L_w, L_w \xrightarrow{\mathsf{rf}} L_r}\text{rml-reads-rf} \qquad \frac{\Gamma \vdash L_w \xrightarrow{\mathsf{rf}} L_r}{\Gamma \vdash \exists\ l\ n, \mathsf{writes}(L_w, l, n) \wedge \mathsf{reads}(L_r, l, n)}\text{rml-rf}$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{co}} L_2 \quad \Gamma \vdash L_2 \xrightarrow{\mathsf{co}} L_1}{\Gamma \vdash \mathsf{false}}\text{rml-co-cycl} \qquad \frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{coi}} L_2}{\Gamma \vdash \neg(L_1 \xrightarrow{\mathsf{co}} L_3 \wedge L_3 \xrightarrow{\mathsf{co}} L_2)}\text{rml-coi} \qquad \frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{coi}} L_2}{\Gamma \vdash L_1 \xrightarrow{\mathsf{co}} L_2}\text{rml-coi-co}$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{co}} L_2 \quad \Gamma \vdash (L_1 \xrightarrow{\mathsf{coi}} L_2) \Rightarrow P\ L_1\ L_2 \quad (\forall L_3, \Gamma \vdash L_1 \xrightarrow{\mathsf{co}} L_3 \Rightarrow P\ L_1\ L_3 \Rightarrow L_3 \xrightarrow{\mathsf{coi}} L_2 \Rightarrow P L_1 L_2)}{\Gamma \vdash P\ L_1\ L_2}\text{rml-co-ind}$$

$$\frac{\Gamma \vdash L_1 \xrightarrow{\mathsf{co}} L_2}{\Gamma \vdash \exists l\, n_1\, n_2,\ \mathsf{writes}(L_1, l, n_1) \wedge \mathsf{writes}(L_2, l, n_2)}\text{rml-co-writes}$$

$$\frac{\Gamma \vdash \mathsf{reads}(L, l_1, n_1) \quad \Gamma \vdash \mathsf{reads}(L, l_2, n_2)}{\Gamma \vdash l_1 = l_2}\text{rml-reads-match-loc} \qquad \frac{\Gamma \vdash \mathsf{reads}(L, l_1, n_1) \quad \Gamma \vdash \mathsf{reads}(L, l_2, n_2)}{\Gamma \vdash n_1 = n_2}\text{rml-reads-match-val}$$

$$\frac{\Gamma \vdash \mathsf{writes}(L, l_1, n_1) \quad \Gamma \vdash \mathsf{writes}(L, l_2, n_2)}{\Gamma \vdash l_1 = l_2}\text{rml-writes-match-loc} \qquad \frac{\Gamma \vdash \mathsf{writes}(L, l_1, n_1) \quad \Gamma \vdash \mathsf{writes}(L, l_2, n_2)}{\Gamma \vdash n_1 = n_2}\text{rml-writes-match-val}$$

C The match Function and actionid Predicate

Note, for the full match function consult the `exmatch` definition in the `Semantics.v` Coq source file in the supplementary material. Here we have elided the parts of the definition that traverses the head of the executions and threads.

$$\mathsf{match}(e, L, e') \triangleq \begin{cases} e = e' & \text{if } L = \varnothing \\ \mathsf{match}(s_1, L', e') & \text{if } e = s_1 + s_2 \wedge L = l_{lopl}; L' \\ \mathsf{match}(s_1, L', e') & \text{if } e = s_1 \bmod s_2 \wedge L = l_{lopl}; L' \\ \mathsf{match}(s_1, L', e') & \text{if } e = s_1 == s_2 \wedge L = l_{lopl}; L' \\ \mathsf{match}(s_2, L', e') & \text{if } e = s_1 + s_2 \wedge L = l_{lopr}; L' \\ \mathsf{match}(s_2, L', e') & \text{if } e = s_1 \bmod s_2 \wedge L = l_{lopr}; L' \\ \mathsf{match}(s_2, L', e') & \text{if } e = s_1 == s_2 \wedge L = l_{lopr}; L' \\ \mathsf{match}(s, L', e') & \text{if } e = [\,s\,] \wedge L = l_{lloc}; L' \\ \mathsf{match}(s_1, L', e') & \text{if } e = [\,s_1\,] := s_2 \wedge L = l_{lloc}; L' \\ \mathsf{match}(s_2, L', e') & \text{if } e = [\,s_1\,] := s_2 \wedge L = l_{lval}; L' \\ \mathsf{match}(e'', L', e') & \text{if } e = l{:}e'' \wedge L = l; L' \\ \mathsf{match}(e'', L, e') & \text{if } e = \texttt{repeat } e'' \texttt{ end} \\ \mathsf{match}(s, L', e') & \text{if } e = \texttt{if } s \texttt{ then } e_1 \texttt{ else } e_2 \wedge L = l_{lcnd}; L' \\ \mathsf{match}(e_1, L, e') & \text{if } e = \texttt{if } s \texttt{ then } e_1 \texttt{ else } e_2 \wedge \neg\mathsf{match}(e_2, L, e') \\ \mathsf{match}(e_2, L, e') & \text{if } e = \texttt{if } s \texttt{ then } e_1 \texttt{ else } e_2 \\ \mathsf{match}(e_1, L, e') & \text{if } e = \texttt{let } x := e_1 \texttt{ in } e_2 \wedge \neg\mathsf{match}(e_2, L, e') \\ \mathsf{match}(e_2, L, e') & \text{if } e = \texttt{let } x := e_1 \texttt{ in } e_2 \\ \mathsf{false} & \text{o/w} \end{cases}$$

As discussed in Section 4, the actionid predicate uses match to track the evolution of a memory access through an execution from expression to identifier to value.

