Let go of that which does not serve you: Semantic Independence in a Model of Relaxed Memory

ANONYMOUS AUTHOR(S)

We define a relaxed memory model, building on simple and traditional mathematical foundations. We show that the model (1) satisfies the DRF-SC criterion, (2) compiles to TSO and ARMV8 without fences on relaxed operations, (3) supports all reasonable sequential compiler optimizations, and (4) supports compositional reasoning for temporal safety properties.

CCS Concepts: • **Theory of computation → Parallel computing models**; **Abstraction**;

Additional Key Words and Phrases: Relaxed Memory Models, Hardware Transactional Memory

ACM Reference Format:
Anonymous Author(s). 2019. Let go of that which does not serve you: Semantic Independence in a Model of Relaxed Memory. 1, 1 (July 2019), 38 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Manson et al. [2005] identify the central problem in the design of software relaxed memory models: "The memory model must strike a balance between ease-of-use for programmers and implementation flexibility for system designers." This paper provides a solution.

There are two dimensions to "implementation flexibility":

- The model should be realizable on modern hardware with minimal synchronization. A canonical example is that the relaxed atomics of C11 (or the plain variables of Java) should not require any extra synchronization.
- The model should facilitate compiler transformations. Ideally, the model should support the known transformations used to optimize synchronization free single threaded code. A canonical example is the commuting of independent statements.

There are also two dimensions to "ease-of-use":

- The *data race free-sequentially consistent* (DRF-SC) criterion [Adve and Hill 1990, 1993] permits the programmer to forget about relaxed memory for correctly synchronized programs.
- All programs, including those with data races, should support compositional reasoning on temporal safety properties [Abadi and Lamport 1993; Misra and Chandy 1981; Pnueli 1985; Stark 1985]. The simplest form of such a composition principle is:

$$\frac{\phi, \mathcal{P}_1 \models \phi \qquad \phi, \mathcal{P}_2 \models \phi}{\mathcal{P}_1 \parallel \mathcal{P}_2 \models \phi}$$

  where ϕ is a temporal safety property on read and write actions, and $\phi, \mathcal{P}_1 \models \phi$ indicates that $\mathcal{P}_1$ satisfies ϕ if the environment satisfies ϕ. This cyclical compositional proof rule permits us to reason separately about individual threads.

We illustrate the last criterion with two examples. First, consider the well-known "Out Of Thin Air" (OOTA) litmus test, with all variables initialized to 0:

$$(y := x \parallel x := y) \tag{1}$$

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license.
© 2019 Copyright held by the owner/author(s).
XXXX-XXXX/2019/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Informally, both threads satisfy the invariant that conjoins "A write of 1 to x requires a prior read of 1 from y" and "A write of 1 to y requires a prior read of 1 from x ". If composition holds, the full program satisfies the invariant. Since the variable declaration closes the program from other writes to x, y, we deduce the conjunction of "A write of 1 to x requires a prior write of 1 to x" and "A write of 1 to y requires a prior write of 1 from x" . Thus, we deduce that "A write of 1 to x requires a prior write of 1 to x", and consequently "there is no write of 1 to x". Composability of safety properties provides an *objectively falsifiable* measurement of OOTA in a memory model.

Now we consider the "type unsafety" example, from [Lochbihler 2013, Figure 8]:

$$x\texttt{:=}y \parallel r\texttt{:=}x\texttt{; if(}b\texttt{)\{}s\texttt{:=new C()\} else \{}r\texttt{:=new D()\}; }y\texttt{:=}r\texttt{;} \parallel b\texttt{:=1} \tag{2}$$

Prima facie, the allocation operation can pick the same address for the two new objects, because only one of them occurs in any one execution. This causes type safety to break for racy programs in the JMM, forcing Lochbihler to include the type information in the address itself. Informally, all threads satisfy the invariant that conjoins "a creation of a `C` object is preceded by a read of `b` as 1" and "a creation of a `D` object is preceded by a read of `b` as 0". If composition holds, the full program satisfies the invariant. Thus, composability of safety properties facilitates the proof of a temporal invariant that forbids"pointer forging" within a single execution<.

None of the extant memory models validate both "implementation flexibility" and "ease-of-use".

Models that validate "ease-of-use" include Sequential Consistency (SC) Lamport [1979], Dolan et al. [2018], Lahav et al. [2017], Jeffrey and Riely [2016], RC11 model[Lahav et al. 2017] and Boehm [2018]. However, *all* of them invalidate reordering of independent statements. [Boehm 2018; Dolan et al. 2018; Jeffrey and Riely 2016] forbid breaking of the program order from reads to writes and thus require extra fences after read actions in hardware implementations. The results of Boehm and Demsky [2014] show that the RC11 model[Lahav et al. 2017] forces a dependency or a fence between a relaxed atomic read and a subsequent relaxed atomic write.

As quoted at the beginning of this section, the JMM was designed with these two aims. However, the JMM and the related models of Jagadeesan et al. [2010] and Kang et al. [2017] invalidate compositional reasoning. In §7 we formalize a variant of Lochbihler's type unsafety example which demonstrates this.

Our approach builds on pomsets [Gischer 1988; Plotkin and Pratt 1997], enriched with sequential preconditions [Hoare 1969]. This simple combination of traditional features provides a conceptual foundation for the notion of time in our model of relaxed memory, even while eschewing the now familiar paraphernalia associated with models of relaxed memory, such as mutliple relations and thread ids.

- The acylicity of the pomset provides an axiomatic basis for a global notion of time, as envisioned in chapter 3.3 of Alglave [2010]. The novelty in our framework is that sequential evolution proceeds silently, *without* advancing global time. Thus, we capture the essence of multi-copy atomicity (MCA) in a software memory model, whereas traditionally MCA is explored in hardware memory models. TSO (see, e.g. [Sewell et al. 2010]) and recent architectures, such as ARMV8 (see, e.g. [Pulte et al. 2018]), are MCA, but not older architectures, such as POWER (see, e.g. [Sarkar et al. 2011]) or ARMV7 (see, e.g. [Alglave et al. 2009]).
- The implicit use of Hoare logic in our model enables us to extract and incorporate all the independence information available in sequential code. Intuitively, this facilitates the *implicit* and *dynamic* discovery of permissible program transformations. Thus, the model fully reaps the benefits of viewing a memory model in terms of (sequential) program transformations, eg. see [Demange et al. 2013; Ferreira et al. 2010; Lahav and Vafeiadis 2016; Saraswat et al. 2007], without explicitly being formalized as such.

We show the following.

- The model satisfies the DRF-SC criterion (§4).
- The model compiles to ARMV8 and TSO *without* extra synchronization for raw variables (§5). However, a semantic version of MCA in our model means that a compilation to ARMV7 or POWER requires extra synchronization.
- The model validates a variety of single threaded transformations including reordering of independent statements and roach motel laws for synchronization (§6). We provide two forms of abstract evidence that our model supports as many sequential transformations as can be expected. First, we show a precise relationship with standard Hoare-triples for sequential code. Second, we prove a "full abstraction" style completeness theorem; the inability of our model to transform a top-level thread C into D is *always* justified by a parallel observer who can see an interaction with D that is not supported by C.
- The model validates compositional reasoning on safety properties (§7). Thus, it supports the infrastructure required to prove a realistic type safety theorem that includes racy programs without requiring that that the type information is included in the address itself.

To the reader interested in models that forbid load buffering, we provide a way to adapt our model to forbid the relaxing of the program order from reads to writes, thus modeling [Boehm 2018; Dolan et al. 2018]. Our new contributions for such a reader are an approach to validating data-sensitive compiler optimizations and compositional reasoning of temporal properties.

We give an informal introduction to the model in §2 before presenting the precise formalities in §3. §4 proves the DRF theorem, whereas §5 provides a compilation into ARMV8 and TSO. Single threaded optimizations, and the associated completeness theorems are addressed in §6. §7 describes a temporal logic, and a compositional proof principle for proving safety properties. We end with a discussion of related work in §8.

2 AN INTRODUCTION TO THE MODEL

In this section, we define the model, provide some intuitions about the semantics, and work through a series of illustrative examples. We present precise details of the semantics in §3.

Let $\mathcal{A}$ be a set of *actions*, that includes actions of the form $(\mathsf{R}\,x\,v)$, which *reads* value v from location x, $(\mathsf{R}^{\mathsf{acq}}\,x\,v)$, which is an *acquire* that reads v from x, $(\mathsf{W}\,x\,v)$, which *writes* v to x, and $(\mathsf{W}^{\mathsf{rel}}\,x\,v)$, which is a *release* that writes v to x. Actions that neither acquire nor release are *relaxed*; other actions are *synchronizations*.

The actions listed above are *external*. Each external action has a corresponding *internal* action, denoted by prefixing τ. Internal actions also read and write locations, just as external actions do, but are not used to model communication between threads, so we do not record their value.

Two actions *conflict* if one writes a location and the other either reads or writes the same location.

Let Φ be a set of logical formulae, such as $(x = 1)$ or $(r = s + 1)$, where r and s are register names.

Formulae are *open*, in that occurrences of register names and memory locations are subject to substitutions of the form $\phi[x/r]$ and $\phi[N/x]$, where x is a memory location, r is a register and N is an memory-location-free expression. Actions are not subject to substitution.

Our model is based on *partially ordered multisets* (*pomsets*) [Gischer 1988].

Definition 2.1. A *(memory model) pomset* is a tuple $(E, \leq, \lambda)$, such that

- E is a set of *states*,
- $\lambda : E \longrightarrow (\Phi \times \mathcal{A})$ is a *labeling*, from which we derive $\lambda_\Phi : E \longrightarrow \Phi$ and $\lambda_\mathcal{A} : E \longrightarrow \mathcal{A}$,
- $\leq\ \subseteq (E \times E)$ is a partial order, and
- if $d \leq e$ then $\lambda_\Phi(e)$ implies $\lambda_\Phi(d)$.

In the parlance of chapter 3.3 of Alglave [2010], $\leq$ is a “global happens-before” relation. The restriction of $<$ to conflicting writes is called the coherence order, co.

We refer to “memory model pomsets” as “pomsets”. We write pairs in $(\Phi \times \mathcal{A})$ as $(\phi \mid a)$. We elide ϕ when it is a tautology. We write $d < e$ when $d \leq e$ and $d \neq e$.

We give the semantics of a program as a set of pomsets, where each pomset corresponds to a single execution. We visualize a pomset as a directed graph where the nodes are drawn from E, each node e is labeled with $\lambda(e)$, and order is drawn as an edge. We elide events with unsatisfiable preconditions. As discussed below, we visualize order using arrows that indicate the reason that the order arises. Although we use multiple arrow, we emphasize that they are all part of the same $\leq$ relation.

Dependencies. A write event e has two kinds of dependencies. First, it may have an external dependency on a read d from another thread, which is reflected in the order $d < e$. Second, it may have an internal dependency on a preceding event in the same thread, which is reflected in the formula $\lambda_\Phi(e)$. In order for the write event to “fire,” both these dependencies need to be satisfied. The external dependency on a read is satisfied by globally visible writes. The internal dependency can only be fulfilled by a preceding actions of the same thread.

Intrathread dependencies. Within a single thread, $<$ captures the notion of dependency, indicating that the related events cannot appear in the opposite order.

For example, semantics of $y := r$ contains the following pomset.

$$\boxed{r = 1 \mid \mathsf{W}\,y\,1} \tag{3}$$

that can be viewed as expressing the Hoare triple $\{r = 1\}\ y := r\ \{y = 1\}$. The Hoare triple $\{x = 1\}\ r := x;\ y := r\ \{y = 1\}$ expresses the effect of prefixing an assignment to r. The assignment to r causes a substitution in the precondition, changing the label to $(x = 1 \mid \mathsf{W}\,y\,1)$. The effect of the read is realized in the semantics in two different ways:

$$\begin{tikzcd} \boxed{\mathsf{R}\,x} \arrow[r] & \boxed{x = 1 \mid \mathsf{W}\,y\,1} \end{tikzcd} \qquad \text{and} \qquad \begin{tikzcd} \boxed{\mathsf{R}\,x\,1} \arrow[r] & \boxed{\mathsf{W}\,y\,1} \end{tikzcd}$$

In the pomset on the left, the read is *internal*, and therefore the formula on the write actions must be discharged by a preceding action in the same thread. In this execution, we have included an *internal action*, which we visualize in grey.

In the pomset on the right, the read is *external*. An external read may discharge the precondition on a event that is ordered after: by introducing order $(\mathsf{R}\,x\,1) \rightarrow (x = 1 \mid \mathsf{W}\,y\,1)$, we may substitute 1 for x in the precondition, resulting in $1 = 1$, which is a tautology and therefore elided.

Consider prefixing a write to the above program: $x := 1;\ r := x;\ y := r$. We again have two cases, depending on whether the write to x is *internal* or *external*:

$$\begin{tikzcd} \boxed{\mathsf{W}\,x} & \boxed{\mathsf{R}\,x} \arrow[r] & \boxed{\mathsf{W}\,y\,1} \end{tikzcd} \qquad \text{and} \qquad \begin{tikzcd} \boxed{\mathsf{W}\,x\,1} & \boxed{\mathsf{R}\,x} \arrow[r] & \boxed{\mathsf{W}\,y\,1} \end{tikzcd}$$

This pomset is derived discharging the internal dependency $x = 1$ from the left pomset above in the semantics of $r := x;\ y := r$. This feature is reminiscent of the *read-from internal* (rfi) in hardware memory models, where writes from a thread are forwarded to a read in the same thread without global publication.

The Hoare triple corresponding to the above pomset is $\{\mathsf{true}\}\ x := 1;\ r := x;\ y := r\ \{y = 1 \wedge x = 1\}$. This triple reflects the store forwarding optimization in compilers.

As seen above, preconditions can be weakened as a result of prefixing. As in [Disselkoen et al. 2019], weakening of preconditions[1] can also happen by merging actions. Consider the program `if`

[1]Formula ψ is *weaker* than a formula ϕ if ϕ implies ψ.

$(r)\{y := 1\}\,\texttt{else}\,\{y := 1\}$. We derive:

$$\frac{\{r = 0\}\ y := 1\ \{y = 1\}, \quad \{r \neq 0\}\ y := 1\ \{y = 1\}}{\{\text{true}\}\ \texttt{if}(r)\{y := 1\}\,\texttt{else}\,\{y := 1\}\ \{y = 1\}}$$

This is reflected in our semantics by a pomset with a single action whose precondition is true

$$\boxed{\mathsf{W}\,y\,1}$$

that is derived as follows. The two branches of the conditional contain pomsets as follows:

$$\text{THEN: } \boxed{r = 0 \mid \mathsf{W}\,y\,1} \qquad \text{ELSE: } \boxed{r \neq 0 \mid \mathsf{W}\,y\,1}$$

In the semantics of the conditional, we take the (not necessarily disjoint) union of the events, that permits events with the same actions to be identified by combining their preconditions with a disjunction. In this case, this yields the event $(r \neq 0 \vee r = 0 \mid \mathsf{W}\,y\,1)$, permitting us to derive the desired pomset in the semantics of $\texttt{if}(r)\{y := 1\}\,\texttt{else}\,\{y := 1\}$.

Within a single thread, $<$ also orders *conflicting* events. We refer order arising from conflict as "weak" and visualize it using a dashed arrow "$\dashrightarrow$". This is in contrast to the "strong" order arising from dependency, which we visualize using a solid arrow "$\rightarrow$". For example, the semantics of $x := 0;\ x := 1$ includes the pomset:

$$\begin{tikzcd} \boxed{\mathsf{W}\,x\,0} \arrow[r, dashed] & \boxed{\mathsf{W}\,x\,1} \end{tikzcd} \tag{4}$$

The weak edge $(\mathsf{W}\,x\,0) \dashrightarrow (\mathsf{W}\,x\,1)$ is required since $x := 0$ precedes $x := 1$ in thread order. This notion of conflict is stable under substitution (since actions are not subject to substitution).

Interthread dependencies. Between threads, $<$ includes reads-from, which always goes from writes to reads across threads. We consider reads-from to be a strong order, which we visualize with a thick arrow "$\rightarrow$". indicates that one thread has *read-from* another. Since new threads may introduce order from write to read, we define our semantics to be closed with respect to augmentation of $\leq$.

At top level, we expect that every read event is *fulfilled* by a matching write. We define fulfillment so that it is monotone with respect to addition of new threads.

Definition 2.2. We say d *fulfills* e *on* x if d externally writes v to x, e externally reads v from x,

- $d < e$, and
- if an event c writes to x then either $c \leq d$ or $e \leq c$.

A pomset is x-closed if every external read on x is fulfilled, and every event is independent of x. A pomset is *top-level* if it is x-closed for every location x.

For example, combining (3) and (4), the semantics of $x := 0;\ x := 1 \parallel r := x;\ y := r$ includes:

$$\begin{tikzcd} \boxed{\mathsf{W}\,x\,0} \arrow[r, dashed] & \boxed{\mathsf{W}\,x\,1} \arrow[r] & \boxed{\mathsf{R}\,x\,1} \arrow[r] & \boxed{\mathsf{W}\,y\,1} \end{tikzcd} \tag{5}$$

We often highlight the required strong edge from a write to a read action that it fulfills, as above.

Between threads, $<$ also defines a partial order on conflicting actions; we consider these to be a weak order, visualized "$\dashrightarrow$". Because of the requirements of fulfillment, any writes to x that are read must be totally ordered. As an example consider the following program and execution:

$$x := 3 \parallel \texttt{if}(x = 1)\{r := x\} \parallel x := 1 \parallel x := 2$$

$$\begin{tikzcd} & \boxed{\mathsf{R}\,x\,1} \arrow[dl, dashed] \arrow[dd] \arrow[ddr, dashed] & \boxed{\mathsf{W}\,x\,1} \arrow[l] \arrow[dd, dashed] \\ \boxed{\mathsf{W}\,x\,3} & & \\ & \boxed{\mathsf{R}\,x\,2} \arrow[ul, dashed] & \boxed{\mathsf{W}\,x\,2} \arrow[l] \end{tikzcd}$$

The restriction of $<$ to conflicting events on a single location is called the *extended coherence order*, eco. Since $<$ is a partial order, so is eco.

