Pomsets with Preconditions

A Simple Model of Relaxed Memory

RADHA JAGADEESAN, DePaul University
ALAN JEFFREY, Mozilla Research
JAMES RIELY, DePaul University

Relaxed memory models must simultaneously achieve efficient implementability and thread-compositional reasoning. Is that why they have become so complicated? We argue that the answer is no: It is possible to achieve these goals by combining an idea from the 60s (preconditions) with an idea from the 80s (pomsets), at least for x64 and ARMv8. We show that the resulting model (1) supports compositional reasoning for temporal safety properties, (2) supports all reasonable sequential compiler optimizations, (3) satisfies the DRF-SC criterion, and (4) compiles to x64 and ARMv8 microprocessors without requiring extra fences on relaxed accesses.

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

Additional Key Words and Phrases: Relaxed Memory Models, Pomsets, Preconditions, Hoare Logic, Temporal Safety Properties, Compiler Optimizations, Thin-Air Reads

ACM Reference Format:
Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with Preconditions: A Simple Model of Relaxed Memory. 1, 1 (September 2020), 30 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."

There are two aspects of "implementation flexibility." First, relaxed atomic access should not require hardware synchronization. Second, the model should facilitate compiler transformations, such as the reordering of independent statements; ideally the model should support all valid optimizations of synchronization-free single-threaded code.

There are also two aspects to "ease of use." First, 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. Second, all programs—even those with data races—should support compositional reasoning on temporal safety properties [Abadi and Lamport 1993; Misra and Chandy 1981; Pnueli 1985; Stark 1985].

Sailing between this Scylla and Charybdis has proven very difficult. Three lines of code can leave the top experts in the field flabbergasted. The solutions that have been proposed are understandable to mechanical proof assistants, but humans have been left behind.

In this paper, we combine two ideas that humans can understand: *preconditions* [Hoare 1969] and *labelled partial orders* (aka *pomsets*) [Gischer 1988; Plotkin and Pratt 1997]. The resulting model

This paper has been greatly improved by the comments of the anonymous reviewers.
This material is based upon work supported by the National Science Foundation under Grant No. CCR-1617175. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.
Authors' addresses: Radha Jagadeesan, DePaul University; Alan Jeffrey, Mozilla Research; James Riely, DePaul University.

XXXX-XXXX/2020/9-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

mostly satisfies the desiderata. We sacrifice only implementability on "non-MCA" processors, such as POWER and ARMV7. As a result, however, there is only one order relation to visualize.

Perhaps you believe the problem has already been solved? Let us try to convince you otherwise.

To get a sense of the problems involved, consider that the existence of an execution in a relaxed memory model may depend on code that was *not* executed. Let r and s be registers and x-z be shared memory locations. Consider:

$$y{:=}x \;\parallel\; r{:=}y;\; \texttt{if}(r)\{x{:=}r;\; z{:=}r\}\,\texttt{else}\,\{x{:=}1\} \tag{$*$}$$

Most programmers would be surprised to learn that this program does allow an execution in which $z = 1$. To see why, imagine that a compiler does type inference and finds that x and y are booleans, with value either 0 or 1. This enables the program to be optimized to the following:

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

Since $x{:=}1$ occurs in both branches of the conditional, the compiler can then lift it, and reorder with the independent read of y, yielding:

$$y{:=}x \;\parallel\; x{:=}1;\; r{:=}y;\; \texttt{if}(r)\{z{:=}1\}$$

Then $z = 1$ at then end of an execution where the first thread is interleaved immediately after executing $x{:=}1$. Compare ($*$) with:

$$y{:=}x \;\parallel\; r{:=}y;\; \texttt{if}(r)\{x{:=}r;\; z{:=}r\}\,\texttt{else}\,\{x{:=}2\} \tag{$\dagger$}$$

which has a similar execution where $z = 2$, but does not have an execution where $z = 1$. Note that the only difference between ($*$) and ($\dagger$) is the code in the branch which was not taken. This means that any model of relaxed memory that supports common compiler optimizations (such as the second transformation above) must take into account code that was not executed. This accounts for the speculative executions common in relaxed memory models.

Pugh [1999, §2.3] initiated the modern study of relaxed memory by noting that Java 1.1 failed to validate Common Subexpression Elimination (CSE) in the presence of aliasing. For example, given that $r_2 \neq s$, is it valid to transform the program on the left to that on the right?

$$(r_1{:=}x;\; s{:=}y;\; r_2{:=}x;\; C) \qquad\qquad (r_1{:=}x;\; r_2{:=}r_1;\; s{:=}y;\; C) \tag{CSE}$$

The resulting Java Memory Model (JMM) [Manson et al. 2005] greatly advanced the state of the art.

Lochbihler's [2013] monumental study of the JMM revealed a surprising limitation: The following program "is type correct if it declares x, y and r of type D. However, it has a legal execution where they reference a C object" [Lochbihler 2013, Fig. 8]:

$$z{:=}1 \;\parallel\; y{:=}x \;\parallel\; r{:=}y;\; \texttt{if}(z)\{s{:=}\texttt{new C}\}\,\texttt{else}\,\{r{:=}\texttt{new D}\};\; x{:=}r \tag{OOTA1}$$

Informally, all threads satisfy the invariant "allocation at type C is preceded by reading 1 for z" and "allocation at type D is preceded by reading 0 for z." If composability of safety were to hold, the full program would satisfy both invariants. The lack of composability forced Lochbihler to partition memory by type in order to prove type safety. This formal device means that memory cannot be used at different types over time, making practical memory reclamation impossible.

As confirmed by [Chakraborty and Vafeiadis 2018; Kang et al. 2018], the promising semantics [Kang et al. 2017] and related models [Chakraborty and Vafeiadis 2019; Jagadeesan et al. 2010] all invalidate compositional reasoning, as in OOTA1. This means that these models *cannot support both type safety and realistic memory reclamation.*

The C11 Memory Model [Batty et al. 2011] does not attempt to validate CSE, at least not for relaxed atomic access (consider the case where x and y are aliased above). C11 *does* allow the transformation for *plain* access, but this comes with the threat of *undefined behavior* should any

plain access ever possibly engage in a data race. Thus the folklore belief that "every substantial C program has undefined behavior."

Strong models, including Sequential Consistency (SC) [Lamport 1979], RC11 [Lahav et al. 2017], and others [Boehm and Demsky 2014; Dolan et al. 2018; Jeffrey and Riely 2016; Lahav et al. 2017] support compositional reasoning. However, all of these models invalidate reordering of independent statements. Several require fences after relaxed reads, even on ARMV8.

In our approach, a program is a set of execution. Each execution is a partial order on a set of read and write events. The order is intended to be read as a "dependency" relation. The dependency relation can vary between executions, so it is dynamic; events that are not related in an execution are "independent" and can be seen by a sequential observer in either order.

The executions of a program are computed compositionally, by induction on the structure of the program. Thus, our model falls into the style of relaxed memory models advocated by Batty [2017].

Cross thread dependencies arise from conflicting actions on the same variable. Thus, any two actions, at least one of which is a write, on the same location and not in the same thread, are ordered. In the parlance of hardware memory models [Alglave et al. 2014], coe, fre, wr are included in the dependency ordering. Thus, our model realizes *multi-copy atomicity* (MCA); when a write becomes visible to one thread it must become visible to all [Pulte et al. 2018].

Within a thread, the dependency calculation can be viewed as the computation of *preserved* program order (ppo) in hardware models. ppo captures the *essential dependencies* between events in the same thread. Our key insight is to that *logic is better than syntax* to capture such dependencies.