$$\begin{aligned} &\mathsf{actionid}(E, L, i) \triangleq \\ &\quad \exists\, e\, n\, E'\, E'', \\ &\qquad E = E_1;;E_2;;E_3 \\ &\qquad e \in \{[\,s\,], [\,s_1\,] := s_2\} \\ &\qquad \mathsf{match}(E, L, e) \\ &\qquad \mathsf{match}(E_2, L, i) \\ &\qquad \mathsf{match}(E_3, L, n) \end{aligned}$$

D Soundness Examples

We will focus here on proof sketches for the soundness of a few key rules we have featured previously. We refer the interested reader to the supplementary material for more extensive, formal proofs.

reads-rf By assumption we have that a read, L_r evaluated completely. We must show that it is connected to a write that can be located in the program with some L_w. We know that if a read evaluated to a value then it must have some identifier i_r and by hist-read in Figure 4 it will have added $\mathsf{rf}(i_w, i_r)$ to the history. Thus by the assumptions of hist-read, there was some write identified by i_w that executed fully. By the assumption of our labeling discipline we have that it must have been located in the program at the point of it executed

with some L_w. All that remains is to construct $L_w \xrightarrow{\mathsf{rf}} L_r$ using the write and read labels with the derived information about their executions for the actionid predicate.

co-ind By assumption $L_1 \xrightarrow{\mathsf{co}} L_2$. By the definition of $\xrightarrow{R}$ we have that there exists some i_1 and i_2 for L_1 and L_2 such that $i_1 \xrightarrow{\mathsf{co}}_E i_2$. We know that $\xrightarrow{\mathsf{co}}_E$ is well founded since it is acyclic (see Figure 4 hist-write and hist-read) and E is finite. The proof proceeds by induction on $i_1 \xrightarrow{\mathsf{co}}_E i_2$ to show $P'\ i_1\ i_2 \triangleq P\ L_1\ L_2$. Note that, since neither i_1 nor i_2 can be free in P by the syntax of our assertions we can simply use $P\ L_1\ L_2$.

In the base case show, $i_1 \xrightarrow{\mathsf{coi}}_E i_2 \Rightarrow P\ L_1\ L_2$. By assumption, $L_1 \xrightarrow{\mathsf{coi}} L_2 \Rightarrow P\ L_1\ L_2$, so it's enough to show that $i_1 \xrightarrow{\mathsf{coi}}_E i_2 \Rightarrow L_1 \xrightarrow{\mathsf{coi}} L_2$, which follows from the definition of $\xrightarrow{R}$ and the assumptions from the definition of $L_1 \xrightarrow{\mathsf{co}} L_2$. In the inductive case we have as an assumption $P'\ i_1\ i_3$ for i_1 and some i_3. According to the the syntax of our assertions we have immediately that $P\ L_1\ L_2$.

if-alt By assumption we have $L \doteq n$ for some n and $\texttt{if} \ldots @ L$. Then, by the definition of E we have some, possibly empty, sequence of steps, $(P_1, H_1) \rightarrow (P_2, H_2)$ from the head of E where $\texttt{if} \ldots @ L$ to the state where $n @ L$. We proceed by induction on execution steps to show that, if there is a match $\texttt{if} \ldots @ L$ in the first state (holding the subexpressions to be arbitrary) and a match $n @ L$ in the end state, then there exist some pair of states where the disjunction in the conclusion of if-alt holds.

In the base case E is only the initial state. Then both $\texttt{if} \ldots @ L$ and $n @ L$ would be true in the same state implying that $\texttt{if} \ldots = n$, a contradiction.

In the inductive case, we have $(P_1, H_1) \rightarrow (P_3, H_3) \rightarrow^* (P_2, H_2)$. We know that $\texttt{if} \ldots @ L$ in P_1 then it must be that there is some $e @ L$ in P_3. Since the equality of expressions is decideable we consider the cases. If $\texttt{if} \ldots = e$ then the inductive hypothesis applies. Otherwise it must be that there was a substitution or the `if` expression, took a step. Again, we consider the cases. In the case of the substitution we can apply the inductive hypothesis since we held the sub-expressions to be arbitrary in our goal. Otherwise, the `if` took a step. Then we have the two states, (P_1, H_1) and (P_3, H_3), where either, the condition was 1 in P_1 and the `then` branch will be the resulting expression at label L in P_3, or, similarly, the condition was 0 and the `else` branch will be the resulting expression at label L, as required.

E Herlihy/Wing Queue Correctness

The concurrent queue specification of Herlihy and Wing [18] tracks closely with the ordering based approach of memory reasoning in our logic. Here we give short proof sketches for how Lemma 6 can be used to prove the key theorems of their specification.

Theorem 6 Since the `Enq x` "precedes" the `Enq y` (here $\xrightarrow{\mathsf{hb}}$) we can show that the index for the first is smaller than the index for the second by Lemma 9. By Lemma 6 the index of `Deq x` must be smaller than the index `Deq y` therefore we can show that `Deq x` must also have happened before `Deq y` using `tryCons` invariant.

Theorem 7 This is similar to Theorem 8 in that we must show that there is a dequeue (`tryCons`) which executed with the same index as the earlier dequeue. The proof here is easier. The proof of our Theorem 8 must establish that there exists a dequeue action which increments the reader index, allowing the write of writer index modulo N in the second dequeue to reference the same buffer location. Instead, we are given the second dequeue, so we only need to "work backward" using the `tryCons` invariant to show that there must exist an earlier dequeue with the same index as the earlier enqueue and use Lemma 6 to show that it must have read from the earlier enqueue.

Theorem 8 Follows directly from the $\xrightarrow{\mathsf{rf}}$ established during an enqueue for a given buffer location and the definition of $\xrightarrow{\mathsf{hb}}$.