Synchronization. The model ensures that events are ordered before a release and after an acquire. This is a strong order which is visualized as a solid arrow, like order derived from dependency. As an example, consider the program:

$$x := 0\,;\ x := 1\,;\ y^{\mathsf{rel}} := 1\ \parallel\ r := y^{\mathsf{acq}};\ s := x$$

which allows:
$$\begin{tikzcd}
\mathsf{W}\,x\,0 \arrow[r, dashed] \arrow[dr] & \mathsf{W}\,x\,1 \arrow[r] \arrow[d] & \mathsf{R}\,x\,1 \\
& \mathsf{W}^{\mathsf{rel}}\,y\,1 \arrow[r] & \mathsf{R}^{\mathsf{acq}}\,y\,1 \arrow[u]
\end{tikzcd}$$
but *not*:
$$\begin{tikzcd}
\mathsf{W}\,x\,0 \arrow[r, dashed] \arrow[dr] \arrow[rr, bend left] & \mathsf{W}\,x\,1 \arrow[d] & \mathsf{R}\,x\,0 \\
& \mathsf{W}^{\mathsf{rel}}\,y\,1 \arrow[r] & \mathsf{R}^{\mathsf{acq}}\,y\,1 \arrow[u]
\end{tikzcd}$$

Since $(\mathsf{W}\,x\,0) \le (\mathsf{W}\,x\,1) < (\mathsf{R}\,x\,0)$, this pomset does not satisfy the requirements to fulfill the read. If we replace the release with a plain write, then the outcome $(\mathsf{R}^{\mathsf{acq}}\,y\,1)$ and $(\mathsf{R}\,x\,0)$ is possible:

$$\begin{tikzcd}
\mathsf{W}\,x\,0 \arrow[r, dashed] \arrow[rr, bend left] & \mathsf{W}\,x\,1 & \mathsf{R}\,x\,0 \arrow[l, dashed] \\
& \mathsf{W}\,y\,1 \arrow[r] & \mathsf{R}^{\mathsf{acq}}\,y\,1 \arrow[u]
\end{tikzcd}$$

since no order is required between $(\mathsf{W}\,x\,1)$ and $(\mathsf{W}\,y\,1)$. Symmetrically, if we replace the acquire of the original program with a plain read, then the outcome $(\mathsf{R}\,y\,1)$ and $(\mathsf{R}\,x\,0)$ is possible.

Impact of internal reads. The ability to remove of the program order from a write to a read of the same thread as discussed earlier causes great expressivity in terms of permitted executions. To see this, consider the example 3.6 from [Podkopaev et al. 2019]:

$$r := x\,;\ y^{\mathsf{rel}} := 1\,;\ z := y\ \parallel\ x := z$$

$$\begin{tikzcd}
\mathsf{R}\,x\,1 \arrow[r] & \mathsf{W}^{\mathsf{rel}}\,y\,1 & \mathsf{R}\,y \arrow[r] & \mathsf{W}\,z\,1 \arrow[dl] \\
& \mathsf{W}\,x\,1 \arrow[ul] & \mathsf{R}\,z\,1 \arrow[l] &
\end{tikzcd} \tag{6}$$

Internal actions do not need to be fulfilled by a matching write. As a result, there is no order from $(\mathsf{W}^{\mathsf{rel}}\,y\,1)$ to $(\mathsf{W}\,z\,1)$. This behavior is allowed by ARMV8. Were the internal action made external, then there would be order due to the requirements of fulfillment; this behavior would be disallowed due to the evident cycle:

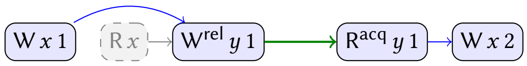
$$\begin{tikzcd}
\mathsf{R}\,x\,1 \arrow[r] & \mathsf{W}^{\mathsf{rel}}\,y\,1 \arrow[r] & \mathsf{R}\,y\,1 \arrow[r] & \mathsf{W}\,z\,1 \arrow[dl] \\
& \mathsf{W}\,x\,1 \arrow[ul] & \mathsf{R}\,z\,1 \arrow[l] &
\end{tikzcd}$$

The explicit use of internal actions facilitate the translation to ARMV8 (§5) and the definitions required in the proof of DRF-SC (§4).

The dependency calculation for $(\mathsf{W}\,z\,1)$ in this execution proceeds as follows. Desugaring, the thread is $(r := x;\ y^{\mathsf{rel}} := 1;\ s := y;\ z := s)$. The semantics of $z := s$ includes a pomset with event $(s = 1 \mid \mathsf{W}\,z\,1)$. The precondition is discharged internally by the write of $y = 1$ without an externally visible dependency to an explicit read: Prefixing $s := y$ transforms the precondition to $y = 1$. As in the discussion of execution (3), the order from $(\mathsf{W}^{\mathsf{rel}}\,y\,1)$ allows the precondition $y = 1$ to be weakened to $1 = 1$, which is a tautology. Note that both write prefixing and read prefixing enable precondition weakening. Only read prefixing imposes order; there is no order from the prefixed write to any subsequent action.

In defining *data races* (and thus in the proof of DRF-SC) it is important to position internal actions relative to synchronization actions. For example, the following execution is consider data-race free.

$$x := 1\,;\ r := x\,;\ y^{\mathsf{rel}} := 1\,;\ \parallel\ s := y^{\mathsf{acq}}\,;\ x := 2$$



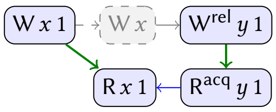
However, if we commute the read of x and the release of y, the resulting program has a data race.

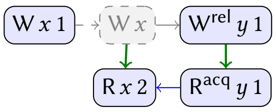
Invalidation of thread inlining. The behavior in execution (6) is not possible if we split the first thread in two: $r := x;\ y^{\mathsf{rel}} := 1 \parallel z := y$. since that would make the read explicit, thus imposing the order shown in gray. This shows that, like the JMM, our model invalidates thread inlining.

Impact of internal writes. In order to validate redundant write removal in §6, we include internal writes in addition to internal reads. Internal writes cannot be used to fulfill an external read, but they can be used to satisfy the precondition on an internal read. In addition, internal writes prevent prior external writes from being used to fulfill a read. Consider the program:

$$x := 1;\ x := 2;\ y^{\mathsf{rel}} := 1 \parallel \mathtt{if}(y^{\mathsf{acq}})\{r := x\}$$

Both of the following executions violate the definition of fulfillment:

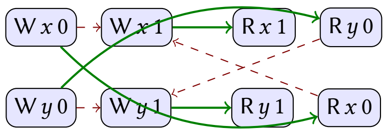




The left execution is not allowed since $(\mathsf{W}\,x\,1) < (\mathsf{W}\,x\,\bot) < (\mathsf{R}\,x\,1)$ and $(\mathsf{W}\,x\,\bot)$ writes x. The right execution is not allowed since $(\mathsf{W}\,x\,\bot)$ is an internal write, not an external one. In this example, if the second write to x is internal, then there is no way to fulfill the read of x by the second thread. Of course, there are always executions where the second write is external, in which case the second thread can read 2 for x.

Multi-copy atomicity. Many of the standard litmus tests for MCA are variants of IRIW (Independent Reads of Independent Writes), such as IRIW with control dependencies between the reads:

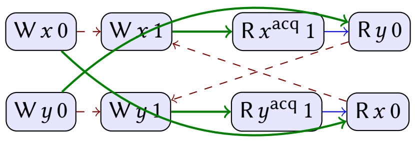
$$\begin{array}{rl} & x := 0;\ x := 1 \parallel \mathtt{if}(x)\{r := y\} \\ \parallel & y := 0;\ y := 1 \parallel \mathtt{if}(y)\{s := x\} \end{array}$$



Since our semantics does not enforce control dependencies between reads, this execution is acyclic and is allowed. If the first read in each thread is acquiring, there is order between the reads.

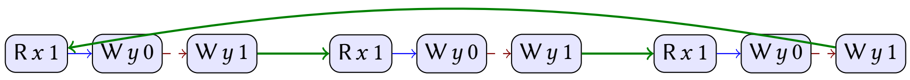
$$\begin{array}{rl} & x := 0;\ x := 1 \parallel \mathtt{if}(x^{\mathsf{acq}})\{r := y\} \\ \parallel & y := 0;\ y := 1 \parallel \mathtt{if}(y^{\mathsf{acq}})\{s := x\} \end{array}$$

The resulting execution is disallowed due to the cycle.



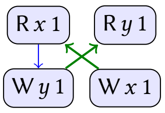
Another class of non-MCA behavior is characterized by the following litmus test.

$$\mathtt{if}(x)\{y := 0\};\ y := 1 \parallel \mathtt{if}(y)\{z := 0\};\ z := 1 \parallel \mathtt{if}(z)\{x := 0\};\ x := 1$$



This execution is disallowed due to the evident cycle.

Load buffering and thin air. The program ($y\,{:=}\,x \parallel s\,{:=}\,y;\, x\,{:=}\,1$) has top level executions that result in the final outcome $x = y = 1$, such as:

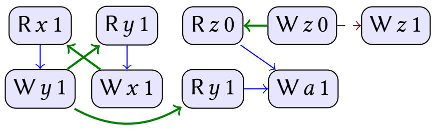


In §7 we provide machinery to prove that this outcome is impossible if there is order from read to write in both threads. This order can be achieved by replacing the second thread ($s\,{:=}\,y;\, x\,{:=}\,1$) with ($s\,{:=}\,y^{\mathsf{acq}};\, x\,{:=}\,1$) or ($\mathtt{if}(y)\{x\,{:=}\,1\}$) or ($x\,{:=}\,y$).

A more interesting example is the following variant of (2):

$$(y\,{:=}\,x \parallel \mathtt{if}(z)\{x\,{:=}\,1\}\,\mathtt{else}\,\{x\,{:=}\,y;\, a\,{:=}\,y\} \parallel z\,{:=}\,0;\, z\,{:=}\,1) \tag{7}$$

This program is allowed to write 1 to a under many speculative memory models [Jagadeesan et al. 2010; Kang et al. 2017; Manson et al. 2005], even though the read of 1 from y in the else branch of the second thread arises out of thin air. Lochbihler [2013] argues that such executions compromise type safety unless object allocation partitions memory by type. In our model, the attempted execution is:

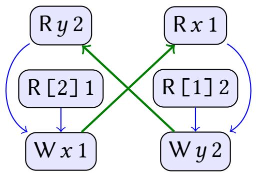


This is forbidden by the evident cycle.

Pointers. Our language allows address calculations. To model this, we allow locations to have the form $[n]$, where n is a natural number. Because we do not enforce order between reads, there is some danger that address calculations could allow thin air behavior. To see how our model addresses this, assume that we have the following locations and initial values:

$$[0] = 0 \qquad [1] = 2 \qquad [2] = 1 \qquad x = 0 \qquad y = 0$$

Consider the program ($x\,{:=}\,[y] \parallel y\,{:=}\,[x]$) with attempted execution:



Although there is no order enforced between the reads, the read-to-write order induced by the semantics is sufficient to prohibit this thin-air behavior.

The dependency calculation in this example is interesting. Desugaring, the first thread is $r\,{:=}\,y;\, s\,{:=}\,[r];\, x\,{:=}\,s$. In isolation, the write action is $(s = 1 \mid \mathsf{W}\,x\,1)$. Following the discussion of execution (3), $s\,{:=}\,[r]$ causes the precondition to become $[r] = 1$; subsequently prefixing with $(r = 2 \mid \mathsf{R}\,[2]\,1)$, this can be weakened to $1 = 1$ by the substitution of 1 for $[2]$, as long as there is order between the events. But the precondition is also constrained by the last clause of the Definition 2.1: if $d \le e$ then $\lambda_\Phi(e)$ implies $\lambda_\Phi(d)$. Thus the we arrive at the label $(r = 2 \mid \mathsf{W}\,x\,1)$. Subsequently, $r\,{:=}\,y$ transforms the precondition to $y = 2$ and prefixing $\mathsf{R}\,y\,2$ allows this to be weakened to $2 = 2$.

Blockers. Fulfillment is often defined in terms of the absence of *blocker*, rather than the presence of order to every conflicting event. In this definition, the last bullet of Definition 2.2 becomes:

- there is no $d < c < e$ such that c writes to x.

This definition is not preserved by parallel composition. To make the consequences clear, we include location declarations in our language, written $\mathtt{var}\,x;\,C$. Fulfillment with blockers violates *scope extrusion* [Milner 1999], in that we can find programs C and D such that the semantics $\mathtt{var}\,x;\,(C \parallel D)$ is different from the semantics of $(\mathtt{var}\,x;\,C) \parallel D$, even if D does not mention x. To see this, consider the program:

$$r := a^{\mathsf{acq}};\ x := 1;\ b :=^{\mathsf{rel}} 1 \ \parallel\ r := c^{\mathsf{acq}};\ x := 2;\ d :=^{\mathsf{rel}} 1 \ \parallel\ r := e^{\mathsf{acq}};\ s := x;\ f :=^{\mathsf{rel}} 1$$

$$\begin{tikzcd}[column sep=small]
\mathsf{R}^{\mathsf{acq}}\,a\,1 \arrow[r] & \mathsf{W}\,x\,1 \arrow[r] \arrow[rrrrrr, bend left] & \mathsf{W}^{\mathsf{rel}}\,b\,1 & \mathsf{R}^{\mathsf{acq}}\,c\,1 \arrow[r] & \mathsf{W}\,x\,2 \arrow[r] & \mathsf{W}^{\mathsf{rel}}\,d\,1 & \mathsf{R}^{\mathsf{acq}}\,e\,1 \arrow[r] & \mathsf{R}\,x\,1 \arrow[r] & \mathsf{W}^{\mathsf{rel}}\,f\,1
\end{tikzcd}$$

In this execution, the read is fulfilled by the first thread; the second thread is not blocking. However, when placed in parallel with $(c := b;\ d := e)$, the second thread may become a blocker:

$$\begin{tikzcd}[column sep=small]
\mathsf{R}^{\mathsf{acq}}\,a\,1 \arrow[r] & \mathsf{W}\,x\,1 \arrow[r] \arrow[rrrrrr, bend left] & \mathsf{W}^{\mathsf{rel}}\,b\,1 \arrow[d] & \mathsf{R}^{\mathsf{acq}}\,c\,1 \arrow[r] & \mathsf{W}\,x\,2 \arrow[r] & \mathsf{W}^{\mathsf{rel}}\,d\,1 \arrow[d] & \mathsf{R}^{\mathsf{acq}}\,e\,1 \arrow[r] & \mathsf{R}\,x\,1 \arrow[r] & \mathsf{W}^{\mathsf{rel}}\,f\,1 \\
& & \mathsf{R}\,b\,1 \arrow[r] & \mathsf{W}\,c\,1 \arrow[u] & & \mathsf{R}\,d\,1 \arrow[r] & \mathsf{W}\,e\,1 \arrow[u] & &
\end{tikzcd}$$

Our definition of fulfillment requires that the read be ordered with respect to both writes:

$$\begin{tikzcd}[column sep=small]
\mathsf{R}^{\mathsf{acq}}\,a\,1 \arrow[r] & \mathsf{W}\,x\,1 \arrow[r] \arrow[rrrrrr, bend left] & \mathsf{W}^{\mathsf{rel}}\,b\,1 & \mathsf{R}^{\mathsf{acq}}\,c\,1 \arrow[r] & \mathsf{W}\,x\,2 \arrow[r] & \mathsf{W}^{\mathsf{rel}}\,d\,1 & \mathsf{R}^{\mathsf{acq}}\,e\,1 \arrow[r] & \mathsf{R}\,x\,1 \arrow[r] \arrow[lll, bend left, dashed] & \mathsf{W}^{\mathsf{rel}}\,f\,1
\end{tikzcd}$$

This makes augmentation in the reverse order impossible.

Read from unexecuted branch (RFUB). Boehm [2018] analyzes programs in which register state from an unexecuted branch can effect the outcome of an execution. Boehm's RFUB1 example can be written in our language as:

$$y := x \ \parallel\ r := y;\ \mathtt{if}(r \neq 1)\{z := 1;\ r := 1;\ x := r\}\,\mathtt{else}\,\{x := r\} \tag{8}$$

Boehm's concern arises from the similarity of this program with the OOTA litmus test that replaces the second thread with $r := y;\ x := r$. Should an unexecuted conditional be allowed to change the outcome of the program? In our semantics the answer is "yes". First note that the program writes 1 to x in both branches of the conditional. Further, the writes to z and x in the then-branch of the conditional are independent. Therefore, it is sensible for a compiler to hoist the write to x out of the conditional.

To analyze the example formally, we use combinators on pomsets that are defined in the next section. These are prefixing ($\Rightarrow$) and composition ($\parallel$). We also include events with unsatisfiable conditions in our drawings; we indicate unsatisfiability by crossing out the event. The execution in question is:

$$\begin{tikzcd} \mathsf{R}\,x\,1 \arrow[d] \\ \mathsf{W}\,y\,1 \end{tikzcd} \ \Big\|\ \mathsf{R}\,y\,1 \Rightarrow \left(\begin{array}{c} \boxed{r \neq 1 \mid \mathsf{W}\,z\,1} \\[1em] \boxed{r \neq 1 \mid \mathsf{W}\,x\,1} \end{array} \ \Big\|\ \boxed{r = 1 \mid \mathsf{W}\,x\,1} \right)$$

With an internal read of y, we have:

$$\begin{tikzcd} \mathsf{R}\,x\,1 \arrow[d] \\ \mathsf{W}\,y\,1 \end{tikzcd} \ \Big\|\ \begin{array}{c} \boxed{y \neq 1 \mid \mathsf{W}\,z\,1} \\[1em] \boxed{\mathsf{W}\,x\,1} \end{array}$$

but the precondition $y \neq 1$ cannot be satisfied locally. With an external read of $(\mathsf{R}\,y\,1)$, we can discharge the precondition, but in this case, the predicate becomes $1 \neq 1$.

$$\begin{tikzcd}
\mathsf{R}\,x\,1 \arrow[d] & \parallel & \mathsf{R}\,y\,1 \arrow[r] & \cancel{1 \neq 1 \mid \mathsf{W}\,z\,1} \\
\mathsf{W}\,y\,1 & & & \mathsf{W}\,x\,1
\end{tikzcd}$$

Causality test cases. Pugh [2004] developed a set of twenty causality test cases in the process of revising the Java Memory Model (JMM) [Manson et al. 2005]. Using hand calculation, we have confirmed that our model gives the desired result for all twenty cases, unrolling loops as necessary. Our model also gives the desired results for all of the examples in Batty et al. [2015, §4] and Ševčík [2008, §5.3]. Our model agrees with the JMM on the "surprising and controversial behaviors" of Manson et al. [2005, §8]. §D develops some of these examples.