Consider the following program fragments:

$$C_1 : x{:=}1;\ y{:=}1$$
$$C_2 : r{:=}x;\ \texttt{if}(r)\{\ y{:=}1\}\ \texttt{else}\ \{y{:=}1\}$$
$$C_3 : x{:=}1;\ r{:=}x;\ \texttt{if}(r)\{y{:=}1\}$$

All these fragments satisfy $\{\text{true}\}\ C_i\ \{y = 1\}$; thus, in each case, the write of y is independent of any code that precedes it in program order. While C_1 reflects syntactic independence, C_2 reflects the independence derived by case analysis, and C_3 reflects the independence deduced from partial evaluation (in the restricted form of constant propagation).

C_3 shows the important role of *internal* reads—reads that are fulfilled by a preceding write in the same thread—in relaxed memory models such as ARMV8. Unlike externally fulfilled reads, internal reads are not necessarily recorded in the dependency relation. This allows a compiler or processor to reorder the write with respect to the code that precedes it.

The logical perspective provides a clear intuition why our model validates the compiler transformations discussed above (∗).

Thread compositionality provides a perspective on how our model forbids "Out Of Thin Air" (OOTA); after all, it is stating that parallel composition does not create unexpected new behaviors. More concretely, as envisioned in [Alglave 2010, §3.3], MCA *permits a single, global notion of time, manifest in the dependency order* that captures all cross-thread dependencies. The absence of cycles in the dependency order forbids OOTA behaviors. Consider the well-known litmus test:

$$x{:=}0;\ y{:=}0;\ (y{:=}x\ \parallel\ x{:=}y)$$

We rule out an assignment of 1 to both locations as follows. The write of x in the LHS is dependent on the read to y. Similarly, the write of y in the RHS is dependent on the read to x. Since a read of 1 can only be satisfied externally, both reads are part of the dependency order, leading to a cycle in the dependency order: a contradiction.

We show that the model:

- validates expected litmus cases and compiler optimizations (§3-4).

- captures all C11 concurrency features (§5),
- allows compositional reasoning for safety (§6),
- compiles to ARMv8/TSO *without* extra synchronization for relaxed-atomic access (§7), and
- satisfies the *local* DRF-SC criterion [Dolan et al. 2018] (§8).

We conclude by discussing relating work (§9) and limitations (§10).

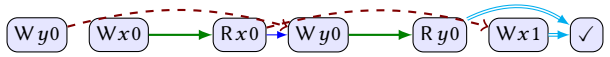
2 THE BASIC MODEL

The model adapts our previous work on microarchitecture [Disselkoen et al. 2019] to the architectural level. In this section, we define the model and use it to give the semantics of a concurrent language. The semantics given here is slightly simplified. As discussed in §3, it fails to validate some important optimizations. We give the full semantics in §4. In §5, we define extensions that incorporate address computation, fences, and read-modify-write operations.

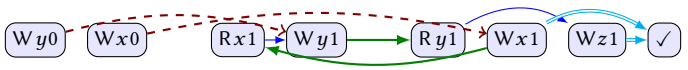
The model is based on *partially ordered multisets* [Gischer 1988; Plotkin and Pratt 1997], where events are labelled with reads and writes, and the partial order tracks control and data dependencies, coherence, concurrent reads and writes using shared memory, and fencing. Here, we are only using one order, which stands in stark contrast with other models that use many different relations.

For example the semantics of (∗) contains the expected pomset (where ✓ indicates termination):

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



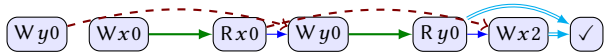
but also the unexpected one:



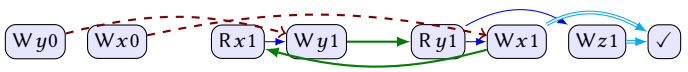
(‡)

The interesting fact about these pomsets is that there is no control dependency between reading y and writing x, since the $(\mathrm{W}x1)$ event happens on both sides of the conditional. Compare this with (†), where there is a control dependency:

$$y{:=}x \parallel r{:=}y\,;\ \texttt{if}(r)\{x{:=}r\,;\ z{:=}r\}\,\texttt{else}\,\{x{:=}2\}$$



An attempt to replicate the unexpected execution of (∗) fails, since it introduces a cycle:



We stress that in these diagrams, color plays no formal role. It is purely to help the reader see where the order comes from:

- $(\mathrm{W}x0) \dashrightarrow (\mathrm{W}x1)$ is a *coherence* requirement: the write of 1 must follow the write of 0, since these are to the same location.
- $(\mathrm{W}x1) \longrightarrow (\mathrm{R}x1)$ is a *reads-from* requirement: the read of x is fulfilled by a matching write from another thread.
- $(\mathrm{R}x1) \longrightarrow (\mathrm{W}y1)$ is a *dependency* requirement: the write to y *depends on* the read of x.
- $(\mathrm{W}^{\mathrm{ra}}z1) \Longrightarrow (\checkmark)$ is a *fencing* requirement (in this case the fence is at the end of the program, but concurrency controls introduce fencing).

2.1 Data models

A *data model* consists of:

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

Let σ range over substitutions of the form $[x/r]$ or $[N/x]$.
We require that data models satisfy the following:

- values, registers, and memory locations are disjoint,
- values include at least the constants 0 and 1,
- expressions include at least registers and values,
- expressions do *not* include memory locations,
- formulae include at least equalities $(M = v)$,
- formulae are closed under negation, conjunction, disjunction, and substitution[1], and
- there is a relation $\vDash$ between formulae, with the expected semantics.

For the actions of a data model, we require that there are partial functions Rd and $\mathsf{Wr} : \mathcal{A} \to (\mathcal{X} \times \mathcal{V})$, and there are subsets of $\mathcal{A}$: Acq, Rel, SC, and Term.

- We say that action a is a *read* if $a \in \mathrm{dom}(\mathsf{Rd})$ and a is a *write* if $a \in \mathrm{dom}(\mathsf{Wr})$. When $\mathsf{Rd}(a) = (x, v)$, we say that a *reads* v *from* x, and similarly for writes. We say that a *accesses* x if it reads or writes x.
- Actions in Acq, Rel and SC, are *synchronization* and *fencing* actions. We say that a is an *acquire* if $a \in \mathsf{Acq}$, a is a *release* if $a \in \mathsf{Rel}$, and a is *SC* if $a \in \mathsf{SC}$. We require that every SC read is an acquire, and every SC write is a release.
- Actions in Term are *termination* actions. We require that termination events are releasing.

Our example language includes actions of the form $(\checkmark)$, which is a *termination*, $(\mathsf{R}^{\mu} x v)$, which *reads* v from x and $(\mathsf{W}^{\mu} x v)$, which *writes* v to x. The *access mode* ($\mu ::= \mathsf{rlx} \mid \mathsf{ra} \mid \mathsf{sc}$) is either *relaxed*, *release-acquire*, or *sequentially-consistent*. ra/sc reads are acquires, and ra/sc writes are releases. We systematically elide the rlx-mode annotation, writing $(\mathsf{R}\, x v)$ as shorthand for $(\mathsf{R}^{\mathsf{rlx}} x v)$.
We do not explicitly include C11-style *plain* access. Plain access is the same as relaxed access for race-free programs; for racy programs, plain access results in undefined behavior.

Formulae. Logical formulae include equations over locations and registers, such as $(x{=}1)$ and $(r{=}s{+}1)$. Formulae are *open*, in that occurrences of register names and memory locations are subject to substitutions of the form $\phi[x/r]$ and $\phi[N/x]$. Actions are not subject to substitution.
We use expressions as formulae, coercing M to $M \neq 0$.

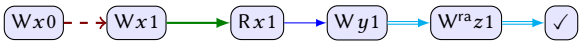
Definition 2.1. We say ϕ is *independent of* x if, for every v, $\phi \vDash \phi[v/x] \vDash \phi$; it is *dependent* otherwise. We say ϕ is *location independent* if it is independent of every location. We say ϕ *implies* ψ if $\phi \vDash \psi$. We say ϕ is a *tautology* if $\mathsf{true} \vDash \phi$. We say ϕ is *unsatisfiable* if $\phi \vDash \mathsf{false}$.