3 THE SEMANTICS OF A SIMPLE CONCURRENT LANGUAGE

In the previous section, we described our semantics informally. In this section we firm things up. In §3.1, we describe data models. In §3.2 we formalize some definitions for sets of pomsets. In both cases, the details are tedious and mundane. The exciting bits are in §3.3, where we describe combinators for sets of pomsets, and §3.4, where we use the combinators to define the semantics of a simple concurrent language.

3.1 Data models

A *data model* consists of:

- a set of *values* $\mathcal{V}$, ranged over by v, w, u and ℓ,
- a set of *registers* $\mathcal{R}$, ranged over by r and s,
- a set of *expressions* $\mathcal{E}$, ranged over by M, N, L and K,
- a set of *memory locations* $\mathcal{X}$, ranged over by x and y,
- a set of *logical formulae* Φ, ranged over by ϕ and ψ, and
- a set of *actions* $\mathcal{A}$, ranged over by a and b.

Let σ range over substitutions of the form $\phi[x/r]$ or $\phi[N/x]$.

We require that data models satisfy the following:

- the sets of values, registers and memory locations are disjoint,
- values include at least the constants 0 and 1,
- expressions include at least registers and values,
- memory locations have the form $[v]$,
- expressions do *not* include memory locations or the operator $[M]$,
- formulae include at least true, false, and equalities of the form $(M = v)$,
- formulae are closed under negation, conjunction, disjunction, and substitution[2], and
- there is a relation $\vDash$ between formulae.

We say that ϕ is *independent of* x whenever $\phi \vDash \phi[v/x] \vDash \phi$ for every v, and that ϕ is *dependent on* x otherwise. We say that ϕ is *location independent* if it is independent of every location.

We say that ϕ *implies* ψ whenever $\phi \vDash \psi$, that ϕ is a *tautology* whenever true $\vDash \phi$, that ϕ is *unsatisfiable* whenever $\phi \vDash$ false.

For the actions of a data model, we require that

- there are partial functions Reads and Writes : $\mathcal{A} \to (\mathcal{X} \times \mathcal{V}_\perp)$,

[2]Since formulae are closed under substitutions of the form $\phi[x/r]$, they must include equalities of the form $(\mathbb{M} = v)$ and $([\mathbb{M}] = x)$, where $\mathbb{M}$ is an *extended expression* that includes memory locations. We elide the details. By composition of the closure conditions, formulae must also be closed under that substitutions of the form $\phi[M/r] = \phi[x/r][M/x]$.

- there are sets Release and Acquire $\subseteq \mathcal{A}$, and
- there is a function internalize : $(\mathcal{X} \times \mathcal{A}) \rightarrow \mathcal{A}$ that satisfies the restrictions given below.

We say that *a reads v from x* whenever Reads$(a) = (x, v)$, and that *a writes v to x* whenever Writes$(a) = (x, v)$. We say that *a reads from x* whenever Reads$(a) = (x, v)$, and that *a writes to x* whenever Writes$(a) = (x, v)$, for some v (possibly $\perp$).

Actions that read or write values are *external*, actions that read or write $\perp$ are *internal*.

We say that a is an *acquire* if $a \in$ Acquire, and that a is a *release* if $a \in$ Release.

We require that internalize satisfy the following:

- internalize(a) reads $\perp$ from x exactly when a reads from x,
- internalize(a) writes $\perp$ from x exactly when a writes from x,
- internalize(a) is an acquire exactly when a is an acquire, and
- internalize(a) is a release exactly when a is a release.

As noted in §2, our example language includes acquiring read ($\mathsf{R}^{\mathsf{acq}}\, x\, v$), relaxed read ($\mathsf{R}\, x\, v$), releasing write ($\mathsf{W}^{\mathsf{rel}}\, x\, v$), and relaxed write ($\mathsf{W}\, x\, v$). For each external action, we also define a corresponding internal action ($\mathsf{R}^{\mathsf{acq}}\, x\, \perp$), ($\mathsf{R}\, x\, \perp$), ($\mathsf{W}^{\mathsf{rel}}\, x\, \perp$), and ($\mathsf{W}\, x\, \perp$). In pictures, we draw internal actions grayed out, rather than using $\perp$.

We also include acquire-release fences of the form ($\mathsf{F}^{\mathsf{acqrel}}$).

3.2 The semantic domain

Recall Definition 2.1 of memory model pomsets.

We lift terminology from logical formulae and actions to events. For example, we say that e is unsatisfiable when $\lambda_\Phi(e)$ is unsatisfiable, and that e is an acquire when $\lambda_{\mathcal{A}}(e)$ is an acquire.

We give the semantics of programs as sets of pomsets. Each pomset $P \in [\![C]\!]$ will represent a single execution of C.

We expect the sets of pomsets given by the semantics to be closed with respect to *isomorphism*, *augmentation* and *implication*.

Definition 3.1. P' is an *isomorphism* of P if there is a bijection $f : E \rightarrow E'$ such that $\lambda(e) = \lambda'(f(e))$, and $e \leq d$ iff $f(e) \leq' f(d)$.

P' is an *augmentation* of P if $E' = E$, $\lambda' = \lambda$, and $\leq' \supseteq \leq$.

P' *implies* P if $E' = E$, $\leq' = \leq$, $\lambda'_{\mathcal{A}} = \lambda_{\mathcal{A}}$, and $\lambda'_\Phi(e)$ implies $\lambda_\Phi(e)$ for all $e \in E$.

Each $P \in [\![C]\!]$ as a *completed* execution. So, we do not expect $[\![C]\!]$ to be prefixed closed. However, implication closure in a memory-model pomset does give something similar: any event e can be given an unsatisfiable precondition, which means that every event ordered after e must also be unsatisfiable, as per Definition 2.1. Since unsatisfiable events are ignored by our model, this provides a kind of prefix closure.

3.3 Combinators

We give the semantics using combinators over sets of pomsets, defined below. Using $\mathcal{P}$ to range over sets of pomsets, these are:

- *substitution* $\mathcal{P}\sigma$, which applies the substitution to every precondition,
- *restriction* $\nu x\,.\,\mathcal{P}$, which internalizes x for pomsets that are x-closed,
- *composition* $\mathcal{P}^1 \parallel \mathcal{P}^2$, which unions pomsets, allowing events to be merged, and
- *prefixing* $(\phi \mid a) \Rightarrow \mathcal{P}$, which adds an event c with label $(\phi \mid a)$ to pomsets in $\mathcal{P}$, ordering c before any e whose predicate depends on the value read by a. We elide ϕ when it is a tautology.

These operations are similar to those from models of concurrency such as [Brookes et al. 1984].

We also define two filtering operations:

- *guarding* $\phi \triangleright \mathcal{P}$, which keeps pomsets where all preconditions imply ϕ, and
- *independency filtering* $\mathcal{X} \triangleright \mathcal{P}$, which keeps pomsets where all preconditions are location independent.

The definitions of substitution, restriction and the filtering operations are straightforward[3]:

Definition 3.2. Let $\mathcal{P}\sigma$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that: $E' = E$, $\leq' = \leq$, and $\lambda'(e) = (\psi\sigma \mid a)$ when $\lambda(e) = (\psi \mid a)$.

Let $(\nu x \,.\, \mathcal{P})$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that P is x-closed and: $E' = E$, $\leq' = \leq$, and $\lambda'(e) = (\psi \mid \mathsf{internalize}(a))$ when $\lambda(e) = (\psi \mid a)$.

Let $(\phi \triangleright \mathcal{P})$ be the subset of $\mathcal{P}$ such that $P \in \mathcal{P}$ whenever ϕ implies $\lambda_\Phi(e)$, for every $e \in E$.

Let $(\mathcal{X} \triangleright \mathcal{P})$ be the subset of $\mathcal{P}$ such that $P \in \mathcal{P}$ whenever $\lambda_\Phi(e)$ is location independent, for every $e \in E$.

Definition 3.3. Let $P' \in (\mathcal{P}^1 \parallel \mathcal{P}^2)$ whenever there are $P^1 \in \mathcal{P}^1$ and $P^2 \in \mathcal{P}^2$ such that:

- $E' = E^1 \cup E^2$,
- $\leq' \supseteq \leq^1 \cup \leq^2$,
- either
  - $\lambda'_{\mathcal{A}}(e) = \lambda^1_{\mathcal{A}}(e) = \lambda^2_{\mathcal{A}}(e)$ and $\lambda'_\Phi(e)$ implies $\lambda^1_\Phi(e) \vee \lambda^2_\Phi(e)$,
  - $\lambda'_{\mathcal{A}}(e) = \lambda^1_{\mathcal{A}}(e)$, $\;e \notin E^2\;$ and $\lambda'_\Phi(e)$ implies $\lambda^1_\Phi(e)$, or
  - $\lambda'_{\mathcal{A}}(e) = \lambda^2_{\mathcal{A}}(e)$, $\;e \notin E^1\;$ and $\lambda'_\Phi(e)$ implies $\lambda^2_\Phi(e)$.

Composition is used in giving the semantics for conditionals and concurrency. $\mathcal{P}^1 \parallel \mathcal{P}^2$ contains the union of pomsets from $\mathcal{P}^1$ and $\mathcal{P}^2$, allowing overlap as long as they agree on actions. For example, if $\mathcal{P}^1$ and $\mathcal{P}^2$ contain:

$$\begin{tikzcd} \boxed{\phi \mid a} \arrow[r] & \boxed{\psi^1 \mid b} & & \boxed{\psi^2 \mid b} \arrow[r, dashed] & \boxed{\chi \mid c} \end{tikzcd}$$

then $\mathcal{P}^1 \parallel \mathcal{P}^2$ contains:

$$\begin{tikzcd} \boxed{\phi \mid a} \arrow[r] & \boxed{\psi^1 \vee \psi^2 \mid b} \arrow[r, dashed] & \boxed{\chi \mid c} \end{tikzcd}$$

Definition 3.4. Let $(\phi \mid a) \Rightarrow \mathcal{P}$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that:

(1) $E' = E \cup \{c\}$,
(2) $\leq' \supseteq \leq$,
(3) $\lambda'_{\mathcal{A}}(c) = a$, $\lambda'_{\mathcal{A}}(e) = \lambda_{\mathcal{A}}(e)$, and $\lambda'_\Phi(c)$ implies ϕ,
(4) if a is not a read then $\lambda'_\Phi(e)$ implies $\lambda_\Phi(e)$,
(5) if a internally reads from x then both
  (5a) $\lambda'_\Phi(e)$ implies $\lambda_\Phi(e)$, and
  (5b) if e is a write then either $c <' e$ or $\lambda_\Phi(e)$ is independent of x,
(6) if a externally reads v from x then both
  (6a) $\lambda'_\Phi(e)$ implies $\lambda_\Phi(e)[v/x]$, and
  (6b) if e is a write then either $c <' e$ or $\lambda'_\Phi(e)$ implies $\lambda_\Phi(e)$,
(7) if a and $\lambda_{\mathcal{A}}(e)$ are external actions in conflict, then $c \leq' e$,
(8) if a is an acquire or $\lambda_{\mathcal{A}}(e)$ is a release then $c <' e$, and
(9) if a is an acquire then $\lambda_\Phi(e)$ is location independent.

[3]We have chosen the definition of restriction for its simplicity. It is worth noting, however, that our definition does not support renaming of variables. In particular $(\nu x \,.\, \mathcal{P} \parallel (\nu x \,.\, \mathcal{Q}))$ is generally not the same as $(\nu x \,.\, \mathcal{P} \parallel (\nu y \,.\, \mathcal{Q}[y/x]))$. To support renaming, $(\nu x \,.\, \mathcal{P})$ would need to either remove or relabel events that mention x.

$(\phi \mid a) \Rightarrow \mathcal{P}$ adds a new event c with label $(\phi \mid a)$ to each pomset in $\mathcal{P}$. As in the definition of parallel composition, the definition allows the new event to overlap with events in $\mathcal{P}$ as long as they agree on the action. Overlapping of synchronization events is disallowed by item 8.

If c writes to a location that is also written by some e in $\mathcal{P}$, item 7 introduces order between them: $c \le e$. This ensures that these writes cannot be given the reverse order in an augmentation.

If c reads from a location that occurs in the predicate of e, then prefixing introduces order from c to e. whose predicate depends on x. For example, if $\mathcal{P}$ contains $(x > 0 \mid \mathsf{W}\, z\, 1)$ then $(\mathsf{R}\, x\, 1) \Rightarrow \mathcal{P}$ contains:

$$\begin{tikzcd} \boxed{\mathsf{R}\,x\,1} \arrow[r] & \boxed{1>0 \mid \mathsf{W}\,z\,1} & \text{and} & \boxed{\mathsf{R}\,x\,1} & \boxed{x>0 \mid \mathsf{W}\,z\,1} \end{tikzcd}$$

In order to weaken the predicate on $(\mathsf{W}\, z\, 1)$, item 6b requires that we include the order from $(\mathsf{R}\, x\, 1)$ to $(\mathsf{W}\, z\, 1)$, but not if the predicate remains $x > 0$ (which trivially implies $1 > 0$).

Item 8 ensures that events are ordered before a release and after an acquire.

Item 9 filters the executions of $\mathcal{P}$, ensuring that thread-local reads do not cross acquire actions. This prevents bad executions like the following, which violate DRF-SC.

$$\mathtt{x\!:=\!1;\ a^{rel}\!:=\!1;\ if(b^{acq})\{r\!:=\!x;\ y\!:=\!r\}} \parallel \mathtt{if(a^{acq})\{x\!:=\!2;\ b^{rel}\!:=\!1\}}$$

$$\begin{tikzcd} \boxed{\mathsf{W}\,x\,1} \arrow[r] & \boxed{\mathsf{W}^{\mathsf{rel}}\,a\,1} \arrow[d] & & \boxed{\mathsf{R}^{\mathsf{acq}}\,b\,1} \arrow[r] & \boxed{\mathsf{R}\,x} \arrow[r] & \boxed{\mathsf{W}\,y\,1} \\ & \boxed{\mathsf{R}^{\mathsf{acq}}\,a\,1} \arrow[r] & \boxed{\mathsf{W}\,x\,2} \arrow[r] & \boxed{\mathsf{W}^{\mathsf{rel}}\,b\,1} \arrow[u] & & \end{tikzcd}$$

In item 9, we do not require that ψ' is independent of every y; were we to require this, the definition would not be augment closed.

The following lemma is immediate from the definitions.

LEMMA 3.5. *All combinators are monotone with respect to subset order.*

3.4 Semantics of programs

We consider a simple shared-memory concurrent language, with statements defined as follows.

$$\begin{aligned} C, D \ ::= &\ \mathtt{skip} \mid C \parallel D \mid \mathtt{var}\, x;\, C \mid \mathtt{fence};\, C \mid \mathtt{if}(M)\{C\}\,\mathtt{else}\,\{D\} \\ &\mid r\!:=\!M;\, C \mid r\!:=\![L]^{\mathsf{acq}};\, C \mid r\!:=\![L];\, C \mid [L]^{\mathsf{rel}}\!:=\!M;\, C \mid [L]\!:=\!M;\, C \end{aligned}$$

We use common syntax sugar, such as *extended expressions*, which include memory locations. For example, if the extended expression $\mathbb{M}$ includes a single occurrence of x, then $y\!:=\!\mathbb{M};\, C$ is shorthand for $r\!:=\!x;\ y\!:=\!\mathbb{M}[r/x];\, C$. Each occurrence of x in an extended expression corresponds to an independent read. We also write $\mathtt{if}(M)\{C^1\}\,\mathtt{else}\,\{C^2\};\, D$ as shorthand for $\mathtt{if}(M)\{C^1;\, D\}$ $\mathtt{else}\,\{C^2;\, D\}$ and $\mathtt{var}\, x\!:=\!M;\, C$ as shorthand for $\mathtt{var}\, x;\ x\!:=\!M;\, C$.

The semantics of programs is as follows[4]:

$$\begin{aligned} [\![\mathtt{skip}]\!] &= \{\emptyset\} \\ [\![C \parallel D]\!] &= \mathcal{X} \triangleright [\![C]\!] \parallel \mathcal{X} \triangleright [\![D]\!] \\ [\![\mathtt{var}\, x;\, C]\!] &= \nu x\, .\, [\![C]\!] \\ [\![\mathtt{fence};\, C]\!] &= (\mathsf{F}^{\mathsf{acqrel}}) \Rightarrow [\![C]\!] \\ [\![\mathtt{if}(M)\{C\}\,\mathtt{else}\,\{D\}]\!] &= \big((M \neq 0) \triangleright [\![C]\!]\big) \parallel \big((M = 0) \triangleright [\![D]\!]\big) \\ [\![r\!:=\!M;\, C]\!] &= [\![C]\!][M/r] \\ [\![r\!:=\![L]^{\mathsf{acq}};\, C]\!] &= \textstyle\bigcup_{x=[\ell]} \bigcup_v\, (L = \ell \mid \mathsf{R}^{\mathsf{acq}}\, x\, v) \Rightarrow [\![C]\!][x/r] \\ [\![r\!:=\![L];\, C]\!] &= \textstyle\bigcup_{x=[\ell]} \bigcup_v\, (L = \ell \mid \mathsf{R}\, x\, v) \Rightarrow [\![C]\!][x/r] \end{aligned}$$

[4]Fork parallelism is easily expressible in this framework: $[\![C \mathbin{\rceil\!\lceil} D]\!] = [\![C]\!] \parallel \mathcal{X} \triangleright [\![D]\!]$.

$$\begin{aligned}
&\cup \textstyle\bigcup_{x=[\ell]} (L = \ell \mid \mathsf{R}\,x\,\bot) \Rightarrow [\![C]\!][x/r] \\
[\![[L]^{\mathsf{rel}} := M;\, C]\!] &= \textstyle\bigcup_{x=[\ell]} \bigcup_{v} (L = \ell \wedge M = v \mid \mathsf{W}^{\mathsf{rel}}\,x\,v) \Rightarrow [\![C]\!][M/x] \\
[\![[L] := M;\, C]\!] &= \textstyle\bigcup_{x=[\ell]} \bigcup_{v} (L = \ell \wedge M = v \mid \mathsf{W}\,x\,v) \Rightarrow [\![C]\!][M/x] \\
&\cup \textstyle\bigcup_{x=[\ell]} (L = \ell \mid \mathsf{W}\,x\,\bot) \Rightarrow [\![C]\!][M/x]
\end{aligned}$$

The semantics of relaxed reads is the union of two sets. The first set adds a read action to each pomset in $[\![C]\!]$; the second adds an internal action. Whereas read actions have values that are important in the semantics, the values of internal actions are ignored. Internal reads are used in the definition of data races in §4. Acquiring reads are never internal. The rule for relaxed writes is similar.

The write rule uses the substitution $[M/x]$ with precondition $M = v$, rather than using $[v/x]$ directly. To see the need for this, consider $[\![\mathtt{if}(s = r)\{z := 1\}]\!]$, which includes $(s = r \mid \mathsf{W}\,z\,1)$. Therefore $[\![s := x;\ \mathtt{if}(s = r)\{z := 1\}]\!]$ includes $(x = r \mid \mathsf{W}\,z\,1)$ and $[\![x := r;\ s := x;\ \mathtt{if}(s = r)\{z := 1\}]\!]$ includes $(r = r \mid \mathsf{W}\,z\,1)$ which is independent of r. If we took the semantics of write to use $[v/x]$, then we would end up with pomsets of the form $(v = r \mid \mathsf{W}\,z\,1)$ which depend on r.

The role of internal writes in the semantics is to facilitate dead store. Consider the semantics of $y := 0;\ y := 1$. It includes not only:

$$\begin{tikzcd} \boxed{\mathsf{W}\,y\,0} \arrow[r, dashed] & \boxed{\mathsf{W}\,y\,1} \end{tikzcd}$$

but also:

$$\begin{tikzcd} \boxed{\mathsf{W}\,y} \arrow[r, dashed] & \boxed{\mathsf{W}\,y\,1} \end{tikzcd}$$

The latter pomset is effectively the semantics of $y := 1$, thus validating the dead store transformation that eliminates the first write.

Prefixing does not necessarily induce a dependency, even for read actions where the read is used. To see that this is desirable, consider the semantics of $y := 0;\ r := y;\ \mathtt{if}(r \leq 1)\{x := 2\}$. To begin, note that $[\![\mathtt{if}(r \leq 1)\{x := 2\}]\!]$ includes $(r \leq 1 \mid \mathsf{W}\,x\,2)$ which depends on r. Then $[\![r := y;\ \mathtt{if}(r \leq 1)\{x := 2\}]\!]$ includes

$$\begin{tikzcd} \boxed{\mathsf{R}\,y} \arrow[r] & \boxed{y \leq 1 \mid \mathsf{W}\,x\,2} \end{tikzcd}$$

Prefixing with a write to y, $[\![y := 0;\ r := y;\ \mathtt{if}(r \leq 1)\{x := 2\}]\!]$ discharges the precondition of the write to x, yielding

$$\begin{tikzcd} \boxed{\mathsf{W}\,y\,0} & \boxed{\mathsf{R}\,y} \arrow[r] & \boxed{0 \leq 1 \mid \mathsf{W}\,x\,2} \end{tikzcd}$$

which simplifies to:

$$\begin{tikzcd} \boxed{\mathsf{W}\,y\,0} & \boxed{\mathsf{R}\,y} \arrow[r] & \boxed{\mathsf{W}\,x\,2} \end{tikzcd}$$

Here the thread-local value of y discharges the predicate.

4 DATA RACE FREE BEHAVIORS ARE SEQUENTIALLY CONSISTENT

In this section, we prove the SC-DRF theorem, which states that any program that lacks data races under the SC semantics must only have executions that are compatible with SC executions. We present the result for programs of the form $\vec{x} := \vec{0};\ \mathtt{fence};\ C$, where C is restriction-free. Thus, all memory locations are initialized to 0, initialization happens-before the execution of any command, and internal actions only arise from (intra-thread) implicit reads.

Define the relation eco so that $(e, d) \in \mathsf{eco}$ if $e \leq d$ and e and d conflict.

The program $x := 1 \parallel x := 2$ is considered to have an SC data race, but $x := 1;\ x := 2$ does not. In our semantics the only difference between these is that $x := 1;\ x := 2$ enforces weak order between the writes. Note also that the $[\![x := 1;\ a := y]\!] = [\![a := y;\ x := 1]\!]$, yet these two must be distinguished in SC, as per the load-buffering and store-buffering litmus tests.

In order to define SC executions and SC data races, it is necessary to augment our semantics to record program order. We extend the definitions in §3.4 with $\mathsf{po} \subseteq E \times E$, defined as follows:

- $\mathsf{po}' = \mathsf{po}$ when $\mathcal{P}' = \mathcal{P}\sigma$ or $\mathcal{P}' = \phi \rhd \mathcal{P}$
- $\mathsf{po}' = \mathsf{po}|_{E'}$ when $\mathcal{P}' = \nu x \,.\, \mathcal{P}$
- $\mathsf{po}' = \mathsf{po}^1 \cup \mathsf{po}^2$ when $\mathcal{P}' = \mathcal{P}^1 \parallel \mathcal{P}^2$
- $\mathsf{po}' = \mathsf{po} \cup \{(c, e) \mid e \in E\}$ when $\mathcal{P}' = a \Rightarrow \mathcal{P}$ and $E' = E \cup \{c\}$

Define the relation rf so that $(e, d) \in \mathsf{rf}$ if e writes x, d reads x, and for any c that writes x either $c \le e$ or $d \le c$. Let [Acquire] be the identity relation on acquire events, and likewise [Release] on release events. Now define sw and hb[5].

$$\mathsf{sw} = [\text{Release}]; (\mathsf{rf} \setminus \mathsf{po}); [\text{Acquire}]$$
$$\mathsf{hb} = (\mathsf{po} \cup \mathsf{sw})^+$$

Note that our semantics guarantees that $\mathsf{sw} \subseteq <$.

A pomset has a *data race* if there are events e and d such that

- e and d are unordered by hb,
- $\lambda_\Phi(e)$ and $\lambda_\Phi(d)$ are tautologies, and
- $\lambda_\mathcal{A}(e)$ and $\lambda_\mathcal{A}(d)$ conflict.

Definition 4.1. Let $[\![C]\!]_{\mathrm{SC}}$ be the subset of $[\![C]\!]$ such that $P \in [\![C]\!]_{\mathrm{SC}}$ whenever P is a top-level pomset and $< \cup\, \mathsf{po}$ is acyclic.

We argue that this definition is sufficient to capture sequential consistency: Any total order that linearizes the acyclic relation is consistent with strong order ($<$) and the program order (po). Since $\le$ contains eco, only the last write to a location is read in such a total order.

We only consider *generators*, which are top-level pomsets that are minimal with respect to augmentation and implication. Since we are considering finite programs without loops, the pomsets in the semantics of threads are finite. Thus, there are no infinite descending chains of augmentations.

We prove the following theorem in §B.

Theorem 4.2. *Let P be a generator for C.*

- *If P does not have a data race, $P \in [\![C]\!]_{\mathrm{SC}}$.*
- *If P has a data race, then there exists $P' \in [\![C]\!]_{\mathrm{SC}}$ that has a data race.*

A key step of this proof is an analysis of the closure properties of the semantics. In order to perform this fine grained analysis of dependency, we describe a variant of the semantics using modal pomsets defined below.

Definition 4.3. A *modal pomset* is a tuple $(E, \trianglelefteq, \le, \lambda)$, such that

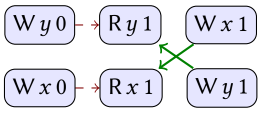
- $(E, \le, \lambda)$ is a pomset, and
- $\trianglelefteq\, \subseteq\, \le$ is a partial order.

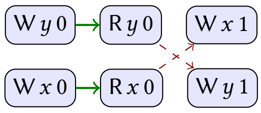
These are the pomset equivalent of the *modal transition systems* of Larsen and Thomsen [1988], Huth et al. [2001] call $\lhd$ a "must transition" and $<$ a "may transition". In anticipation, we have used the terms "strong order" and "weak order" respectively, drawing $\lhd$ as a solid arrow "$\rightarrow$" and $<$ dashed "$\dashrightarrow$' in the pictures following Lamport's [1986] notation. The intuitive temporal meaning of $e \lhd d$ is that e *must* strictly precede d, whereas $e < d$ is intended to connote that e may precede d.

[5]For simplicity, the definition of sw does not include release sequences or fences. For example, we consider $(\mathsf{W}^{\mathsf{rel}}\, x\, 1) \Rightarrow (\mathsf{W}\, x\, 2) \parallel (\mathsf{R}^{\mathsf{acq}}\, x\, 2)$ to be racy, whereas this pomset is race-free using release sequences. If we include release sequences and fences, then hb relates more events and thus there are fewer races. Our results hold under either definition.

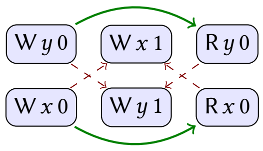
The semantics of programs in the modal model proceeds as before, mutatis mutandis, with details described in the appendix. This finer analysis of necessary and possible dependency allows us to establish the existence of pomsets in the semantics as we search for sequential witnesses to data races. We provide some illustrative examples below.

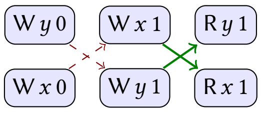
For the program $(y := 0;\ r := y;\ x := 1) \parallel (x := 0;\ s := x;\ y := 1)$:

From 

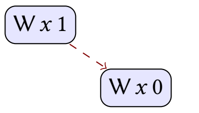
infer 

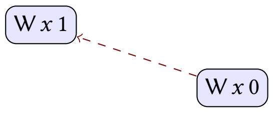
For the program $(y := 0;\ x := 1;\ r := y) \parallel (x := 0;\ y := 1;\ s := x)$:

From 

infer 

For the program $(x := 1) \parallel (x := 0)$:

From 

infer 

5 EFFICIENT IMPLEMENTATION ON ARMV8

In this section, we consider the fragment of our language without restriction. For simplicity, we allow release and acquire synchronization but ban fences. We assume that all memory locations are initialized to 0 and parallel-composition occurs only at top level. We take the set of memory locations to be finite. In other words, we assume that programs have the form

$$x_1 := 0;\ \cdots;\ x_m := 0;\ (C^1 \parallel \cdots \parallel C^n)$$

where $C^1, \ldots, C^n$ do not include composition, restriction or fence operations.

Our language can be translated to ARM using `ldr` for relaxed read, `ldar` for acquiring read, `str` for relaxed write, and `stlr` for releasing write. Relative to the ARM specification, we have removed loops and read-modify-write (RMW) operations, in addition to fences[6].

We show that any ARM-consistent execution graph for this sublanguage can be considered an execution of our semantics. Due to space limitations, we do not include a full description ARM consistency in the main text . Here we provide a birds eye view of the details, drawing on the intuitions gleaned from [Pulte et al. 2018]. Interested readers should see §C for further details.

An ARM execution graph G defines many relations, including program order (po), reads-from (rf), coherence (co) and several dependency orders. From these are derived:

- poloc, which is the subrelation of po that only relates actions on the same location,
- ob, which is required to be acyclic (EXTERNAL), and
- eco, with the requirement that poloc ∪ eco be acyclic (SC-PER-LOC).

Given an execution graph G, we say that e is an *internal read* if $e \in \text{codomain}(\text{po} \cap \text{rf})$.

The ob order is an acyclic global order on events, agreed upon by all threads, reflecting the progress of time in an ARMV8 execution. The cross thread component of the ordering is induced by the ordering on conflicting actions on the same location from different threads. The intra

[6]We leave out fences for simplicity. Following Podkopaev et al. [2019], our `fence` instruction can be translated to `dmb.sy`, since it has release-acquire semantics. Acquire fences map to `dmb.ld`, and release fences to `dmb.sy` — `dmb.st` does not provide order to prior reads.

thread component of the ordering is induced by barrier ordering and data ordering. Notably, these dependencies are determined syntactically. In particular, ob may not necessarily include the intra thread component of poloc ordering.

This motivates the translation of an ARMV8 execution into our setting. In our setting, the progress of time is given by $<$. We accommodate intra-thread reordering by internal read actions, thus excusing us from the obligation of placing them on the global $<$-timeline.

Formally, from G we construct a candidate pomset P as follows:

- $E = \mathsf{E}$,
- $\lambda_{\mathcal{A}}(e) = \tau \mathsf{lab}(e)$, if e is a relaxed internal read,
- $\lambda_{\mathcal{A}}(e) = \mathsf{lab}(e)$, if e is not a relaxed internal read,
- $\lambda_{\Phi}(e) = \mathsf{true}$,
- $\leq\ = (\mathsf{ob} \cup \mathsf{eco})^*$, where $*$ denotes reflexive and transitive closure.

Theorem 5.1. *If G is ARM consistent, the constructed candidate satisfies the requirements for a top-level memory model pomset.*

Any $<$-ordering imposed in our model is enforced by ARMV8, since our notion of semantic dependency is more permissive than ARMV8's syntactic dependency. So, the heart of the proof is showing the acyclicity of $(\mathsf{ob} \cup \mathsf{eco})^*$ for the events under consideration. Since the cross thread portion of eco (coe, fre, rfe) is included in ob, this result is really about the influence of $\mathsf{eco} \cap \mathsf{po}$. Our translation of ARMV8's rfi as silent internal actions removes them from order considerations. Consequently, we only have to consider the suborder of ob derived without ever using rfi for the following key property demonstrated in §C.

Lemma 5.2. *Let e, d be distinct events and d' $(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})$ d $((\xrightarrow{\mathsf{eco}} \cap \xrightarrow{\mathsf{po}}) \setminus \xrightarrow{\mathsf{rfi}})$ e $(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})$ e'. Then $d' \xrightarrow{\mathsf{ob}} e'$.*

Remark 5.3 (Proof for TSO). The proof for compilation into TSO is very similar. In particular the facts listed above hold for TSO, where ob is replaced by (the transitive closure of) the propagation relation defined for TSO [Alglave et al. 2014].

6 SINGLE THREADED OPTIMIZATIONS

As we have seen already, our model *invalidates* thread inlining. We argue that our model is fully flexible with respect to single threaded optimizations; concretely by validating several single threaded optimizations including reordering of independent statements; and abstractly, by proving a full abstraction theorem for single threads without synchronization and connecting to traditional Hoare logic for sequential programs.

Let $\mathsf{sat}(\mathcal{P})$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that: $E' = \{e \in E \mid \lambda_{\Phi}(e)$ is satisfiable and $\lambda_{\mathcal{A}}(e)$ is external$\}$, $\leq' = \leq|_{E'}$, and $\lambda' = \lambda|_{E'}$.

By Lemma 3.5, if $[\![C]\!] \supseteq [\![D]\!]$ then C can be transformed into D in any program context. Since unsatisfiable events are ignored in our architecture model, it is an immediate corollary that it is sufficient to show that $\mathsf{sat}[\![C]\!] \supseteq \mathsf{sat}[\![D]\!]$.

6.1 Validating single threaded optimizations

We follow the terminology and presentation of section 7.1 of Dolan et al. [2018], to maintain a clear comparison with models that enforce extra ordering.

Certain transformations involving adjacent operations on the same location are permissible.

Lemma 6.1. *Suppose $M = 0$ is a tautology and $L = K$ is a tautology. Then the following hold.*

$$\mathsf{sat}[\![\mathtt{if}(M)\{C\}\,\mathtt{else}\,\{D\}]\!] \supseteq \mathsf{sat}[\![D]\!] \qquad \text{(Dead Code)}$$

$$\mathsf{sat}[\![[L] := M;\ [K] := N;\ C]\!] \supseteq \mathsf{sat}[\![[K] := N;\ C]\!] \qquad \text{(Dead Store)}$$
$$[\![r := [L];\ s := [K];\ C]\!] \supseteq [\![r := [L];\ C[r/s]]\!] \qquad \text{(Redundant load)}$$
$$[\![[L] := M;\ s := [K];\ C]\!] \supseteq [\![[L] := M;\ C[M/s]]\!] \qquad \text{(Store forwarding)}$$

Proof. For dead code elimination, the result is immediate from the semantics.

For dead store, note that no event in C can depend on the first store, therefore it may be taken with precondition false.

For the others, the proof proceeds by observing that the semantics allows us to use an implicit action on the lefthand side to mimic the right hand side. For example, taking $[L] = [K] = x$, the argument for redundant load is as follows.

$$\begin{aligned}[\![r := x;\ s := x;\ C]\!] &\supseteq [\![s := x;\ C[x/r]]\!] \\ &\supseteq [\![C]\!][x/r][x/s] \cup \textstyle\bigcup_v\ (\mathsf{R}\,x\,v) \Rightarrow [\![C]\!][x/r][x/s] \\ &= [\![C]\!][r/s][x/r] \cup \textstyle\bigcup_v\ (\mathsf{R}\,x\,v) \Rightarrow [\![C]\!][r/s][x/r] \\ &= [\![r := x;\ C[r/s]]\!]\end{aligned}$$ □

Certain reorderings involving adjacent operations on the distinct locations are also permissible.

Lemma 6.2. *Suppose $L \neq K$ is a tautology. Then the following reorderings hold.*

$$[\![r := [L];\ s := [K];\ C]\!] = [\![s := [K];\ r := [L];\ C]\!] \qquad \textit{if } r \notin \mathsf{fn}(K) \textit{ and } s \notin \mathsf{fn}(L) \qquad \text{(R-R)}$$
$$[\![r := [L];\ [K] := N;\ C]\!] = [\![[K] := N;\ r := [L];\ C]\!] \qquad \textit{if } r \notin \mathsf{fn}(K) \cup \mathsf{fn}(N) \qquad \text{(R-W)}$$
$$[\![[L] := M;\ [K] := N;\ C]\!] = [\![[K] := N;\ [L] := M;\ C]\!] \qquad \text{(W-W)}$$

Proof. The proof of both of the above lemmas follows from the semantics of prefixing where the only enforced $<$ relationships come from conflict on x or release actions in C. □

Roach-motel reorderings that increase the scope of synchronization are valid.