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

2.2 Semantic domain

We model single executions as *pomsets with preconditions*—*pomsets*, for short—ranged over by P. These extend the well-known model of partially ordered multisets [Gischer 1988] with formulae and a termination event.

The pomset order relation, $\leq$, represents *causality*. We visualize pomsets as directed graphs. For example, the semantics of $\mathtt{var}\, x;\ (x{:=}0;\ x{:=}1 \parallel y{:=}x;\ z^{\mathsf{ra}}{:=}1)$ includes:



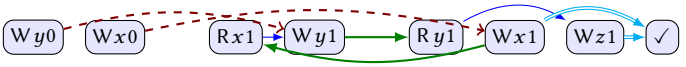
We visualize order using arrows that indicate the reason that the order arises. $(\mathrm{W}x0) \dashrightarrow (\mathrm{W}x1)$ is a *coherence* requirement: the write of 1 must follow the write of 0, since these are in *conflict* and in program order. $(\mathrm{W}x1) \rightarrow (\mathrm{R}x1)$ is a *reads-from* requirement: the read of x must be *fulfilled* by a matching write. $(\mathrm{R}x1) \rightarrow (\mathrm{W}y1)$ is a *dependency* requirement: the write to y *depends on* the read of x. $(\mathrm{W}y1) \Rightarrow (\mathrm{W}^{\mathsf{ra}}z1)$ and $(\mathrm{W}^{\mathsf{ra}}z1) \Rightarrow (\checkmark)$ are *fencing* requirements.

A pomset is *completed* if it contains a unique termination action, ordered after all other events. Henceforth, we will elide this uninteresting termination event in drawings.

In addition to actions, each pomset event is labeled with a *precondition*. Whereas read actions represent an obligation that must be *fulfilled* concurrently by a matching write (Definition 2.5), preconditions represent an obligation that must be *satisfied* sequentially via substitution.

To get a sense of how preconditions are satisfied, let us consider the evolution of the precondition of $(\mathrm{W}x1)$ during the calculation of (‡):

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



First consider the else-branch of the conditional: the semantics of "$\mathtt{if}(\neg r)\{x{:=}1\}$" contains $(r{=}0 \mid \mathrm{W}x1)$, indicating the control dependency. The then-branch is more complex: the semantics of "$\mathtt{if}(r)\{x{:=}r\}$" contains $(r{\neq}0 \wedge r{=}1 \mid \mathrm{W}x1)$ indicating both a control and a data dependency. This can be simplified to $(r{=}1 \mid \mathrm{W}x1)$. Combining the two branches of the conditional, we have $(r{=}0 \vee r{=}1 \mid \mathrm{W}x1)$. Prepending $r{:=}y$ substitutes $[y/r]$, resulting in $(y{=}0 \vee y{=}1 \mid \mathrm{W}x1)$. Prepending the initializer $y{:=}0$ substitutes $[0/y]$, resulting in $(0{=}0 \vee 0{=}1 \mid \mathrm{W}x1)$. This is a tautology, which we write as $(\mathrm{W}x1)$. We repeat this calculation in §2.7, after giving the formal definitions.

Each pomset represents a single execution. Thus we require that all preconditions are *consistent*. For example, the semantics of $\mathtt{if}(r < 0)\{y{:=}1\}\,\mathtt{else}\,\{z{:=}1\}$ includes pomsets with either $(r < 0 \mid \mathrm{W}y1)$ or $(r \geq 0 \mid \mathrm{W}z1)$, but not with both, since $(r < 0 \wedge r \geq 0)$ is unsatisfiable.

Preconditions are linked to pomset order via *causal strengthening*, which requires that formulae do not weaken over time, as measured by $\leq$. We give an example that requires causal strengthening on page 19. Note that, by causal strengthening, the precondition of the termination event of a completed pomset must imply the preconditions of all other events.

Definition 2.2. A *pomset with preconditions* is a tuple $(E, \leq, \lambda)$:

- E is a set of *states*,
- $\leq \subseteq (E \times E)$ is a partial order,
- $\lambda : E \to (\Phi \times \mathcal{A})$ is a *labeling*, from which we derive functions $\Phi : E \to \Phi$ and $\mathcal{A} : E \to \mathcal{A}$,
- $\bigwedge_e \Phi(e)$ is satisfiable *(consistency)*, and
- if $d \leq e$ then $\Phi(e)$ implies $\Phi(d)$ *(causal strengthening)*, and

We write pairs in $(\Phi \times \mathcal{A})$ as $(\phi \mid a)$, eliding ϕ when it is a tautology. We write $d < e$ when $d \leq e$ and $d \neq e$. We often elide explicit universal quantifiers in phrases such as "*for all* e, $\Phi'(e)$ implies

$\Phi(e)$." We lift terminology and notation from actions and formulae to events. For example, we may say that e is a read when $\mathcal{A}(e)$ is a read.

The notion of *downset* for pomsets is similar to the notion of *prefix* for strings:

Definition 2.3. P' is a *downset* of P if $E \supseteq E' \supseteq \{d \in E \mid \exists e \in E'.\ d \leq e\}$, $\leq' = \leq|_{E'}$, and $\lambda' = \lambda|_{E'}$.

We say that $\mathcal{P}$ is *downset-closed* if $P \in \mathcal{P}$ and P' is a downset of P implies $P' \in \mathcal{P}$.

The semantics of programs is given as sets of completed pomsets that are closed with respect to order *augmentation*, which may add order, and *implication*, which may have stronger formulae. In examples, we draw pomsets that are *augmentation-minimal* and *implication-minimal*.

Definition 2.4. We say that P' is an *augment* of P if $E' = E$, $\lambda' = \lambda$, and $\leq' \supseteq \leq$.
We say that P' *implies* P if $E' = E$, $\leq' = \leq$, $\mathcal{A}' = \mathcal{A}$, and $\Phi'(e)$ implies $\Phi(e)$.

2.3 Example Language

We define the language by prefixing individual reads and writes.

$$C, D ::= \texttt{skip} \mid r := M; C \mid r := x^{\mu}; C \mid x^{\mu} := M; C \\ \mid C \parallel D \mid \texttt{var}\, x; C \mid \texttt{if}(M)\{C\}\texttt{else}\{D\}$$

We use common syntax sugar, such as *extended expressions*, $\mathbb{M}$, which include memory locations. For example, if $\mathbb{M}$ 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 separate read.

We write $\texttt{if}(M)\{C\}$ as shorthand for $\texttt{if}(M)\{C\}\texttt{else}\{\texttt{skip}\}$ and $\texttt{if}(M)\{C^1\}\texttt{else}\{C^2\}; D$ as shorthand for $\texttt{if}(M)\{C^1; D\}\texttt{else}\{C^2; D\}$.

The semantic function $[\![-]\!]$ takes a command and yields a set of pomsets, ranged over by $\mathcal{P}$.

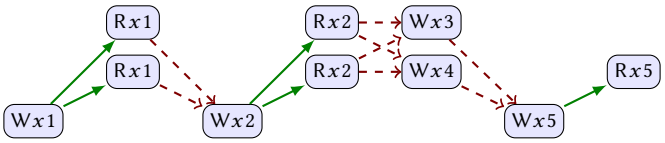
2.4 Local Declarations

At the point that x is bound, we can require that every read of x be *fulfilled*. Fulfillment plays the role that *reads-from* and *coherence* play in other relaxed memory models, yet it is not the same.