Lemma 6.3. *Suppose $L \neq K$ is a tautology. Then the following reorderings hold.*

$$[\![r := [L];\ s := [K]^{\mathsf{acq}};\ C]\!] \supseteq [\![s := [K]^{\mathsf{acq}};\ r := [L];\ C]\!] \qquad \textit{if } r \notin \mathsf{fn}(K) \textit{ and } s \notin \mathsf{fn}(L) \qquad \text{(R-Acq)}$$
$$[\![[K] := N;\ r := [L]^{\mathsf{acq}};\ C]\!] \supseteq [\![r := [L]^{\mathsf{acq}};\ [K] := N;\ C]\!] \qquad \textit{if } r \notin \mathsf{fn}(K) \cup \mathsf{fn}(N) \qquad \text{(W-Acq)}$$
$$[\![[K]^{\mathsf{rel}} := N;\ r := [L];\ C]\!] \supseteq [\![r := [L];\ [K]^{\mathsf{rel}} := N;\ C]\!] \qquad \textit{if } r \notin \mathsf{fn}(K) \cup \mathsf{fn}(N) \qquad \text{(Rel-R)}$$
$$[\![[L]^{\mathsf{rel}} := M;\ [K] := N;\ C]\!] \supseteq [\![[K] := N;\ [L]^{\mathsf{rel}} := M;\ C]\!] \qquad \text{(Rel-W)}$$

Proof. The proof follows from noticing that the pomsets in the semantics of the right hand sides are augmentations of a pomset on the left hand side. □

Compiler optimizations. Reordering and peephole optimizations can be combined to describe common compiler optimizations. We illustrate using common subexpression elimination, following Dolan et al. [2018]: Consider the command $(r := x * 2;\ C;\ s := x * 2)$ where C is independent of r. Reordering yields $(C;\ r := x * 2;\ s := x * 2)$, followed by redundant load to yield $(C;\ r := x * 2;\ s := r)$.

Similarly, the treatment of loop-invariant code motion, dead-store elimination and constant propagation from Dolan et al. [2018] follow.

Since our model is more generous about permitted reorderings, we permit optimizations that they forbid. Consider: $(r := x;\ y := z;\ x := r)$. Reordering, permitted by us, but forbidden by them, yields $(r := x;\ x := r;\ y := z)$, followed by the valid elimination of redundant load $(r := x;\ x := r;\ y := z)$.

6.2 Full abstraction for synchronization free threads

Our semantics is complete for reasoning about full-thread optimizations of synchronization free programs.

In the rest of this section, we only consider commands C, D that are restriction-free, composition-free (ie. single threaded), and synchronization-free (ie. no acquire, release, fence).

In order to develop the proof, we first make the semantics insensitive to reads that are never used. Let $\mathrm{read}(\mathcal{P})$ be the smallest augmentation closed set containing $\mathcal{P}$ that also satisfies closure under the inclusion of useless reads: if $P \in \mathrm{read}(\mathcal{P})$, then $P' \in \mathrm{read}(\mathcal{P})$, where $E' = E \cup \{e\}$, $e \notin E$, $\leq' = \leq$, $\lambda'|_E = \lambda$, and $\lambda'(e) = \mathsf{R}\,x\,v$ for any x and v.

Let $\mathrm{readsat}(\mathcal{P}) = \mathrm{read}(\mathrm{sat}(\mathcal{P}))$.

This closure permits us to describe a normal form for the top-level pomsets that arise in single threaded and synchronization free code.

Definition 6.4. P is in normal form if:

- If $d < e$ and d is a write, then e is a write or read on the same variable.
- If $d < e$ and e is a read, then d is a write on the same variable.
- If e, e' have the same read action label, then there exists a write d on the same location such that $e \leq d \leq e'$.

In a normal form pomset, the successors of a write event (resp. the predecessors of a read) are related in the coherence order to the event. Any two events with the same read label are separated by a write in the eco order.

It suffices to consider normal forms when distinguishing single threaded, synchronization free code at the top level. The normal forms determine the full semantics by the closure properties of the semantics.

Lemma 6.5. *Every top-level pomset of* $\mathrm{readsat}[\![C]\!]$ *is a top-level pomset of* $\mathrm{readsat}[\![D]\!]$ *if and only if every top-level normal form pomset of* $\mathrm{readsat}[\![C]\!]$ *is a top-level normal form pomset of* $\mathrm{readsat}[\![D]\!]$.

Proof. The proof proceeds by induction on structure. The key case is prefixing. The only edges $<$-edges out of writes and into read actions enforced by prefixing are eco edges. Read prefixing permits the reuse of events to ensure that distinct read events not separated by eco have distinct read labels. □

We develop testers for top-level pomsets in normal form. We follow Plotkin and Pratt [1997], albeit in a concrete form appropriate to our setting.

Let P be a top-level pomset in normal form with events $e_1, \ldots, e_n$. For all i, we assume a new location b_i. Let v_{fr} be a fresh value that does not occur in P.

Let $\vec{d}$ be all the predecessors of e_i in $<$. Let $\vec{c}$ be the corresponding sub vector of b_i's.

- If e_i has label $\mathsf{R}\,x\,v$, we define a program $\mathtt{test}_i^P$ as follows:

$$\mathtt{test}_i^P = (\mathtt{if}(\vec{c} = \vec{1})\{x := v;\ \mathtt{fence};\ b_i := 1\})$$

- If e_i has label $\mathsf{W}\,x\,v$, we define a program $\mathtt{test}_i^P$ as follows:

$$\mathtt{test}_i^P = (\mathtt{if}(\vec{c} = \vec{1})\{\mathtt{if}(x = v)\{x = v_{\mathsf{fr}};\ \mathtt{fence};\ b_i := 1\}\})$$

$\mathtt{test}_i^P$ is as expected, matching the reads (resp. writes) in P by writes (resp. reads). The fresh value is used to flush out prior writes and reset to see fresh writes.

Definition 6.6 (Tester for P)**.** The tester context for P, $\mathtt{test}^P[-]$ is

$$b := b_1 \wedge \cdots \wedge b_n \parallel \mathtt{test}_1^P \parallel \cdots \parallel \mathtt{test}_n^P \parallel [-]$$

The constraints of the normal form ensure that if the labels of any two events are the same, then one is a predecessor of the other.

Lemma 6.7. *$[\![\texttt{test}^P[C]]\!]$ has a pomset that sets b to 1 iff $P \in \mathsf{readsat}[\![C]\!]$.*

Proof. It suffices to prove that there is a pomset that sets b to 1 in $\mathsf{readsat}[\![b := b_1 \wedge \cdots \wedge b_n \parallel \texttt{test}^P_1 \parallel \cdots \parallel \texttt{test}^P_n]\!] \parallel P'$ iff P is an augmentation of P'.

We prove by induction on $<$ that $\texttt{test}^P_i[C]$ sets b_i to 1 iff the event e_i is enabled. Result follows. □

Theorem 6.8. *Let C_1 and C_2 be restriction-free, composition-free and synchronization free. Then all top-level pomsets of $\mathsf{readsat}[\![C_1]\!]$ are also pomsets of $\mathsf{readsat}[\![C_2]\!]$ if and only if for all parallel contexts D, all top-level pomsets of $\mathsf{readsat}[\![C_1 \parallel D]\!]$ are also pomsets of $\mathsf{readsat}[\![C_2 \parallel D]\!]$*

Proof. The forward implication follows from the compositionality of the semantics.

For the converse, pick a top level pomset in normal form in $P \in \mathsf{readsat}[\![C_1]\!] \setminus \mathsf{readsat}[\![C_2]\!]$. The required context is given by the tester context $\texttt{test}^P[-]$. □

The full abstraction theorem applies only to full threads. The reason for this limited statement is that our impoverished language does not permit parallel composition to be used freely as the continuation in arbitrary sequential contexts. A full abstraction theorem that applies also to commands can be achieved with these richer distinguishing contexts.

6.3 Relationship to Hoare logic

In order to provide useful information about sequential commands in sequential contexts, we establish a relationship with Hoare triples for a sequential language.

This invariant captured in the following theorem underlies the definitions in the semantics of sequential commands. In this theorem, we consider Hoare triples over formulas as per as per §3.1.

Theorem 6.9. *Let C be a command without either synchronization or parallelism.*

The following Hoare triple is valid,

$$\{\phi \wedge y_1 = v_1 \wedge \cdots \wedge y_n = v_n\}\ C\ \{x = w\}$$

if and only if $\mathsf{readsat}[\![C]\!]$ contains an execution that includes

$$\begin{tikzcd} \vec{\psi} \mid \mathsf{R}\,\vec{y}\,\vec{v} \arrow[dr] & & \vec{\chi} \mid \mathsf{W}\,x\,\vec{u} \arrow[dl, dashed] \\ & e{:}\ \phi \mid \mathsf{W}\,x\,w & \end{tikzcd}$$

where $\mathsf{R}\,\vec{y}\,\vec{v}$ are all the events that precede e and $\mathsf{W}\,x\,\vec{u}$ are all the writes on x.

Proof. The proof proceeds by structural induction on the command. □

We illustrate with a few examples. $\{\mathsf{true}\}\ x := 1;\ x := 2\ \{x = 2\}$ holds. In the semantics, this is mirrored by the execution:

$$\begin{tikzcd} \mathsf{W}\,x\,1 \arrow[r, dashed] & \mathsf{W}\,x\,2 \end{tikzcd}$$

The two pomsets corresponding to the triple $\{x = 1\}\ r := x;\ \texttt{if}(r = 1)\{y := 1\}\ \{y = 1\}$ illustrate the two ways to satisfy preconditions in the above theorem.

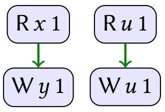
$$\text{Internal: } \boxed{x = 1 \mid \mathsf{W}\,y\,1} \qquad \text{External: } \begin{tikzcd} \mathsf{R}\,x\,1 \arrow[r] & \mathsf{W}\,y\,1 \end{tikzcd}$$

$\{\mathsf{true}\}\ r := x;\ \texttt{if}(r = 1)\{y := 1\}\texttt{else}\{y := 1\}\ \{y = 1\}$ holds. In the semantics, this is mirrored by the execution:

$$\boxed{\mathsf{W}\,y\,1} \quad \boxed{\mathsf{R}\,x\,v}$$

In contrast, in a semantics that forbids load buffering, where the best that one can prove is $\{x = v\}$ $r := x$; $\mathtt{if}(r = 1)\{y := 1\}\,\mathtt{else}\,\{y := 1\}$ $\{y = 1\}$.

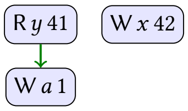
Let $C = r := x;\ \mathtt{if}(r = 1)\{y := 1\};\ s := u;\ \mathtt{if}(s = 1)\{v := 1\}$. Then: $\{x = 1\}\ C\ \{y = 1\}$ and $\{u = 1\}\ C\ \{v = 1\}$ hold. In the semantics, this is mirrored by the execution:



Consider a fragment from example (8).

$$D : r := y;\ \mathtt{if}(r \neq 42)\{a := 1;\ s := 42\};\ x := 42$$

Then, $\{\mathsf{true}\}\ D\ \{x = 42\}$ and $\{y \neq 42\}\ D\ \{a = 1\}$. Thus, the traditional sequential semantics *does not* attribute any cause for the write of 42 to x. In the semantics, this is mirrored by the execution:



7 INVARIANT REASONING IN TEMPORAL LOGIC

In this section, we develop sufficient logical infrastructure to prove that our semantics disallows thin air executions. In particular, we provide a proof that thin air execution (7) from §2 is impossible.

We adapt past linear temporal logic (PLTL) [Lichtenstein et al. 1985] to pomsets by dropping the previous instant operator and adopting strict versions of the temporal operators. The atoms of our logic are write and read events. Given a pomset P and event e, define:

$$\begin{array}{lcl}
P, e & \models & \mathsf{W}\,x\,v,\ \text{if } \lambda(e) = (\mathsf{true}, \mathsf{W}\,x\,v) \\
P, e & \models & \mathsf{R}\,x\,v,\ \text{if } \lambda(e) = (\mathsf{true}, \mathsf{R}\,x\,v) \\
P, e & \models & \phi \wedge \psi,\ \text{if } P, e \models \phi \text{ and } P, e \models \psi \\
P, e & \models & \mathsf{true} \\
P, e & \models & \neg\phi,\ \text{if } P, e \not\models \phi \\
P, e & \models & \Box^{-1}\phi,\ \text{if } (\forall d < e)\ P, d \models \phi
\end{array}$$

Definition 7.1. Define $P \models \phi$ if $(\forall e \in E)\ P, e \models \phi$ and $\mathcal{P} \models \phi$ if $(\forall P \in \mathcal{P})\ P \models \phi$.

Thus, $P \models \phi \wedge \Box^{-1}\phi$ whenever $P \models \phi$; a fact that relies crucially on the universal quantification over all events of a pomset in the above definition.

Define the other connectives standardly following DeMorgan laws.

$$\begin{array}{lcl}
\Diamond^{-1}\phi & = & \neg(\Box^{-1}\neg\phi) \\
\mathsf{false} & = & \neg(\mathsf{true}) \\
\phi \vee \psi & = & \neg(\neg(\phi) \wedge \neg(\psi)) \\
\phi \Rightarrow \psi & = & \neg(\phi) \vee\ \psi
\end{array}$$

The past operators do not include the current instant, and thus they do *not* satisfy the rule $\Box^{-1}\phi \Rightarrow \Diamond^{-1}\phi$. because the roots — the minimal elements of a pomset — always validate $\Box^{-1}\phi$ and always invalidate $\Diamond^{-1}\phi$. However, they do satisfy:

Lemma 7.2. *Given an pomset P.*

$$P \models (\phi \Rightarrow \Diamond^{-1}\phi) \Rightarrow \neg\phi \qquad \text{(Coinduction)}$$

$$P \models (\Box^{-1}\phi \Rightarrow \phi) \Rightarrow \phi \qquad \text{(Induction)}$$

Proof. We prove that any node in a pomset satisfies these formulas. The proof for both rules proceeds by induction on the length of the maximal path from a root to a node. □

We now present two proof rules for programs.

Proof rules for programs. The first rule captures the semantics of local variables. Define $\mathsf{closed}(x) = (\mathsf{R}\,x\,v \Rightarrow \Diamond^{-1}\mathsf{W}\,x\,v)$. Although this definition does not mention intervening writes, it is sufficient for our example. It is straightforward to establish that following rule is sound:

$$\frac{\phi \text{ is independent of } x \qquad P \models \mathsf{closed}(x) \Rightarrow \phi}{\nu x \,.\, P \models \phi} \qquad \text{(Closing } x\text{)}$$

The second rule describes composition, in the style of Abadi and Lamport [1993]. To simplify the presentation, we consider the special case with a single invariant. In order to state the theorem, we generalize the satisfaction relation to include environment assumptions. Let $\mathsf{Models}(\phi) = \{\mathcal{P} \mid \mathcal{P} \models \phi\}$ be the set of pomsets that satisfy ϕ. We say that ϕ is prefix closed if $\mathsf{Models}(\phi)$ is prefix-closed[7]. Define $\phi, \mathcal{P} \models \psi$ if $\mathsf{Models}(\phi) \parallel \mathcal{P} \models \psi$.

Proposition 7.3. *Let ϕ be prefix-closed. Let $\mathcal{P}_1, \mathcal{P}_2$ be augmentation-closed.Then:*

$$\frac{\phi, \mathcal{P}_1 \models \phi \qquad \phi, \mathcal{P}_2 \models \phi}{\mathcal{P}_1 \parallel \mathcal{P}_2 \models \phi} \qquad \text{(Composition)}$$

Proof sketch. We will show that all prefixes in the prefix closures of $\mathcal{P}_1 \parallel \mathcal{P}_2$ satisfy the required property. Proof proceeds by induction on prefixes of $P \in \mathcal{P}_1 \parallel \mathcal{P}_2$.

The case for empty prefix follows from assumption that ϕ is prefix closed.

For the inductive case, consider $P \in P_1 \parallel P_2$ where $P_i \in \mathcal{P}_i$. Since $\mathcal{P}_1$ and $\mathcal{P}_2$ are augmentation closed, we can assume that the restriction of P to the events of P_i coincides with P_i, for $i = 1, 2$. Consider a prefix P' derived by removing a maximal element e from P. Suppose e comes from P_1 (the other case is symmetric). Since P_2 is a prefix of P' and $P' \models \phi$ by induction hypothesis, we deduce that $P_2 \models \phi$. Since $P_1 \in \mathcal{P}_1$, by assumption $\phi, \mathcal{P}_1 \models \phi$ we deduce that $P \models \phi$. □

A more general principle, in the spirit of Abadi and Lamport [1993] can be proved. We chose the simple case of temporal invariants to illustrate the idea in a simple form. Even this simple version has interesting consequences.

Example 7.4. We now turn the conditional TAR-pit program, given in (7):

$$\texttt{var}\,x\texttt{:=0;}\;\texttt{var}\,y\texttt{:=0;}\;\texttt{var}\,z\texttt{:=0;}\;(y\texttt{:=}x \parallel \texttt{if(}z\texttt{)\{}x\texttt{:=1\}else\{}x\texttt{:=}y\texttt{;}\;a\texttt{:=}y\texttt{\}} \parallel z\texttt{:=1}) \qquad (9)$$

We prove the formula $\neg\Diamond^{-1}(\mathsf{W}\,a\,1)$ holds for the models of this program in our semantics. We start with the following invariant, which holds for each of the three threads, and thus, by composition, for the aggregate program:

$$\begin{gathered}[\Diamond^{-1}(\mathsf{W}\,y\,1) \Rightarrow \Diamond^{-1}(\mathsf{R}\,x\,1)] \wedge \\ [\Diamond^{-1}(\mathsf{W}\,a\,1) \Rightarrow (\Diamond^{-1}(\mathsf{R}\,y\,1) \wedge \Box^{-1}(\mathsf{W}\,x\,1 \Rightarrow \Diamond^{-1}(\mathsf{R}\,y\,1)))]\end{gathered}$$

Closing y, we have, $\Diamond^{-1}(\mathsf{R}\,y\,1) \Rightarrow \Diamond^{-1}(\mathsf{W}\,y\,1)$ which we substitute into the left conjunct to get:

$$\Diamond^{-1}(\mathsf{R}\,y\,1) \Rightarrow \Diamond^{-1}(\mathsf{R}\,x\,1)$$

which in turn we substitute into the right conjunct to get:

$$\Diamond^{-1}(\mathsf{W}\,a\,1) \Rightarrow (\Diamond^{-1}(\mathsf{R}\,x\,1) \wedge \Box^{-1}(\mathsf{W}\,x\,1 \Rightarrow \Diamond^{-1}(\mathsf{R}\,x\,1)))$$

[7] P' is a prefix of P if $E' \subseteq E$, $e \in E'$ and $d \leq e$ imply $d \in E'$, and $(\lambda', \leq')$ coincide with $(\lambda, \leq)$ for elements of E'.