Definition 2.5. Two actions *conflict* if one writes a location and the other either reads or writes the same location. We say d *fulfills* e *(on* x*)* if (F1) d writes v to x, (F2) e reads v from x, (F3) $d < e$, and (F4) for every conflicting write c, either $c \leq d$ or $e \leq c$.

Item F3 requires that a write d is ordered before any read e it fulfills; this order is typically called *reads from*. Item F4 requires that any conflicting write c is ordered before d or after e; this order is typically called *extended coherence*. For readability, we draw the order required by F3 using bold green arrows and the order required by F4 using dashed red arrows. As an example, consider:

$$x := 1 \parallel x := 2 \parallel x := 3 \parallel x := 4 \parallel x := 5 \parallel r := x; r := x; r := x; r := x \qquad \text{(Co1)}$$



A write is *relevant* if it is read from. In order to fulfill all of the reads on x in the example, we pick a total order on the relevant writes: in this case, $(\mathrm{W}x1) \leq (\mathrm{W}x2) \leq (\mathrm{W}x5)$. The reads slot between these, immediately after their fulfilling write. Reads are not necessarily ordered with respect to each other, even if they come from the same thread, as do the reads here. Irrelevant writes also float relative to each other, as do $(\mathrm{W}x3)$ and $(\mathrm{W}x4)$. But irrelevant writes must be ordered with respect to relevant writes and reads. The resulting order is somewhat weaker than traditional extended

coherence, which requires a total order on the writes, regardless of whether they are relevant. We discuss coherence further on page 12.

Definition 2.6. A pomset is *x-closed* if every read on x is fulfilled, and every formula is independent of x ($\forall v.\ \phi \vDash \phi[v/x] \vDash \phi$). Let $(vx.\mathcal{P})$ be $\mathcal{P}' \subseteq \mathcal{P}$ such that $P' \in \mathcal{P}'$ when P' is x-closed. We define:

$$[\![\mathsf{var}\ x;\ C]\!] \triangleq vx\ .\ [\![C]\!]$$

A pomset is *top-level* if it is x-closed for every location x.

2.5 Composition

Parallel composition is roughly pomset union, allowing that some events may *coalesce*, with the resulting precondition being the disjunction of the precondition taken from the two sides. Composition is used to define conditionals (as in [Disselkoen et al. 2019]). We also use it to define address calculation (§5).

Because of consistency (Definition 2.2), we do not include events with contradictory preconditions. To make coalescing more explicit in the following examples, we show the element of the carrier set E to left of its label. Consider:

$\mathtt{if}(r < 0)\{y := 1\}$ $\qquad$ $\mathtt{if}(r \geq 0)\{y := 1\}$

e: $\boxed{r < 0 \mid \mathrm{W}y1}$ $\qquad$ e: $\boxed{r \geq 0 \mid \mathrm{W}y1}$

The parallel composition includes pomsets with either one of the two events, but not both. However, events with the same label may coalesce, taking the disjunction of their preconditions. Thus, the semantics of the combined program also includes $e{:}(r < 0 \vee r \geq 0 \mid \mathrm{W}y1)$. Coalesced events inherit order from both sides.

Definition 2.7. Let $P' \in (\mathcal{P}^1 \parallel \mathcal{P}^2)$ when there are $P^1 \in \mathcal{P}^1$ and $P^2 \in \mathcal{P}^2$ such that $E' = E^1 \cup E^2$, E^1 is completed exactly when E^2 is completed, there is at most one termination in E', $\leq' \supseteq \leq^1 \cup \leq^2$, and for all $e \in E'$, either:

$$\begin{array}{c} \mathcal{A}'(e) = \mathcal{A}^1(e) = \mathcal{A}^2(e) \text{ and } \Phi'(e) \text{ implies } \Phi^1(e) \vee \Phi^2(e), \\ e \notin E^2,\ \mathcal{A}'(e) = \mathcal{A}^1(e) \text{ and } \Phi'(e) \text{ implies } \Phi^1(e), \quad \text{or} \\ e \notin E^1,\ \mathcal{A}'(e) = \mathcal{A}^2(e) \text{ and } \Phi'(e) \text{ implies } \Phi^2(e). \end{array}$$

We then define:

$$[\![C \ \|\ D]\!] \triangleq [\![C]\!] \parallel [\![D]\!]$$

The definition requires that if $P' \in (P^1 \parallel P^2)$ is completed, then both P^1 and P^2 are completed, and further, the termination events *must* coalesce in P'.

2.6 Conditional, Register Assignment, and Skip

Conditional execution is defined using parallel composition and *filtering*: $(\phi \rhd \mathcal{P})$ selects the subset of pomsets in $\mathcal{P}$ that are implied by ϕ. Register assignment is defined using substitution: $(\mathcal{P}\sigma)$ performs the substitution σ on every formula in $\mathcal{P}$. The semantics of `skip` is defined using singleton pomsets with label $\checkmark$.

Definition 2.8. Let $(\phi \rhd \mathcal{P})$ be the set $\mathcal{P}' \subseteq \mathcal{P}$ such that $P' \in \mathcal{P}'$ when ϕ implies $\Phi(e')$ $(\forall e' \in E')$. Let $(\mathcal{P}\sigma)$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ when there is $P \in \mathcal{P}$ such that: $E' = E$, $\leq' = \leq$, $\mathcal{A}' = \mathcal{A}$, and $\Phi'(e) = \Phi(e)\sigma$. Let $P \in \mathsf{SKIP}$ when E has one element labelled with action $\checkmark$. We then define:

$$[\![\mathtt{if}(M)\{C\}\,\mathtt{else}\,\{D\}]\!] \triangleq (M \rhd [\![C]\!]) \parallel (\neg M \rhd [\![D]\!]) \quad [\![r := M;\ C]\!] \triangleq [\![C]\!][M/r] \quad [\![\mathtt{skip}]\!] \triangleq \mathsf{SKIP}$$

Substitution applies to formulae, not actions. For example, $(x{=}1 \mid \mathrm{W}x2)[y/x] = (y{=}1 \mid \mathrm{W}x2)$.

As an example of the conditional, consider the following fragments:

$$\texttt{if}(s)\{x:=1;\ x:=2\} \qquad\qquad \texttt{if}(\neg s)\{x:=1;\ x:=3\} \tag{**}$$

Putting these together, we can coalesce the $(\mathrm{W}x1)$ events:

$$C_{\mathsf{cond}} = \texttt{if}(s)\{x:=1;\ x:=2\}\,\texttt{else}\,\{x:=1;\ x:=3\}$$

Let us focus on the left pomset above. It is derived from the composition:

The existence of the singleton $(\neg s \mid \mathrm{W}x1)$ is guaranteed by downset closure on (**). Consistency prevents any pomset in the above $[\![C_{\mathsf{cond}}]\!]$ from containing both $(\mathrm{W}x2)$ and $(\mathrm{W}x3)$.

Note that the definitions of consistency, downset, and composition prevent the coalescing of $(\mathrm{W}y3)$ in $[\![\texttt{if}(s)\{y:=1;\ y:=3\}\,\texttt{else}\,\{y:=2;\ y:=3\}]\!]$. Any pomset that included $(\mathrm{W}y3)$ would need to contain both $(s \mid \mathrm{W}y1)$ and $(\neg s \mid \mathrm{W}y2)$, which violates consistency.

2.7 Prefixing

We present a candidate definition for prefixing. The full definition follows in §4.

Candidate 2.9. Let $\nabla\mathcal{P} = \{P' \mid P' \text{ is a downset of some } P \in \mathcal{P}\}$.

Let $(\phi \mid a) \Rightarrow \mathcal{P}$ be the set $\nabla\mathcal{P}'$ where $P' \in \mathcal{P}'$ when there is $P \in \mathcal{P}$ such that (1) $E' = E \cup \{d\}$, (2) $\leq' \supseteq \leq$, (3a) $\mathcal{A}'(d) = a$, (3b) $\mathcal{A}'(e) = \mathcal{A}(e)$,

(4a) $\Phi'(d)$ implies ϕ,
(4b) if d reads v from x then $\Phi'(e)$ implies $\Phi(e)[v/x]$,
(4c) if d does not read then $\Phi'(e)$ implies $\Phi(e)$,
(5a) if e writes then either $d <' e$ or $\Phi'(e)$ implies $\Phi(e)$,
(5b) if d and e are actions in conflict, then $d <' e$,
(5c) if d is an acquire or e is a release, then $d <' e$, and
(5d) if d is an SC write and e is an SC read, then $d <' e$.

We then define:

$$[\![r := x^{\mu};\ C]\!] \triangleq \textstyle\bigcup_v\ (\mathrm{R}^{\mu}xv) \Rightarrow [\![C]\!][x/r]$$
$$[\![x^{\mu} := M;\ C]\!] \triangleq \textstyle\bigcup_v\ (M = v \mid \mathrm{W}^{\mu}xv) \Rightarrow [\![C]\!][M/x]$$

The semantics of read and write introduce a pomset for each possible value. Note that the value in the actions is fixed. For writes, the dependence on M appears in the precondition: $M = v$.

The main work happens in the definition of prefixing ($\Rightarrow$): Item 1 introduces a new event. Item 2 ensures that no order is removed from old events. Item 3 specifies the actions labelling the events; item 4 specifies the preconditions. Item 5 specifies the preserved program order.

Whereas a write action introduces a precondition—that must be satisfied sequentially—on the value to be written; a read introduces a pomset requirement—that may be fulfilled concurrently—on the value to be read.

In the next section, we explain the concurrent semantics using standard litmus tests: Read fulfillment is constrained by the program order that is preserved by 5. Item 5a captures *read to write dependency*[2]. It only requires order from read to write when the precondition of the write is

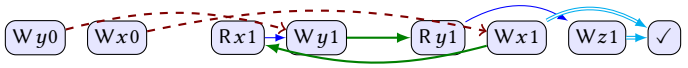
[2]When d is not a read, 4c trivially implies 5a.

weakened using 4b. Item 5b captures the extended coherence requirement on actions that touch the same location. Item 5c imposes the order required by acquire and release actions[3]. Item 5d imposes the additional order required by SC actions[4].

In this section, we explain the sequential semantics: Preconditions are satisfied using the substitutions in the semantic rules and in 4b. We also discuss 5a, which connects the concurrent and sequential semantics.

Let us revisit the unexpected execution of (∗), as explained in §2.2:

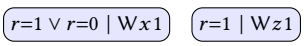
$$y{:=}0;\ x{:=}0;\ (y{:=}x \parallel r{:=}y;\ \texttt{if}(r)\{x{:=}r;\ z{:=}r\}\ \texttt{else}\ \{x{:=}1\})$$



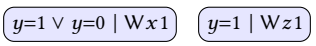
It is immediate from the definition that $[\![x{:=}r;\ z{:=}r]\!]$ contains pomset candidates such as:

Consistency (Definition 2.2) rules out the right pomset, since the conjunction of preconditions is unsatisfiable. No order is required between the writes.

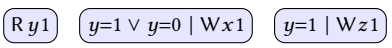
Combining the two sides of the conditional, $[\![\texttt{if}(r)\{x{:=}r;\ z{:=}r\}\ \texttt{else}\ \{x{:=}1\}]\!]$ contains:



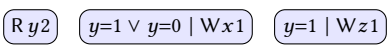
When prefixing "$r{:=}y$," we first substitute $[y/r]$, resulting in:



Adding the read action, $[\![r{:=}y;\ \texttt{if}(r)\{x{:=}r;\ z{:=}r\}\ \texttt{else}\ \{x{:=}1\}]\!]$ contains:

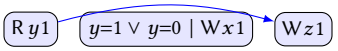


Since $(y{=}1)[2/y]$ is unsatisfiable, 4b prevents pomsets such as:

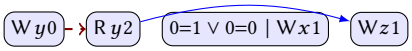


Thus the read must be consistent with the predicates.

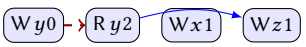
4b also allows predicates to weaken, in which case 5a requires order:



Adding the write, $[\![y{:=}0;\ r{:=}y;\ \texttt{if}(r)\{x{:=}r;\ z{:=}r\}\ \texttt{else}\ \{x{:=}1\}]\!]$ substitutes $[0/y]$, resulting in:



Since 0=1 ∨ 0=0 is a tautology, we have:



3 PROPERTIES OF THE BASIC MODEL

It is amazing how much Candidate 2.9 "gets right" out of the box, including "internal reads", which greatly complicate other models [Pulte et al. 2018]. Before refining the model, we walk through several litmus tests, valid rewrites and invalid rewrites.

[3]Recall that termination actions are releases.
[4]Recall that SC reads are acquires and SC writes are releases.

3.1 Litmus Tests

Pugh [2004] developed a set of litmus tests for the java memory model. Our model gives the expected result for all but cases 16, 19 and 20' (unrolling loops): we discuss TC16 below; TC19 and TC20 involve a thread join operation, which is not expressible in our languages. Our model also agrees with the OOTA examples in Batty et al. [2015, §4] and the "surprising and controversial behaviors" of Manson et al. [2005, §8].

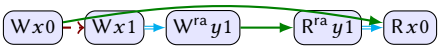
Buffering. Consider the *store buffering* and *load buffering* litmus tests:

$$x:=0;\ y:=0;\ (x:=1;\ r:=y \parallel y:=1;\ r:=x) \qquad r:=y;\ x:=1 \parallel r:=x;\ y:=1 \tag{SB/LB}$$

The desired outcomes are allowed, as shown.

Publication and SC Access. Items 5b and 5c ensure correct publication. For example, they disallow the following candidate execution, which sees a stale value for x:

$$x:=0;\ x:=1;\ y^{\mathsf{ra}}:=1 \parallel r:=y^{\mathsf{ra}};\ s:=x \tag{Pub1}$$

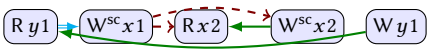


By Definition 2.5, $(\mathsf{R}x0)$ is *unfulfilled* in this pomset. It fails the last requirement of the definition, since $(\mathsf{W}x0) \le (\mathsf{W}x1) < (\mathsf{R}x0)$. In order to satisfy this requirement, $(\mathsf{R}x0)$ must be ordered before $(\mathsf{W}x1)$, but this creates a cycle.

Item 5d ensures that program order between SC operations is always preserved. Combined with the requirements for fulfillment, this is sufficient to establish that programs with only SC access have only SC executions; for example, the executions of SB/LB are banned when the actions of the two threads are all sc. It is also immediate that SC actions can be totally ordered, using any linearization of pomset order. Just as SC access in ARMv8 is simplified by MCA, it is simplified here by the global pomset order.

SC access is not as strict as volatile access in Java. For example, our model allows the following, which is disallowed by Dolan et al. [2018, §8.2]:

$$r:=y;\ x^{\mathsf{sc}}:=1;\ s:=x \parallel x^{\mathsf{sc}}:=2;\ y:=1 \tag{SC1}$$



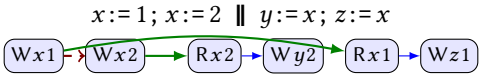
Note that there is no order from $(\mathsf{W}^{\mathsf{sc}}x2)$ to $(\mathsf{W}y1)$.

For further comparison to Java volatiles and Dolan et al. [2018], see the end of §8.

Coherence. Our model of coherence does not correspond to either Java or C11. We have chosen the model to validate CSE (unlike C11 relaxed atomics) and the local DRF-SC theorem (unlike Java).