Closing x, we can replace $\Diamond^{-1}(\mathsf{R}\,x\,1)$ with $\Diamond^{-1}(\mathsf{W}\,x\,1)$:

$$\Diamond^{-1}(\mathsf{W}\,a\,1) \Rightarrow (\Diamond^{-1}(\mathsf{W}\,x\,1) \wedge \Box^{-1}(\mathsf{W}\,x\,1 \Rightarrow \Diamond^{-1}(\mathsf{W}\,x\,1)))$$

Applying coinduction to the right conjunct, we have:

$$\Diamond^{-1}(\mathsf{W}\,a\,1) \Rightarrow (\Diamond^{-1}(\mathsf{W}\,x\,1) \wedge \Box^{-1}(\neg\mathsf{W}\,x\,1))$$

Simplifying, we have, as required:

$$\Diamond^{-1}(\mathsf{W}\,a\,1) \Rightarrow \mathsf{false}$$

Example 7.5 (Jeffrey and Riely [2016]; Svendsen et al. [2018]). We show that the following program never writes 2 to z.

$$\mathtt{var}\,x := 0;\ \mathtt{var}\,y := 0;\ \mathtt{var}\,z := 0;\ (y := x + 1;\ z := y \parallel x := y)$$

We start with the following invariant, which holds for each of the two threads, and thus, by composition, for the aggregate program:

$$\begin{aligned}
&[(\Diamond^{-1}(\mathsf{W}\,y\,v_1) \wedge \Diamond^{-1}(\mathsf{W}\,y\,v_2)) \Rightarrow ((v_1 == v_2) \vee (v_1 == 0) \vee (v_2 == 0))]\wedge\\
&[\mathsf{W}\,x\,1 \Rightarrow (\Diamond^{-1}(\mathsf{R}\,y\,1))]\wedge\\
&[\mathsf{W}\,y\,2 \Rightarrow (\Diamond^{-1}(\mathsf{R}\,x\,1))]\wedge\\
&[\mathsf{W}\,z\,2 \Rightarrow (\Diamond^{-1}(\mathsf{R}\,y\,2))]
\end{aligned}$$

Closing both x, y and substituting, we deduce:

$$\mathsf{W}\,z\,2 \Rightarrow (\Diamond^{-1}(\mathsf{W}\,y\,2) \wedge \Diamond^{-1}(\mathsf{W}\,y\,2))$$

Using the first conjunct, we deduce the required conclusion $\mathsf{W}\,z\,2 \Rightarrow \mathsf{false}$.

8 COMPARISON TO RELATED WORK

A memory consistency model for a shared-memory multiprocessor defines the values that a read may return. There has been extensive research on hardware and software memory models. Podkopaev et al. [2019] provided formal foundations to bridge the gap between the two. Alglave [2010] provided a historical and conceptual perspective on hardware memory models.

This paper focusses on software memory models. For a survey and history of the Java Memory Model, see [Lochbihler 2013]; for a survey and history of C11, see [Batty 2015].

This paper follows a line of work on relaxed memory models in the framework of true concurrency [Cenciarelli et al. 2007; Jeffrey and Riely 2016; Pichon-Pharabod and Sewell 2016].

We have already discussed related work in context earlier in the paper. Here, we discuss in more detail the relationship with the most closely related work.

8.1 Memory models via program transformations

We discuss prior approaches to defininig memory models using program transformations in chronological order. The general flavor in these approaches is to consider SC executions, albeit with the threads subject to sequential program transformation.

Saraswat et al. [2007] aimed to describe a JMM-like model this way. DRF-SC holds, but Sevcik [2011] discovered that it permits OOTA behavior.

Ferreira et al. [2010] described a relaxed memory model as being parametrized by available program transformations. They showed DRF-SC, but do not study OOTA, relate to concrete models, or demonstrate compilation results.

Demange et al. [2013] developed BMM, a model whose only permitted reordering is of a relaxed write with a following relaxed read. The paper develops an axiomatic and studies effects on compilation. By design, it is designed as a restriction of the JMM that invalidates several compiler optimizations.

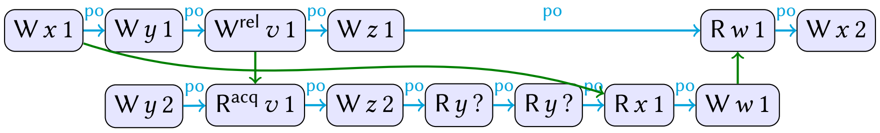
Lahav and Vafeiadis [2016] characterized TSO as being derived by considering Write-Read (WR) reordering and Read-After-Write (RAW) elimination. They also showed that the release acquires of C11 are less expressive than considering WR, RAW and thread-inlining. Our paper is inspired by their implicit challenge: "Some memory models can be defined via transformations. But there is more to weak memory than transformations."

8.2 Modeling the invalidation of load buffering

In this subsection, we argue that our paper is relevant even to the reader who desires a semantics that invalidates "load buffering," or "reads from the future." There are two related lines of research that suggest to prohibit the reordering of loads with following stores.

- Dolan et al. [2018] imposed this restriction in order to "bound data races in space and time." Consider the following program:

$$\begin{array}{ll} & x := 1;\ y := 1;\ v^{\mathsf{rel}} := 1;\ z := 1;\ \texttt{if}(w)\{x := 2\} \\ \parallel & y := 2;\ r := v^{\mathsf{acq}};\ z := 2\ ;\ \texttt{if}(r)\{w := y - y + x\} \end{array}$$



  In their model, one can deduce that w must be assigned 1, despite the past race on y, concurrent race on z and future race on x.
- In the C11 family of models, eg. see [Boehm and Demsky 2014; Lahav et al. 2017; Vafeiadis and Narayan 2013], load buffering is forbidden to avoid OOTA. The elegant discussion in Boehm [2018] elucidates the issues via a series of "OOTA" like examples. In particular, to rule out example 8.

Our model can easily capture models that forbid load buffering. It necessitates two changes.

- Modify the semantic rule for read to remove the possibility of an internal read:

$$[\![r := [L];\ C]\!] = \bigcup_{x=[w]} (L = w) \triangleright \bigcup_{v} (\mathsf{R}\,x\,v) \Rightarrow [\![C]\!][x/r]$$

- Modify item 6b in the definition of prefixing to require order from a read to subsequent write. Item 6b′ becomes: if e is a write then $c <' e$. In 6b′ we removed the "or or $\lambda'_\Phi(e)$ implies $\lambda_\Phi(e)$."

While Dolan et al. [2018] already describes DRF and the compilation to ARMV8, our approach yields the following new insights.

- Compositional treatment of temporal invariants from §7 holds mutatis mutandis, since all executions of the restricted model are already accounted for in the general model.
- With regards to single-threaded optimizations in §6, our approach provides different methods to prove optimizations, which complement the extant methods of Dolan et al. [2018] with data sensitivity; for example, our methods provides a simple proof of the transformation of $\texttt{if}(M)\{x := 1;\ C\}\,\texttt{else}\,\{x := 1;\ D\}$ to $x := 1;\ \texttt{if}(M)\{C\}\,\texttt{else}\,\{D\}$.

8.3 Revisiting OOTA

Batty et al. [2015] describe the problem of thin-air executions. Boehm [2018] revisits community discussions on "OOTA" like examples. We classify the models on the basis of their behavior on three examples: (1), (7) and (8).

- RC11[Lahav et al. 2017], JMM [Manson et al. 2005] and the related models of Jagadeesan et al. [2010] and Kang et al. [2017] forbid only (1).
- The model of this paper forbids (1) and (7).
- Forbidding load buffering as per [Boehm 2018; Dolan et al. 2018] forbids all three examples. RC11[Lahav et al. 2017] also forbids all three examples.

One of our novel contributions to this line of research is the desideratum of compositional reasoning of safety properties. The presence of such compositional reasoning suffices to remove the undesirable effects of OOTA, eg. type safety. And while OOTA in a model is an elusive creature, and hence hard to prove or disprove, the support of a model for a compositional proof principle is objectively provable or falsifiable.

8.4 Relationship to models of speculation

Disselkoen et al. [2019] is a close formal cousin to our approach. In the spectrum from micro architectures to architectures to software,whereas Disselkoen et al. [2019] lies between the micro and architectural levels, we aim to be as abstract as possible to bridge the gap to programming languages. This is reflected in the several differences in the formal model.

We consider a single global acyclic order in contrast to the two studied in Disselkoen et al. [2019]. We do not track false branches of conditionals, as seen in our monotonicity axiom, whereby any event with a precondition false can only have false successors in $<$, whereas in that paper, speculation on false branches can lead to observable effects. We are also less accurate about tracking reads, as reflected in the removal of program order between reads and reads.

Disselkoen et al. [2019] does not study the properties as a memory model. We conjecture that our proofs of DRF and compositional reasoning can be carried over to that setting. Disselkoen et al. [2019] also preserves some po between reads, whereas we do not; so, we certainly support more compiler optimizations of single threaded code.

9 CONCLUSIONS

We have defined a relaxed memory model, using standard tools from the semantics and demonstrated that it satisfies the fundamental criterion for software memory models, namely "the memory model must strike a balance between ease-of-use for programmers and implementation flexibility for system designers."

One key lesson we draw is that any software relaxed memory model should support the cyclical composition proof rule of Abadi and Lamport [1993] for temporal properties. We posit that all the benefits of OOTA, such as typing guarantees, can be derived from this principle.

We conclude with a suggestion for the consideration of programming language designers. The semantics of a programming language should support a clear and executable prescription of dependency in a program. Such a specification provides a nice interface between the language user and the compiler writer, and a clear bridge from the compiler writer to architectures.

Our current paper is to be viewed as a step towards this goal. We will explore developing an executable specification based on these ideas in future work.

10 CONCLUSIONS

From [?]:

> It would be nice to have a succinct description of the set of test cases in which perturbation functions introduced inconsistencies. Ali Sezgin pointed out this set is described by $rf \cup sdep$, where rf is the reads-from relationship and sdep is "semantic dependence", roughly defined as those dependency relationships in which

> at least some changes in the value at the head of the dependency relationship propagate through, resulting in a change at the tail of that relationship.
>
> Prohibiting executions that have cycles in $rf \cup sdep$ can therefore be expected to prohibit OOTA behaviors.
>
> One beneficial consequence of this relationship to semantic dependency is that $rf \cup nsdep$ cycles are allowed, where $nsdep \cap sdep$ is the empty set and where $nsdep \cup sdep = dep$. This means that the compiler is free to replace expressions that are known to always result in a single value with the corresponding constant, without danger of introducing OOTA behavior. We hypothesize that non-speculative code-reordering optimizations are similarly unable to introduce OOTA behavior.
>
> Defining "semantic dependency" sufficiently for formal modeling remains an open issue. In the general case, this the question of whether or not a given dependency is a semantic dependency is of course undecidable. However, this question can be decided straightforwardly in many common cases. One approach would be to flag dependencies that the tool was unable to classify. Another approach would be to consider cases that a given compiler might optimize, and to classify other cases as semantic dependencies.

And also this:

> This perturbation analysis appears to be equivalent to requiring that $rf \cup sdep$ be acyclic. This is an extremely important result: It means that any compiler optimization that substitutes a constant value for a read known to return that value cannot induce OOTA behavior. This constraint should also ensure that non-speculative code-movemenet optimizations should be similarly unable to induce OOTA behavior.
>
> Effective and efficient modeling of semantic dependencies (sdep) remains an important open problem, although there is important work in progress [Pichon-Pharabod and Sewell 2016].

REFERENCES

Martín Abadi and Leslie Lamport. 1993. Composing Specifications. *ACM Trans. Program. Lang. Syst.* 15, 1 (Jan. 1993), 73–132. https://doi.org/10.1145/151646.151649

Sarita V. Adve and Mark D. Hill. 1990. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990.* ACM, 2–14. https://doi.org/10.1145/325164.325100

Sarita V. Adve and Mark D. Hill. 1993. A Unified Formalization of Four Shared-Memory Models. *IEEE Trans. Parallel Distrib. Syst.* 4, 6 (1993), 613–624. https://doi.org/10.1109/71.242161

J. Alglave. 2010. *A shared memory poetics.* PhD thesis. Université Paris 7 and INRIA.

Jade Alglave, Anthony C. J. Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The semantics of power and ARM multiprocessor machine code. In *Proceedings of the POPL 2009 Workshop on Declarative Aspects of Multicore Programming, DAMP 2009, Savannah, GA, USA, January 20, 2009.* 13–24. https://doi.org/10.1145/1481839.1481842

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Proc. European Symp. on Programming.* 283–307.

Mark John Batty. 2015. *The C11 and C++11 concurrency model.* Ph.D. Dissertation. University of Cambridge, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.708458

Hans Boehm. 2018. 1217R0: Out-of-thin-air, revisited, again. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1217r0.html.

Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2618128.2618134

S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. 1984. A Theory of Communicating Sequential Processes. *J. ACM* 31, 3 (June 1984), 560–599. https://doi.org/10.1145/828.833

Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. 2007. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*. 331–346. https://doi.org/10.1007/978-3-540-71316-6_23

Delphine Demange, Vincent Laporte, Lei Zhao, Suresh Jagannathan, David Pichardie, and Jan Vitek. 2013. Plan B: a buffered memory model for Java. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. 329–342. https://doi.org/10.1145/2429069.2429110

C. Disselkoen, R. Jagadeesan, A. Jeffrey, and J. Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 930–947. https://doi.org/10.1109/SP.2019.00047

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 242–255. https://doi.org/10.1145/3192366.3192421

Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. 2010. Parameterized Memory Models and Concurrent Separation Logic. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. 267–286. https://doi.org/10.1007/978-3-642-11957-6_15

Jay L. Gischer. 1988. The equational theory of pomsets. *Theoretical Computer Science* 61, 2 (1988), 199–224. https://doi.org/10.1016/0304-3975(88)90124-7

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Michael Huth, Radha Jagadeesan, and David A. Schmidt. 2001. Modal Transition Systems: A Foundation for Three-Valued Program Analysis. In *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. 155–169. https://doi.org/10.1007/3-540-45309-1_11 Full version in Mathematical Structures in Computer Science 2004(14), 469—505.

Radha Jagadeesan, Corin Pitcher, and James Riely. 2010. Generative Operational Semantics for Relaxed Memory Models. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science)*, Andrew D. Gordon (Ed.), Vol. 6012. Springer, 307–326. https://doi.org/10.1007/978-3-642-11957-6_17

A. Jeffrey and J. Riely. 2016. 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, New York, NY, USA, July 5-8, 2016*, M. Grohe, E. Koskinen, and N. Shankar (Eds.). ACM, 759–767. https://doi.org/10.1145/2933575.2934536

J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189. https://doi.org/10.1145/3009837

Ori Lahav and Viktor Vafeiadis. 2016. Explaining Relaxed Memory Models with Program Transformations. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*. 479–495. https://doi.org/10.1007/978-3-319-48989-6_29

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing sequential consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 618–632. https://doi.org/10.1145/3062341.3062352

L. Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Leslie Lamport. 1986. On Interprocess Communication. Part I: Basic Formalism. *Distributed Computing* 1, 2 (1986), 77–85. https://doi.org/10.1007/BF01786227

Kim Guldstrand Larsen and Bent Thomsen. 1988. A Modal Process Logic. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*. IEEE Computer Society, 203–210. https://doi.org/10.1109/LICS.1988.5119

Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. 1985. The Glory of the Past. In *Proceedings of the Conference on Logic of Programs*. Springer-Verlag, London, UK, UK, 196–218. http://dl.acm.org/citation.cfm?id=648065.747612

A. Lochbihler. 2013. Making the Java memory model safe. *ACM Trans. Program. Lang. Syst.* 35, 4 (2013), 12:1–12:65. https://doi.org/10.1145/2518191

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. *SIGPLAN Not.* 40, 1 (Jan. 2005), 378–391. https://doi.org/10.1145/1047659.1040336

Robin Milner. 1999. *Communicating and Mobile Systems: The π-calculus.* Cambridge University Press, New York, NY, USA.

J. Misra and K. M. Chandy. 1981. Proofs of Networks of Processes. *IEEE Trans. Softw. Eng.* 7, 4 (July 1981), 417–426.

Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 622–633. https://doi.org/10.1145/2837614.2837616

Gordon Plotkin and Vaughan Pratt. 1997. Teams Can See Pomsets (Preliminary Version). In *Proceedings of the DIMACS Workshop on Partial Order Methods in Verification (POMIV '96)*. AMS Press, Inc., New York, NY, USA, 117–128. http://dl.acm.org/citation.cfm?id=266557.266600

Amir Pnueli. 1985. In Transition From Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems*, Krzysztof R. Apt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 123–144.

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *PACMPL* 3, POPL (2019), 69:1–69:31. https://dl.acm.org/citation.cfm?id=3290382

W. Pugh. 2004. Causality Test Cases. http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html.

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. 2007. A Theory of Memory Models. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. ACM, New York, NY, USA, 161–172. https://doi.org/10.1145/1229428.1229469

Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. 2011. Understanding POWER multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 175–186. https://doi.org/10.1145/1993498.1993520

Sevcik. 2011. OOTA in the PPoPP memory model. Personal Communication.

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM* 53, 7 (2010), 89–97. https://doi.org/10.1145/1785414.1785443

Eugene W. Stark. 1985. A proof technique for rely/guarantee properties. In *Foundations of Software Technology and Theoretical Computer Science*, S. N. Maheshwari (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 369–391.

Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A Separation Logic for a Promising Semantics. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science)*, Vol. 10801. Springer, 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 867–884. https://doi.org/10.1145/2509136.2509532

Jaroslav Ševčík. 2008. *Program Transformations in Weak Memory Models.* PhD thesis. Laboratory for Foundations of Computer Science, University of Edinburgh.

A MODAL POMSETS

In order to perform a sharper analysis of dependency, we present an alternate semantics using modal pomsets defined below. Modal pomsets make a formal distinction between strong order and weak order.

Definition A.1. A *modal (memory model) pomset* is a tuple $(E, \trianglelefteq, \leq, \lambda)$, such that

- $(E, \leq, \lambda)$ is a (memory model) pomset, and
- $\trianglelefteq \subseteq \leq$ is a partial order.

We write $d \triangleleft e$ when $d \trianglelefteq e$ and $d \neq e$, and similarly for $\leq$. Thus, $(\trianglelefteq \cup \text{eco})^* \subseteq \leq$.

We list out a few observations to illustrate the relationship between modal pomsets and pomsets. We are given a modal pomset, $(E, \trianglelefteq, \leq, \lambda)$. Then:

- $(E, \leq, \lambda)$ is a pomset with the same reads-from relation.
- Let eco be the restriction of $\leq$ to conflicting actions on the same location. Then, $(E, \trianglelefteq, (\trianglelefteq \cup \text{eco})^*, \lambda)$ is a modal pomset, and $(\trianglelefteq \cup \text{eco})^* \subseteq \leq$.