Since reads are not ordered by 5b, we allow the following unintuitive behavior. C11 includes read-read coherence between relaxed atomics in order to forbid this:

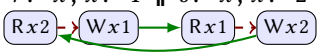
$$x:=1;\ x:=2 \parallel y:=x;\ z:=x \tag{Co2}$$



Here, the reader sees 2 then 1, although they are written in the reverse order. This behavior is allowed by Java in order to validate CSE.

However, our model is more coherent than Java, which permits the following:

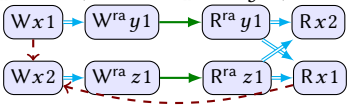
$$r:=x;\ x:=1 \parallel s:=x;\ x:=2$$



(TC16)

We also forbid the following, which Java allows:

$$x:=1;\ y^{\mathsf{ra}}:=1 \parallel x:=2;\ z^{\mathsf{ra}}:=1 \parallel r:=y^{\mathsf{ra}};\ r:=z^{\mathsf{ra}};\ r:=x;\ r:=x$$

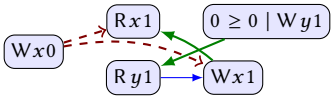


(Co3)

We disallow the outcome due to the cycle: the order from $(\mathrm{R}x1)$ to $(\mathrm{W}x2)$ is required to fulfill $(\mathrm{R}x1)$. Such an outcome would allow racing writes to be visible, even after a full synchronization, invalidating local reasoning about data races (§8).

Internal Reads. The JMM causality test cases [Pugh 2004] are justified via compiler analysis, possibly in collusion with the scheduler: If every observed value can be shown to satisfy a precondition, then the precondition can be dropped. For example, TC1 determines that the following top-level execution should be allowed, as it is in our model:

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

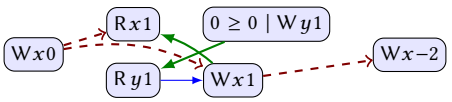


(TC1)

In this example, $(\mathrm{W}x0)$ "fulfills" the read of x that is used in the guard of the conditional. This is possible when prefixing $(\mathrm{R}x1)$ performs the substitution $[x/r]$, but does not weaken the resulting precondition $(x \geq 0 \mid \mathrm{W}y1)$. Subsequently prefixing $(\mathrm{W}x0)$ substitutes $[0/x]$, resulting in the tautological precondition $(0 \geq 0 \mid \mathrm{W}y1)$. Note that the execution does not have an action $(\mathrm{R}x0)$.

This execution is an example of an *internal read*, using ARMV8 terminology [Pulte et al. 2018]. Unlike [Jeffrey and Riely 2016], our semantics is robust with respect to the introduction of concurrent writes, as in TC9:

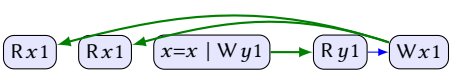
$$x:=0;\ (r:=x;\ \mathtt{if}(r \geq 0)\{y:=1\} \parallel x:=y \parallel x:=-2)$$



(TC9)

The reasoning for TC2 is similar, but in this case no value is necessary to satisfy the precondition:

$$r:=x;\ s:=x;\ \mathtt{if}(r{=}s)\{y:=1\} \parallel x:=y$$



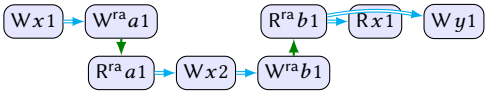
(TC2)

Note that in the prefix $[\![s:=x;\ \mathtt{if}(r{=}s)\{y:=1\}]\!]$, the precondition on $(\mathrm{W}y1)$ must imply $r = 1 \wedge r = x$. The first conjunct is imposed by 4b, the second by 5a. Thus the two reads must see the same value.

Write substitution only effects subsequent reads, and a read action always creates an event that must be fulfilled. In combination, these ensure that an *internal read* cannot ignore a blocking write. In the following execution candidate, there is no order from $(\mathrm{R}x1)$ to $(\mathrm{W}y1)$, potentially allowing the program to write a stale value. However, $(\mathrm{R}x1)$ cannot be fulfilled, causing the execution

candidate to be disallowed:

$$x{:=}1;\ a^{\mathsf{ra}}{:=}1;\ \mathtt{if}(b^{\mathsf{ra}})\{y{:=}x\} \parallel \mathtt{if}(a^{\mathsf{ra}})\{x{:=}2;\ b^{\mathsf{ra}}{:=}1\}$$

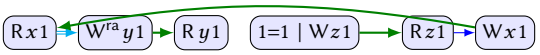


(Internal1)

The execution becomes inconsistent if we change $(\mathsf{R}x1)$ to $(\mathsf{R}x2)$, resulting in $(2{=}1 \mid \mathsf{W}y1)$.

Internal reads are notoriously difficult to get right. Consider [Podkopaev et al. 2019, Ex 3.6]:

$$r{:=}x;\ y^{\mathsf{ra}}{:=}1;\ s{:=}y;\ z{:=}s \parallel x{:=}z$$



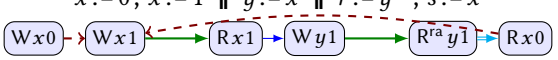
(Internal2)

This behavior is allowed in our model, as it is in ARMv8. Note that $[\![z{:=}s]\!]$ includes $(s{=}1 \mid \mathsf{W}z1)$. Prepending a read, $[\![s{:=}y;\ z{:=}s]\!]$ may update the precondition to $(y{=}1 \mid \mathsf{W}z1)$ without introducing order. Further prepending $(\mathsf{W}^{\mathsf{ra}}y1)$ results in $(1{=}1 \mid \mathsf{W}z1)$.

Our model drops order into actions that depend on a read that can be fulfilled *internally*, by a prefixed write. This is natural consequence of substitution. The ARMv8 model has to jump through some hoops to ensure that internal reads are handled correctly. ARMv8 takes the symmetric approach: rather than dropping order *out of* an internal read, ARMv8 drops the order *into* it. This difference complicates the proof for ARMv8 (§7).

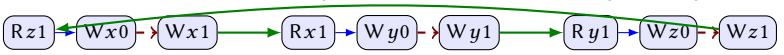
MCA. We present a few examples that are hallmarks of MCA architectures. In MCA1, $(\mathsf{W}x1)$ is delivered to the first second thread, but not the third. MCA2 is an example of *write subsumption* [Pulte et al. 2018, §3].

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



(MCA1)

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



(MCA2)

These candidate executions are invalid, due to cycles. If y^{ra} is changed to y^{rlx} in MCA1, then there would be no order from $(\mathsf{R}^{\mathsf{rlx}}y1)$ to $(\mathsf{R}x0)$, and the execution would be allowed. Since read-read dependencies do not appear in pomset order, the execution would still be allowed if a control or address dependency were to be introduced between the reads. See §10 for further discussion.

3.2 Valid and Invalid Rewrites

When $[\![C]\!] \supseteq [\![C']\!]$, we say that C' is a *valid transformation* of C. In this subsection, we show the validity of specific optimizations. Let $\mathrm{id}(C)$ be the set of locations and registers that occur in C.

The semantics validates several peephole optimizations.

$$[\![r{:=}x;\ s{:=}x;\ C]\!] \supseteq [\![r{:=}x;\ s{:=}r;\ C]\!] \tag{RL}$$

$$[\![x{:=}M;\ y{:=}N;\ C]\!] = [\![y{:=}N;\ x{:=}M;\ C]\!] \quad \text{if } x \neq y \tag{WW}$$

$$[\![r{:=}x;\ y{:=}N;\ C]\!] = [\![y{:=}N;\ r{:=}x;\ C]\!] \quad \text{if } \mathrm{id}(r{:=}x) \cap \mathrm{id}(y{:=}N) = \emptyset \tag{RW}$$

$$[\![r{:=}x;\ s{:=}y;\ C]\!] = [\![s{:=}y;\ r{:=}x;\ C]\!] \quad \text{if } r \neq s \tag{RR}$$

Redundant load elimination (RL) follows from item 1 in the definition of prefixing, taking $d \in E$. The independent reorderings (WW, RW and RR) follow from item **??** in the definition of prefixing,

since no order is imposed. Item **??** also validates roach-motel reorderings. For example, the rules for relaxed writes are as follows:

$$[\![x := M;\ r := y^{\mathsf{ra}};\ C]\!] \supseteq [\![r := y^{\mathsf{ra}};\ x := M;\ C]\!] \qquad \text{if } x \neq y \qquad \text{(AcqW)}$$

$$[\![y^{\mathsf{ra}} := N;\ x := M;\ C]\!] \supseteq [\![x := M;\ y^{\mathsf{ra}} := N;\ C]\!] \qquad \text{if } x \neq y \qquad \text{(RelW)}$$

Since item 5b does not impose order between reads of the same location, RR can allow the possibility that $x = y$. As a result, read optimizations are not limited by the power of aliasing analysis. By composing RR and RL, we validate CSE:

$$[\![r_1 := x;\ s := y;\ r_2 := x;\ C]\!] \supseteq [\![r_1 := x;\ r_2 := r_1;\ s := y;\ C]\!] \qquad \text{if } r_2 \neq s \qquad \text{(CSE)}$$

Many laws hold for the conditional, such as dead code elimination (DC) and code lifting (CL):

$$[\![\mathtt{if}(M)\{C\}\,\mathtt{else}\,\{D\}]\!] = [\![C]\!] \qquad \text{if } M \text{ is a tautology} \qquad \text{(DC)}$$

$$[\![\mathtt{if}(M)\{C\}\,\mathtt{else}\,\{C\}]\!] \supseteq [\![C]\!] \qquad \text{(CL)}$$

Code lifting also applies to program fragments inside a conditional. For example:

$$[\![\mathtt{if}(M)\{x := N;\ C\}\,\mathtt{else}\,\{x := N;\ D\}]\!] \supseteq [\![x := N;\ \mathtt{if}(M)\{C\}\,\mathtt{else}\,\{D\}]\!]$$

We discuss the inverse of CL in §4.

As expected, parallel composition commutes with conditionals and declarations, and conditionals and declarations commute with each other. For example, we have *scope extrusion* [Milner 1999]:

$$[\![C \parallel \mathtt{var}\, x;\ D]\!] = [\![\mathtt{var}\, x;\ (C \parallel D)]\!] \qquad \text{if } x \notin \mathsf{id}(C) \qquad \text{(SE)}$$

As we noted in [Disselkoen et al. 2019, §B], scope extrusion fails if fulfillment requirement F4—for every conflicting write c, either $c \leq d$ or $e \leq c$—is replaced by an existential—there is no conflicting write c such that $d < c < e$[5].

Invalid Rewrites. The definition of location binding does not validate renaming of locations: if $x \neq y$ then $[\![\mathtt{var}\, y;\ C]\!] \neq [\![\mathtt{var}\, x;\ C[x/y]]\!]$, even if C does not mention x. This is consistent with support for address calculation, which is required by realistic memory allocators.

Internal2 shows that—like most relaxed models—our model fails to validate *thread inlining*: The execution above is impossible for $r := x;\ y^{\mathsf{ra}} := 1 \parallel s := y;\ z := s \parallel x := z$. The write in the first thread cannot discharge the precondition in the second.

Some rewrites are invalid in a concurrent setting, such as relevant read introduction:

$$[\![r := x;\ \mathtt{if}(r \neq r)\{z := 1\}]\!] \not\supseteq [\![r := x;\ s := x;\ \mathtt{if}(r \neq s)\{z := 1\}]\!]$$

Observationally, these are distinguished by the context $[-] \parallel x := 1 \parallel x := 2$.

Write introduction is also invalid, even when duplicating an existing write:

$$[\![x := 1]\!] \not\supseteq [\![x := 1;\ x := 1]\!]$$

These are distinguished by the context: $[-] \parallel r := x;\ x := 2;\ s := x;\ \mathtt{if}(r = s)\{z := 1\}$

[5]Such conflicting writes are called *blockers*. The universal quantification ensures that parallel threads cannot turn a non-blocker into a blocker. In that paper, we advocated the use of *three-valued pomsets*, which use strong order for F3 and weak order for F4. The pomsets studied in this paper are far simpler.

4 REFINEMENTS

The previous section shows the simplicity and beauty of pomsets with preconditions as a model weak memory. In this section we look at some of the complications and ugliness.

We limit attention to relaxed access—Candidate 2.9 provides our final definition of prefixing for ra/sc access. In particular, we do not attempt to validate rewrites that eliminate ra/sc accesses. In this section, we elide rlx-mode annotations in definitions.

For relaxed access, the following definition enriches Candidate 2.9 with additional pomsets. As we discuss below, this definition validates read elimination (RE), store forwarding (SF), dead store elimination (DS), and case analysis (CA). We end this section with a brief discussion of a read-enriched semantics that validates irrelevant read introduction (RI).

Definition 4.1. Let $(\phi \mid a) \Rightarrow \mathcal{P}$ be the set $\nabla\mathcal{P}'$ where $P' \in \mathcal{P}'$ when there is some $P \in \mathcal{P}$ that satisfies items 1-5 of Candidate **??** such that:

(6) if d is a release, e_1 is an acquire, $e_1 \le e_2$, then $\Phi(e_2)$ is location independent.

Let $(\mathsf{weaken}_{\mathsf{R}xv}\mathcal{P})$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ when there is $P \in \mathcal{P}$ such that: $E' = E$, $\le' = \le$, $\mathcal{A}' = \mathcal{A}$, and either $\Phi'(e)$ implies $\Phi(e)$ or e is $\le$-minimal[6] and $\mathcal{A}(e) = \mathsf{R}xv$.

Let $(\mathsf{weaken}_{\mathsf{R}x}\mathcal{P})$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ when there is $P \in \mathcal{P}$ such that: $E' = E$, $\le' = \le$, $\mathcal{A}' = \mathcal{A}$, and either $\Phi'(e)$ implies $\Phi(e)$ or e is $\le$-minimal and $\mathcal{A}(e) = \mathsf{R}xv$ (for some v).

Let $(\mathsf{weaken}_{\mathsf{W}xM}\mathcal{P})$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ when there is $P \in \mathcal{P}$ such that: $E' = E$, $\le' = \le$, $\mathcal{A}' = \mathcal{A}$, and either $\Phi'(e)$ implies $\Phi(e)$ or e is $\le$-minimal, $\mathcal{A}(e) = \mathsf{W}xv$, and $\Phi'(e)$ implies $(M = v \vee \Phi(e))$

Let $(\mathsf{cover}_x\mathcal{P})$ be the set $\mathcal{P}' \subseteq \mathcal{P}$ such that $P' \in \mathcal{P}'$ when for every release $e' \in E'$, there is some $d' \in E'$ such that $d' \le e'$ and d' writes x.

$$[\![r := x\,;\, C]\!] \triangleq \mathsf{weaken}_{\mathsf{R}x}[\![C]\!][x/r] \cup \bigcup_v (\mathsf{R}xv) \Rightarrow \mathsf{weaken}_{\mathsf{R}xv}[\![C]\!][x/r]$$
$$[\![x := M\,;\, C]\!] \triangleq \mathsf{weaken}_{\mathsf{W}xM}(\mathsf{cover}_x[\![C]\!][M/x]) \cup \bigcup_v (M = v \mid \mathsf{W}xv) \Rightarrow [\![C]\!][M/x]$$

There are six changes in the definition: To validate read elimination, we include $[\![C]\!][x/r]$. To ensure that this does not allow stale reads, we add item 6 to the definition of prefixing. To validate case analysis on reads actions, we apply $\mathsf{weaken}_{\mathsf{R}xv}$ to $[\![C]\!][x/r]$ before prefixing. To validate case analysis with eliminated reads, we apply $\mathsf{weaken}_{\mathsf{R}x}$ to $[\![C]\!][x/r]$. To validate write elimination, we include $\mathsf{cover}_x[\![C]\!][M/x]$. To validate case analysis with eliminated writes, we apply $\mathsf{weaken}_{\mathsf{W}xv}$.