Changes to definitions. The definition of the semantics of programs using modal pomset largely follows the one using pomsets. We sketch the changes to definitions below.

- We say that d *fulfills* e *on* x if d writes v to x, e reads v from x,
  - $d \triangleleft e$, and
  - if an event c writes to x then either $c \leq d$ or $e \leq c$.
- Augmentation has to include $\triangleleft$. i.e P' is an *augmentation* of P if $E' = E$, $\lambda' = \lambda$, $\trianglelefteq' \supseteq \trianglelefteq$, and $\leq' \supseteq \leq$.
- The definitions of substitution, restriction and the filtering operations stay the same, with $\trianglelefteq$ carried over unchanged. For example, substitution is defined as follows:
  Let $\mathcal{P}\sigma$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that: $E' = E$, $\trianglelefteq' = \trianglelefteq$, $\leq' = \leq$, and $\lambda'(e) = (\psi\sigma \mid a)$ when $\lambda(e) = (\psi \mid a)$.
- In composition,we require $\trianglelefteq' \supseteq \trianglelefteq^1 \cup \trianglelefteq^2$
- The changes to the definition 3.4 of prefixing are as follows. The key changes are that synchronization and dependency enforce $\triangleleft$ whereas coherence only enforces $\leq$.
  - $\trianglelefteq' \supseteq \trianglelefteq$.
  - Item 6b changes to: if e is a write then either $c \triangleleft' e$ or $\lambda'_\Phi(e)$ implies $\lambda_\Phi(e)$.
  - Item 8 changes to: if a is an acquire or $\lambda_{\mathcal{A}}(e)$ is a release then $c \triangleleft' e$.

We use $[\![C]\!]_{\text{M}}$ to stand for the modal pomset semantics of C.

A.1 Generators.

Modal pomsets provide a characterization of generators from section 4.

Recall that *generators* in the pomset semantics are pomsets that are minimal with respect to augmentation and implication. These generators are induced by pomsets that are minimal with respect to augmentation and implication in the modal pomset semantics in the following sense.

$(E, \leq, \lambda)$ is a generator for $[\![C]\!]$ if there exists $(E, \triangleleft, \leq, \lambda) \in [\![C]\!]_{\text{M}}$ minimal wrt augmentation and implication, and $\leq = (\trianglelefteq \cup \text{eco})^*$.

Furthermore, any strong order that is outside of program order must be induced by a reads-from. In the two-thread case, we can state the latter property as follows: suppose e and d are not related by program order and $e \triangleleft d$; then there exist d' that reads-from e' such that $e \xrightarrow{\text{po}} e'$, $d' \xrightarrow{\text{po}} d$ and $e \triangleleft e' \triangleleft d' \triangleleft d$.

A.2 Closure properties

The fine grain analysis of dependency in the modal semantics allows us to establish some closure properties of the semantics of programs.

We consider programs of the form $\vec{x} := \vec{0}$; `fence`; C, where C is restriction-free. Thus, all memory locations are initialized to 0, initialization happens-before the execution of any command,

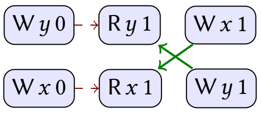
We say that $P' = P|_{E'}$ when $E' \subseteq E$, $\lambda' = \lambda|_{E'}$, and $\leq' = \leq|_{E'}$.

Definition A.2. Let $(P \text{ after } e) = \{d \in E \mid e \leq d\}$ be the set of events that follow e in P.

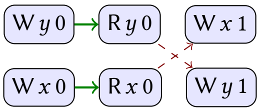
The semantics of read is "input"-enabled, since it permits the read of any visible value. Thus, any racy read in a program can be replaced by a read of a earlier value (wrt eco), even while the races with existing independent writes are maintained. A canonical example to keep in mind for this lemma is the program:

$$(y := 0;\ r := y;\ x := 1) \parallel (x := 0;\ s := x;\ y := 1)$$

with both registers getting value 1 via the execution:



The lemma constructs the execution:



Lemma A.3. *Let $P \in [\![C]\!]_{\mathsf{M}}$ be a top level pomset. Let $e \in P$ read from write event d on x, $\neg(d \xrightarrow{\text{hb}} e)$. Then, there exists $P' \in [\![C]\!]_{\mathsf{M}}$ such that:*

- *e' reads from x, with matching write event d', such that $d' \xrightarrow{\text{eco}} d$ in P'*
- *The restriction of $\trianglelefteq$ in P to $E_P \setminus (P \text{ after } e)$ agrees with the the restriction of $\trianglelefteq$ in P' to $E_{P'} \setminus (P \text{ after } e)$ in P'.*
- *The restriction of $\leq$ in P to $E_P \setminus (P \text{ after } e)$ agrees with the the restriction of $\leq$ in P' to $E_{P'} \setminus (P \text{ after } e)$ in P'.*

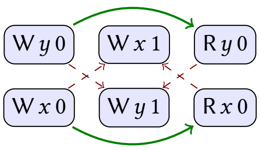
Proof. The form of C ensures that there is always a write to x that is related by $\xrightarrow{\text{hb}}$ to any read. Thus, there is at least one other write than can satisfy the read recorded as e.

The key observation behind the proof is that change in a prefixing read action can only affect the events that are dependent, ie. in the $\triangleleft$ order to the read action. □

In the following lemma, invert the eco relationship between a read and a write. A canonical example to keep in mind for this lemma is the program:

$$(y := 0;\ x := 1;\ r := y) \parallel (x := 0;\ y := 1;\ s := x)$$

with both registers getting value 0 via the execution:



The lemma constructs the execution:

$$\begin{tikzcd}
\mathsf{W}\,y\,0 \arrow[dr, dashed] & \mathsf{W}\,x\,1 \arrow[dr] & \mathsf{R}\,y\,1 \\
\mathsf{W}\,x\,0 \arrow[ur, dashed] & \mathsf{W}\,y\,1 \arrow[ur] & \mathsf{R}\,x\,1
\end{tikzcd}$$

Lemma A.4. *Let $P \in [\![C]\!]_{\mathsf{M}}$ be a top-level pomset. Let $d \in P$ be a write on x. Let $e \in P$ read from x such that $e \xrightarrow{\mathsf{eco}} d$ and $\neg(e \lhd d)$. Then, there exists $P' \in [\![C]\!]_{\mathsf{M}}$ such that:*

- *$e' \in P' \setminus P$ reads from x, with matching write d.*
- *The restriction of $\trianglelefteq$ in P to $E_P \setminus (P \ \text{after} \ e)$ agrees with the the restriction of $\trianglelefteq$ in P' to $E_{P'} \setminus (P \ \text{after} \ e)$.*

Proof. The proof proceeds similar to the above proof; in this case, replace the value read in e to come from d. □

Any new event d' in P' after e' reading from x cannot have a matching write event $d'' \xrightarrow{\mathsf{eco}} d$ since that implies $d' \xrightarrow{\mathsf{eco}} d$ and a eco cycle $d \lhd e \lhd e' \xrightarrow{\mathsf{eco}} d$. Thus, the above lemma can be iterated if the new pomset is has any further reads that precede d in eco, so we can finally derive a pomset with no reads and writes satisfying the hypothesis of the lemma.

The eco order between writes that are not related by $<$ can be reversed. A canonical example to keep in mind for this lemma is the program:

$$(x := 1) \parallel (x := 0)$$

$$\begin{tikzcd}
\mathsf{W}\,x\,1 \arrow[r, dashed] & \mathsf{W}\,x\,0
\end{tikzcd}$$

The lemma constructs the execution:

$$\begin{tikzcd}
\mathsf{W}\,x\,1 & \mathsf{W}\,x\,0 \arrow[l, dashed]
\end{tikzcd}$$

Lemma A.5. *Let $P \in [\![C]\!]_{\mathsf{M}}$ be a top level pomset. Let d, e be a writes to x such that:*

- *$d \leq e$*
- *forall writes c to x such that $d \leq c \leq e$, it is the case that $\neg(c \lhd e)$ and $\neg(c \xrightarrow{\mathsf{po}} e)$*

Then, there exists $P' \in [\![C]\!]_{\mathsf{M}}$ such that $E_P = E_{P'}$, $\trianglelefteq_P = \trianglelefteq_{P'}$, and $e \leq d$ in P'.

Proof. We show how to interchange e, d adjacent in $\leq$, ie. we assume that $\neg(\exists c)\ d \leq c \leq e$. The full proof follows by induction.

Since $[\![C]\!]$ is augmentation closed, it suffices to show that we can build P' while satisfying the constraints between $\lhd, \leq$. We list the changes below.

- $e \leq d$ in P'
- Forall reads c matched to e, change from $d \leq c$ in P to $c \leq d$ in P'
- Forall reads c matched to d, change from $c \leq e$ in P to $e \leq c$ in P'

□

B PROOF OF DRF

In this section of the appendix, we develop a proof of DRF for modal pomsets. By the results in the earlier section, it yields DRF for the pomset semantics, since the races are identical in both models.

In the rest of this section, we assume that P is a generator for $[\![C]\!]_{\mathsf{M}}$.

We prove:

DRF1: If P does not have a race, $P \in [\![C]\!]_{\mathsf{MSC}}$.
DRF2: If P has a race, then there exists $P' \in \mathsf{closed}[\![C]\!]_{\mathsf{M}}$ such that $P' \in [\![C]\!]_{\mathsf{MSC}}$ and has a race.

Proof of DRF1. We first show that if $P \in [\![C]\!]_{\mathrm{M}} \setminus [\![C]\!]_{\mathrm{MSC}}$, then P has a race. By assumption, there is a cycle in po $\cup \lhd \cup \xrightarrow{\text{eco}}$. Let this cycle be $e_0, e'_0, e_1, e'_1, \ldots, e_n, e'_n, e_0$ where for all i, $e_i \xrightarrow{\text{po}} e'_i$ and $e'_i \not\xrightarrow{\text{po}} e'_{i+1}$. If for all i, $e'_i \xrightarrow{\text{hb}} e'_{i+1}$, then the above is a cycle in hb, which is a contradiction. So, there is at least one i such that $e'_i \not\xrightarrow{\text{hb}} e'_{i+1}$. There are two cases to consider.

- $e'_i \xrightarrow{\text{eco}} e'_{i+1}$. In this case, there is a race.
- $e'_i \lhd e'_{i+1}$. In this case, e'_i is a write and e'_{i+1} is a conflicting read, so there is a race.

Proof of DRF2. We define a size $|P|$ as follows: size(P) is the number of events in P. Since we are considering loop free programs, there is an $P \in [\![C]\!]_{\mathrm{MSC}}$ with maximum size, which we identify as size(C).

We prove by induction on size(C) − size(P') that given (P, P') such that:

- P' is a prefix of some $P' \in [\![C]\!]_{\mathrm{MSC}}$
- P' is a prefix of P under all of $\xrightarrow{\text{po}}, \leq, <$
- P has a race

there exists $P' \in [\![C]\!]_{\mathrm{M}}$ that demonstrates the race.

The required theorem follows by setting P' to be the empty pomset.

For the base case, $P' = |P|$. In this case, P is the required witness.

Otherwise, consider a maximal sequential prefix, extending P', wrt to all of po, eco, $\lhd$. If it strictly contains P', result follows from induction hypothesis.

If not, P' is already maximal. Consider the set of all events in $P \setminus P'$ that are minimal wrt hb. In particular, these events will also be minimal wrt po.

If one of these events, say e is a write, we proceed as follows. Using hb-minimality of e, we deduce po minimality of e. Using the generator properties, we deduce that e is $\lhd$-minimal . Using lemma A.4, we build P_1 from P without changing P' to ensure that there are is no read $d \in P_1 \setminus P'$ such that $d \xrightarrow{\text{eco}} e$. Using lemma A.5, we build P_2 from P_1 without changing P' to ensure that there are is no write $d \in P_2 \setminus P'$ such that $d \xrightarrow{\text{eco}} e$. Thus, e is eco-minimal in $P_2 \setminus P'$. Result follows from induction hypothesis by considering (P_2, P'_1) where P'_1 is got from P' by adding e.

So, we can assume that all events in $P \setminus P'$, say $e_0, \ldots, e_n$ that are minimal wrt hb are reads, and we have events $e'_0, e'_1, \ldots, e'_n, e_0$ such that:

$$e_i \xrightarrow{\text{po}} e'_i$$
$$e'_i\ (\text{eco} \ \cup \lhd)\ e_{(i+1) \mod n}$$

Let d be the matching write for $e_{(i+1) \mod n}$. If $d_i \in P'$bEv , then by eco prefix closure of P', $d \xrightarrow{\text{eco}} e'_i$ and $e_{(i+1) \mod n}$ eco e'_i, which is a contradiction to eco being a partial order per location. So, we can assume that $e'_i \ \lhd \ e_{(i+1) \mod n}$.

We proceed as follows. We use lemma A.3 on the pomset P and read $e_{(i+1) \mod n}$ and write e'_i to construct P_1 that changes the value read in e_j to a value from P'. P'_1 is derived adding the modified read yielded by lemma A.3 to P'. Result follows by induction hypothesis since P'_1 is a prefix of P_1 under all of $\xrightarrow{\text{po}}$, $<$, eco, P_1 has a race, and size(P'_1) = size(P') + 1.

C PROOF OF COMPILATION FOR ARMV8

In this section, we develop the proof of correctness of compilation to ARMV8. In order to ease readability, we reproduce the definitions from the main text.

Given a relation R, $R^?$ denotes reflexive closure, R^+ denotes transitive closure and R^* denotes reflexive and transitive closure. Given relations R and S, $R; S$ denotes composition.

The ARMv8 model is described using the following relations.

- [R], [W], [Acquire], [Release]: identity on reads, writes, acquires and releases.
- $[\mathcal{X}]$: relates any two events that touch the same location.

- po: program order.
- data, ctrl, addr: data, control and address dependencies.
- rf: reads-from. rf^{-1} relates each read to a matching write on the same location.
- co: coherence, which is a total order on the writes to a single location.
- $\mathsf{fr} = \mathsf{co}; \mathsf{rf}^{-1}$: from-read, which relates reads to subsequent writes.

For any relation, the cross-thread subrelation is denoted by appending e; the intra-thread subrelation is denoted by appending i. For example, $\mathsf{rfe} = \mathsf{rf} \setminus \mathsf{po}$ and $\mathsf{rfi} = \mathsf{rf} \cap \mathsf{po}$. The subrelation restriction attention to actions on the same location is given by appending loc. For example, $\mathsf{poloc} = \mathsf{po} \cap [\mathcal{X}]$.

The ARMv8 model defines the following relations. In our presentation, we have elided rules concerning fences and RMW operations.

$$\mathsf{eco} = \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{co} \qquad \text{(Extended coherence)}$$

$$\mathsf{obs} = \mathsf{rfe} \cup \mathsf{fre} \cup \mathsf{coe} \qquad \text{(Observed externally)}$$

$$\mathsf{dob} = (\mathsf{addr} \cup \mathsf{data}); \mathsf{rfi}^? \cup (\mathsf{ctrl} \cup \mathsf{data}); [\mathsf{W}]; \mathsf{coi}^? \cup \mathsf{addr}; \mathsf{po}; [\mathsf{W}] \qquad \text{(Dependency order)}$$

$$\mathsf{bob} = [\mathsf{Acquire}]; \mathsf{po} \cup \mathsf{po}; [\mathsf{Release}]; \mathsf{coi}^? \qquad \text{(Barrier order)}$$

$$\mathsf{ob} = (\mathsf{obs} \cup \mathsf{dob} \cup \mathsf{bob})^+ \qquad \text{(Acyclic order)}$$

Definition C.1. An RMW-free and fence-free execution is *ARM-consistent* if

codomain(rf) = domain(Reads)	(rf-COMPLETENESS)
For every location x, co totally orders the writes of x	(co-TOTALITY)
poloc ∪ rf ∪ fr ∪ co is acyclic	(SC-PER-LOC)
ob is acyclic	(EXTERNAL)

Given an execution graph G, we say that e is an *internal read* if $e \in \mathsf{codomain}(\mathsf{po} \cap \mathsf{rf})$. We are going to translate internal reads of execution graphs into internal reads of the semantics.

From G we construct a candidate pomset P as follows:

- $E = \mathsf{E}$,
- $\lambda_{\mathcal{A}}(e) = \tau\mathsf{lab}(e)$, if e is a relaxed internal read,
- $\lambda_{\mathcal{A}}(e) = \mathsf{lab}(e)$, if e is not a relaxed internal read,
- $\lambda_{\Phi}(e) = \mathsf{true}$,
- $\leq = \mathsf{ob}$, and
- $\leq = (\mathsf{ob} \cup \mathsf{eco})^*$

To reempphasize, in this candidate pomset, ob is calculated by considering the definition of ob without rfi, ie.:

$$\mathsf{dob} = (\mathsf{addr} \cup \mathsf{data}); \cup(\mathsf{ctrl} \cup \mathsf{data}); [\mathsf{W}]; \mathsf{coi}^? \cup \mathsf{addr}; \mathsf{po}; [\mathsf{W}] \qquad \text{(Dependency order)}$$

$$\mathsf{bob} = [\mathsf{Acquire}]; \mathsf{po} \cup \mathsf{po}; [\mathsf{Release}]; \mathsf{coi}^? \qquad \text{(Barrier order)}$$

$$\mathsf{ob} = (\mathsf{obs} \cup \mathsf{dob} \cup \mathsf{bob})^+ \qquad \text{(Acyclic order)}$$

We show that P is a top-level pomset, reasoning as follows.

- $\leq$ is a partial order. This holds since G.ar is acyclic.
- $\lambda_{\Phi}(e)$ implies $\lambda_{\Phi}(d)$ whenever $d \leq e$. Trivial, since every formula is true.
- e is location independent. Trivial, since every formula is true.
- If e reads v from x, then there is some d such that
  - $d < e$,
  - d writes v to x, and
  - if c writes to x then either $c \leq d$ or $e \leq c$.

C.1 Proof that $(\mathsf{ob} \cup \mathsf{eco})^*$ is irreflexive.

Proof of lemma 5.2. Let e, d be distinct events and d' $(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})$ d $((\xrightarrow{\mathsf{eco}} \cap \xrightarrow{\mathsf{po}}) \setminus \xrightarrow{\mathsf{rfi}})$ e $(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})$ e'. Then $d' \xrightarrow{\mathsf{ob}} e'$.

Proof. If d' is an acquire, or e is an release, or e' is a release, result is immediate.

We next consider the case where e is a read. In this case, d is a write. Since d $((\xrightarrow{\mathsf{eco}} \cap \xrightarrow{\mathsf{po}}) \setminus \xrightarrow{\mathsf{rfi}})$ e, there is a write d_1 such that $d \xrightarrow{\mathsf{coe}} d_1 \xrightarrow{\mathsf{rfe}} e'$. So, $d \xrightarrow{\mathsf{ob}} e$ and result follows in this case.

So, it suffices to prove the following assuming that d' is not an acquire and e' is not a release and e is not a release or a read and e, d are distinct.

- If d' $(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})$ $d(\xrightarrow{\mathsf{eco}} \cap \xrightarrow{\mathsf{po}})e$ then $d' \xrightarrow{\mathsf{ob}} e$.
- If d $(\xrightarrow{\mathsf{eco}} \cap \xrightarrow{\mathsf{po}})$ $e(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})e'$ then $d \xrightarrow{\mathsf{ob}} e'$.

We first prove that if d' $(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})$ d $(\xrightarrow{\mathsf{eco}} \cap \xrightarrow{\mathsf{po}})$ e then $d' \xrightarrow{\mathsf{ob}} e$. Proof proceeds by cases on the witness for d' $(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})$ d.

- If $d' \xrightarrow{\mathsf{bob}} d$, then:
$$d' \ ([\mathrm{Acquire}]; \mathsf{po} \cup \mathsf{po}; [\mathrm{Release}]; \mathsf{coi}^?) \ d$$
Since d' is not an acquire, $d'(\mathsf{po}; [\mathrm{Release}]; \mathsf{coi}^?)d$, so d is a write. Since e is not a read, $d \xrightarrow{\mathsf{coi}} e$. Thus, result follows.
- If $d' \xrightarrow{\mathsf{dob}} d$, then:
$$d' \ ((\mathsf{ctrl} \cup \mathsf{data}); [\mathrm{W}]; \mathsf{coi}^? \cup \mathsf{addr}; \mathsf{po}; [\mathrm{W}] \ d$$
So, d is a write. Since e is also a write, we deduce that
$$d' \ ((\mathsf{ctrl} \cup \mathsf{data}); [\mathrm{W}]; \mathsf{coi}^? \cup \mathsf{addr}; \mathsf{po}; [\mathrm{W}] \ e$$

We next prove that if d $(\xrightarrow{\mathsf{eco}} \cap \xrightarrow{\mathsf{po}})$ e $(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})$ e' then $d \xrightarrow{\mathsf{ob}} e'$, under the assumptions that e' is not a release and e is not a release or a read and e, d are distinct.

Proof proceeds by cases on the witness for $e(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})e'$.

- If $e \xrightarrow{\mathsf{bob}} e'$, then:
$$e \ ([\mathrm{Acquire}]; \mathsf{po} \cup \mathsf{po}; [\mathrm{Release}]; \mathsf{coi}^?) \ e'$$
Since e is not a read, $e(\mathsf{po}; [\mathrm{Release}]; \mathsf{coi}^?)e'$. Result follows since $d \xrightarrow{\mathsf{po}} e$.
- If $e \xrightarrow{\mathsf{dob}} e'$, then e is a read.

□

Lemma C.2. *If $d \xrightarrow{\mathsf{ob}} e$ then $\neg(e \xrightarrow{\mathsf{eco}} d)$.*

Proof. Proof by contradiction. Let
$$e \xrightarrow{\mathsf{ob}} e' \xrightarrow{\mathsf{eco}} d' \xrightarrow{\mathsf{ob}} d \xrightarrow{\mathsf{ob}} c \xrightarrow{\mathsf{ob}} c' \xrightarrow{\mathsf{ob}} e$$
where $e' \xrightarrow{\mathsf{po}} d'$.

By lemma 5.2, if $e \neq e'$, we deduce $e \xrightarrow{\mathsf{ob}} d'$, and thus $e \xrightarrow{\mathsf{ob}} d$. If $d \neq d'$, we deduce $e' \xrightarrow{\mathsf{ob}} d$ and thus $e \xrightarrow{\mathsf{ob}} d$.

Thus, if $e \neq e'$ or $d \neq d'$, then there is a cycle $e \xrightarrow{\mathsf{ob}} d \xrightarrow{\mathsf{ob}} c \xrightarrow{\mathsf{ob}} c' \xrightarrow{\mathsf{ob}} e$.

So we can assume that $e' = e, d' = d$ and
$$e \xrightarrow{\mathsf{eco}} d \xrightarrow{\mathsf{ob}} c \xrightarrow{\mathsf{ob}} c' \xrightarrow{\mathsf{eco}} e$$
where all of e, d, c, c' access the same location and at least one of e, d is a write, at least one of e, c' is a write, and at least one of d, c is a write.

We reason by cases.

- If c' is a write or both (e, d) are writes.
We deduce that $d \xrightarrow{\mathsf{eco}} c' \xrightarrow{\mathsf{eco}} e$ and thus $d \xrightarrow{\mathsf{eco}} e$.

- c' is a read. e is a write. d is a read.
  In this case c is a write. From $c \xrightarrow{\text{ob}} e$, we deduce $c \xrightarrow{\text{eco}} e$. Combining with $d \xrightarrow{\text{eco}} c$, we deduce that $d \xrightarrow{\text{eco}} e$.

In either case, there is a contradiction $e \xrightarrow{\text{eco}} d \xrightarrow{\text{eco}} e$. □

Lemma C.3. *$(\text{ob} \cup \text{eco})^*$ is irrreflexive.*

Proof. The simple case that ob; eco is irreflexive is proved above. The full proof by contradiction. Let $n \geq 1$ be the minimum such that:

$$\begin{array}{r}
e_0^0 \xrightarrow{\text{ob}} e_1^0 \xrightarrow{\text{eco}} d_0^0 \xrightarrow{\text{ob}} d_1^0 \\
(\xrightarrow{\text{eco}} \cap \xrightarrow{\text{ob}})\ e_0^1 \xrightarrow{\text{ob}} e_1^1 \xrightarrow{\text{eco}} d_0^1 \xrightarrow{\text{ob}} d_1^1 \\
(\xrightarrow{\text{eco}} \cap \xrightarrow{\text{ob}})\ \ldots \\
\ldots d_1^n \\
(\xrightarrow{\text{eco}} \cap \xrightarrow{\text{ob}})\ e_0^0
\end{array}$$

where for all i, we have:

$$e_0^i \xrightarrow{\text{po}} e_1^i (\xrightarrow{\text{eco}} \cap \xrightarrow{\text{po}}) d_0^i \xrightarrow{\text{po}} d_1^i$$

and

$$\neg(d_1^i \xrightarrow{\text{po}} (e_0^{(i+1) \mod n}$$

For any i, if $e_0^i \neq e_1^i$ or $d_0^i \xrightarrow{\text{po}} d_1^i$, via lemma 5.2, we deduce that $e_0^i \xrightarrow{\text{ob}} d_1^i$, contradicting minimality of n.

So, we can assume that $n \geq 1$ is such that:

$$\begin{array}{r}
e^0 \xrightarrow{\text{eco}} d^0 \\
(\xrightarrow{\text{eco}} \cap \xrightarrow{\text{ob}})\ e^1 \xrightarrow{\text{eco}} d^1 \\
(\xrightarrow{\text{eco}} \cap \xrightarrow{\text{ob}})\ \ldots \\
\ldots d^n \\
(\xrightarrow{\text{eco}} \cap \xrightarrow{\text{ob}})\ e^0
\end{array}$$

which is a contradiction since it is a cycle in $\xrightarrow{\text{eco}}$ and since at least one of e^i, d^i is a write for all i. □

D CAUSALITY TEST CASES

In this section, we discuss three of the causality test cases and the thread inlining from [Manson et al. 2005]. In presenting the examples, we unroll loops, correct typos and simplify the code.

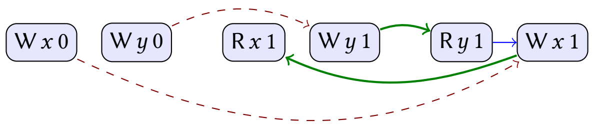
D.1 Causality test case 8

Test case 8 asks whether:

$$x\texttt{:=}0;\ y\texttt{:=}0;\ (\texttt{if}(x < 2)\{y\texttt{:=}1\} \parallel x\texttt{:=}y)$$

may read 1 for both x and y. This behavior is allowed, since “interthread analysis could determine that x and y are always either 0 or 1.” This breaks the dependency between the read of x and the write to y in the first thread, allowing the write to be moved earlier.

The semantics of TC8 includes



Where we require (W x 0) < (R x 1) but not (R x 1) < (W y 1). To see why this execution exists, consider the left thread with syntax sugar removed:

$$r := x;\ \texttt{if}(r < 2)\{y := 1\}$$

$[\![\texttt{if}(r < 2)\{y := 1\}]\!]$ includes $(r < 2 \mid \mathrm{W}\,y\,1)$. Thus, by the definition of prefixing read, $[\![r := x;\ \texttt{if}\ (r < 2)\{y := 1\}]\!]$ includes $(\mathrm{R}\,x\,1) \Rightarrow (r < 2 \mid \mathrm{W}\,y\,1)[x/r]$ which simplifies to $(\mathrm{R}\,x\,1) \Rightarrow (x < 2 \mid \mathrm{W}\,y\,1)$, which, by Definition 3.4, includes:

$$\boxed{\mathrm{R}\,x\,1}\quad \boxed{x < 2 \mid \mathrm{W}\,y\,1}$$

Prefixing with (W x 0) allows us to discharge the assumption $x < 2$, arriving at:

$$\boxed{\mathrm{W}\,x\,0}\quad \boxed{\mathrm{R}\,x\,1}\quad \boxed{\mathrm{W}\,y\,1}$$

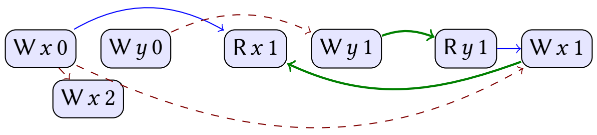
D.2 Causality test case 9

Test case 9 asks whether:

$$x := 0;\ y := 0;\ (\texttt{if}(x < 2)\{y := 1\} \parallel x := y \parallel y := 2;\ )$$

may read 1 for both x and y. This behavior is also allowed. This is “similar to test case 8, except that x is not always 0 or 1. However, a compiler might determine that the read of x by thread 1 will never see the write by thread 3 (perhaps because thread 3 will be scheduled after thread 1)”

Reasoning as for test case 8, the semantics of test case 9 includes:



Thus, with respect to the introduction of new threads, our model appears to be more robust than the event structures semantics of [Jeffrey and Riely 2016], which fails on this test case.

D.3 Causality test case 14

Test case 14 asks whether:

$$a := 0;\ b := 0;\ y := 0;\ (\texttt{if}(a)\{b := 1\}\,\texttt{else}\,\{y := 1\} \parallel \texttt{while}\,(y + b == 0)\{\texttt{skip}\}\ a := 1)$$

may read 1 for a and b, yet 0 for y. Here a and b are regular variables and y is volatile, which is equivalent to release/acquire in this example. This behavior is also disallowed, since “in all sequentially consistent executions, [the read of a gets 0] and the program is correctly synchronized. Since the program is correctly synchronized in all SC executions, no non-SC behaviors are allowed.”

Unrolling the loop once, we have:

$$a := 0;\ b := 0;\ y := 0;\ (\texttt{if}(a)\{b := 1\}\,\texttt{else}\,\{y := 1\} \parallel \texttt{if}(y \vee b)\{a := 1\})$$

We argue that any execution with (R a 1), (R b 1), and (R y 0) must be cyclic. The closure requirements require that (W a 1) $<$ (R a 1) and (R b 1) $<$ (R b 1). Ignoring initialization, least ordered execution that includes all of these actions is:

$$\begin{tikzcd}
\mathrm{R}\,a\,1 \arrow[d] & \mathrm{R}\,b\,1 \arrow[d] & \mathrm{R}\,y\,0 \\
\mathrm{W}\,b\,1 \arrow[ru] & \mathrm{W}\,a\,1 \arrow[lu] &
\end{tikzcd}$$

where the read of a is ordering for (W b 1) but not (W y 1), and the read of b is ordering for (W a 1) but the read of y is not. (W y 1) is crossed out, since its precondition must imply $(\neg a)[1/a]$, which is equivalent to false. To avoid order from (R y 0) to (W a 1), we have strengthened the predicate on (W a 1) from $(y \vee b)$ to $(y = 0 \wedge b = 1)$. Note that we cannot use this trick symmetrically to remove the order from (R b 1) to (W a 1), since $b = 1$ does not follow from the initialization of b.

D.4 Thread inlining

One property one could ask of a model of shared memory is thread inlining: any execution of $[\![P\,;\,Q]\!]$ is an execution of $[\![P \parallel Q]\!]$. This is *not* a goal of our model, and indeed is not satisfied, due to the different semantics of concurrent and sequential memory accesses. We demonstrate this by considering an example from the Java Memory Model [Manson et al. 2005], which shows that Java does not satisfy thread inlining either.

The lack of thread inlining is related to the different dependency relations introduced by sequential and concurrent access. The program $(x := 0;\ y := x + 1)$ has execution:

$$\begin{tikzcd}
\mathrm{W}\,x\,0 & \mathrm{W}\,y\,1
\end{tikzcd}$$

but $(x := 1 \parallel y := x + 1)$ has:

$$\begin{tikzcd}
\mathrm{W}\,x\,1 \arrow[r] & \mathrm{R}\,x\,1 \arrow[r] & \mathrm{W}\,y\,2
\end{tikzcd}$$

That is, in the sequential case there is no dependency from the write of x to the write of y, but in the concurrent case there is such a dependency.

This can be used to construct a counter-example to thread inlining, based on [Manson et al. 2005, Ex 11]:

$$(x := 0;\ \mathtt{if}(x)\{z := 1\}\,\mathtt{else}\,\{x := 1\}) \parallel (y := x) \parallel (x := y)$$

This has no execution containing (W z 1). Any attempt to build such an execution results in a cycle:

$$\begin{tikzcd}
\mathrm{R}\,x\,1 \arrow[r] & \mathrm{W}\,z\,1 & & & \\
 & \mathrm{R}\,x\,1 \arrow[r] & \mathrm{W}\,y\,1 \arrow[r] & \mathrm{R}\,y\,1 \arrow[r] & \mathrm{W}\,x\,1 \arrow[llllu, bend right] \arrow[lll, bend right]
\end{tikzcd}$$

Inlining the thread `(y := x)` gives [Manson et al. 2005, Ex 12]:

$$(x := 0;\ \mathtt{if}(x)\{z := 1\}\,\mathtt{else}\,\{x := 1\};\ y := x) \parallel (x := y)$$

with execution:

$$\begin{tikzcd}
\mathrm{R}\,x\,1 \arrow[r] & \mathrm{W}\,y\,1 \arrow[r] & \mathrm{R}\,y\,1 \arrow[r] & \mathrm{W}\,x\,1 \arrow[lll, bend right]
\end{tikzcd}$$

To see why this execution exists, consider the program fragment:

$$\mathtt{if}(x)\{z := 1\}\,\mathtt{else}\,\{x := 1\};\ y := x$$

Desugaring, we have:

$$r_1 := x;\ \mathtt{if}(r_1)\{z := 1;\ r_2 := x;\ y := r_2\}\,\mathtt{else}\,\{x := 1;\ r_3 := x;\ y := r_3\}$$

Now, $[\![z := 1;\, r_2 := x;\, y := r_2]\!]$ includes pomset:

$$\boxed{r_1 = 1 \mid \mathsf{W}\,z\,1} \quad \boxed{r_1 = x = 1 \mid \mathsf{W}\,y\,1}$$

and $[\![x := 1;\, r_3 := x;\, y := r_3]\!]$ includes pomset:

$$\boxed{r_1 \neq 1 \mid \mathsf{W}\,x\,1} \quad \boxed{r_1 \neq 1 \mid \mathsf{W}\,y\,1}$$

so $[\![\texttt{if}(r_1 = 1)\{z := 1;\, r_2 := x;\, y := r_2\}\,\texttt{else}\,\{x := 1;\, r_3 := x;\, y := r_3\}]\!]$ includes:

$$\boxed{r_1 = 1 \mid \mathsf{W}\,z\,1} \quad \boxed{r_1 \neq 1 \mid \mathsf{W}\,x\,1}$$
$$\boxed{(r_1 = x = 1) \lor (r_1 \neq 1) \mid \mathsf{W}\,y\,1}$$

which means $[\![\texttt{if}(r_1 = 1)\{z := 1;\, r_2 := x;\, y := r_2\}\,\texttt{else}\,\{x := 1;\, r_3 := x;\, y := r_3\}]\!][x/r_1]$ includes:

$$\boxed{x = 1 \mid \mathsf{W}\,z\,1} \quad \boxed{x \neq 1 \mid \mathsf{W}\,x\,1}$$
$$\boxed{(x = x = 1) \lor (x \neq 1)) \mid \mathsf{W}\,y\,1}$$

Now $(x = x = 1) \lor (x \neq 1)$ is a tautology, so this is just:

$$\boxed{x = 1 \mid \mathsf{W}\,z\,1} \quad \boxed{x \neq 1 \mid \mathsf{W}\,x\,1} \quad \boxed{\mathsf{W}\,y\,1}$$

and so $[\![r_1 {:=} x;\, \texttt{if}(r_1 = 1)\{z := 1;\, r_2 := x;\, y := r_2\}\,\texttt{else}\,\{x := 1;\, r_3 := x;\, y := r_3\}]\!]$ includes:

$$\begin{tikzcd}
\boxed{\mathsf{R}\,x\,1} \arrow[r] \arrow[rr, bend left] & \boxed{1 = 1 \mid \mathsf{W}\,z\,1} & \boxed{1 \neq 1 \mid \mathsf{W}\,x\,1} & \boxed{\mathsf{W}\,y\,1}
\end{tikzcd}$$

which simplifies to:

$$\begin{tikzcd}
\boxed{\mathsf{R}\,x\,1} \arrow[r] & \boxed{\mathsf{W}\,z\,1} & \boxed{\mathsf{W}\,y\,1}
\end{tikzcd}$$

as required. The rest of the example is straightforward, and shows that our semantics agrees with the JMM in not supporting thread inlining.