Read Elimination and Store Forwarding. In our work on microarchitecture [Disselkoen et al. 2019], read actions could be observed using cache effects. Candidate 2.9 maintains this perspective—for example, it distinguishes $[\![r := x]\!]$ and $\supseteq [\![\mathtt{skip}]\!]$ even though there is no context in the language of this paper that can distinguish these programs. If one accepts these programs should be equated at an architectural level, then one would expect the semantics to validate read elimination (RE) and store forwarding (SF).

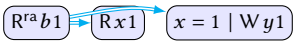
$$[\![r := x\,;\, C]\!] \supseteq [\![C]\!] \qquad \text{if } r \notin \mathsf{id}(C) \qquad \text{(RE)}$$
$$[\![x := M\,;\, r := x\,;\, C]\!] \supseteq [\![x := M\,;\, r := M\,;\, C]\!] \qquad \text{(SF)}$$

These optimizations are validated by Definition 4.1, since $[\![r := x\,;\, C]\!] \supseteq [\![C]\!][x/r]$. The proof of SF also appeals to the definition of write and the definition of register assignment.

Let us revisit the semantics internal read examples from §3.1. With read elimination, the read action $(\mathsf{R}\,y\,1)$ can be elided in Internal2; regardless, the substitution into the write of z is the same. On a more troubling note, the read action $(\mathsf{R}x1)$ can be also elided in Internal1, potentially converting

[6] e is *$\le$-minimal* if there is no d such that $d \le e$

this non-execution into a valid execution, violating DRF-SC. The addition of requirement6 to the definition of prefixing prevents this outcome. When computing $[\![x := 1;\ a^{\mathrm{ra}} := 1;\ \mathtt{if}(b^{\mathrm{ra}})\{y := x\}]\!]$, requirement 6 prevents prefixing $(\mathrm{W}^{\mathrm{ra}}a1)$ in front of:



In order to satisfy requirement 6, the precondition of $(\mathrm{W}y1)$ must be location independent.

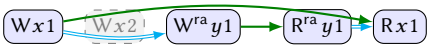
Dead Store Elimination. Dead store elimination (DS) is symmetric to redundant load elimination.

$$[\![x := M;\ x := N;\ C]\!] \supseteq [\![x := N;\ C]\!] \tag{DS}$$

The rewrite is less general than RE because general store elimination is unsound. For example, "$x := 0$" and "$x := 0;\ x := 1$" can be distinguished by the context "$z := x$".

DS is valid under Definition 4.1. A write may only be removed if it is *covered* by a following write. In general, there may need to be many following writes, one for each subsequent release. This prevents bad executions, such as:

$$x := 1;\ x := 2;\ y^{\mathrm{ra}} := 1 \ \|\ r := y^{\mathrm{ra}};\ s := x$$



In this drawing, we have included a "non-event"—dashed border—to mark the eliminated write.

Case Analysis. Definition 4.1 satisfies *disjunction closure*.

Definition 4.2. We say that P is a *disjunct of P' and downset P''* when $E = E' \supseteq E''$, $\leq\ =\ \leq' \supseteq \leq''$, $\mathcal{A} = \mathcal{A}' \supseteq \mathcal{A}''$, $\Phi(e)$ implies $\Phi'(e) \vee \Phi''(e)$ if $e \in E''$, and $\Phi(e)$ implies $\Phi'(e)$ otherwise.

We say that $\mathcal{P}$ is *disjunction closed* if $P \in \mathcal{P}$ whenever there are $\{P',\ P''\} \subseteq \mathcal{P}$ such that P is a disjunct of P' and downset P''.

In the definition of composition, any actions with the same label and downset can coalesce. Disjunction closure is sufficient to establish case analysis (CA):

$$[\![C]\!] \supseteq [\![\mathtt{if}(M)\{C\}\,\mathtt{else}\,\{C\}]\!] \tag{CA}$$

As we discuss below, Candidate 2.9 is not disjunction closed; however, it *becomes* closed if 5b is strengthened to include read-read coherence, thus sacrificing sacrifice CSE. This compromise is considered reasonable for C11 atomics, which are meant to be used sparingly. It is less attractive for plain access in Java-like languages.

We start the discussion in this section assuming Candidate 2.9 with read-coherence. We then look at three relaxations in turn: write elimination, reads elimination, and finally dropping read-read coherence. Definition 4.1 performs a bit of disjunctive closure in each case.

Write elimination causes disjunction closure to fail. For example, consider the following executions of C_{W}; the second uses write elimination:

$$C_{\mathrm{W}} = x := 1;\ \mathtt{if}(r)\{x := 1\}$$

Using disjunction, $[\![\mathtt{if}(s)\{C_{\mathrm{W}}\}\,\mathtt{else}\,\{C_{\mathrm{W}}\}]\!]$ includes the singleton $d{:}(\mathrm{W}x1)$, but $[\![C_{\mathrm{W}}]\!]$ does not. Our solution is to weaken the preconditions on writes to the same location when eliminating a write. We do this using the function $\mathrm{weaken}_{\mathrm{W}xM}$, which weakens the precondition of any $<$-minimal $\mathrm{W}xv$ to $(M = v \vee \phi)$, where ϕ is formula before prefixing. When prefixing $(\mathrm{W}x1) \Rightarrow (r \mid \mathrm{W}x1)$ in the example above, we arrive at the formula $(1 = 1 \vee r)$ for the "old" write, which is a tautology.

With read-read coherence, reads and read elimination are completely symmetric.

Relaxing read-read coherence causes disjunction closure to fail. For example, consider the two sides of the composition defined by the conditional, where $C_{\mathsf{R}} = r := x;\ \texttt{if}(M)\{s := x\}$.

$$\texttt{if}(N)\{C_{\mathsf{R}}\} \qquad\qquad \texttt{if}(\neg N)\{C_{\mathsf{R}}\}$$
$$d\colon \boxed{N \mid \mathsf{R}\,x\,0} \quad e\colon \boxed{N \wedge M \mid \mathsf{R}\,x\,0} \qquad e\colon \boxed{\neg N \mid \mathsf{R}\,x\,0} \quad d\colon \boxed{\neg N \wedge M \mid \mathsf{R}\,x\,0}$$

Because the reads are unordered, they can be confused when coalescing, resulting in:

$$\texttt{if}(N)\{C_{\mathsf{R}}\}\,\texttt{else}\,\{C_{\mathsf{R}}\}$$
$$d\colon \boxed{N \vee (\neg N \wedge M) \mid \mathsf{R}\,x\,0} \quad e\colon \boxed{(N \wedge M) \vee \neg N \mid \mathsf{R}\,x\,0}$$

which is:

$$d\colon \boxed{N \vee M \mid \mathsf{R}\,x\,0} \quad e\colon \boxed{\neg N \vee M \mid \mathsf{R}\,x\,0}$$

But this pomset does not occur in $[\![C_{\mathsf{R}}]\!]$. Our solution is to weaken the preconditions on reads so that both f$[\![C_{\mathsf{R}}]\!]$ and $[\![\texttt{if}(N)\{C_{\mathsf{R}}\}\,\texttt{else}\,\{C_{\mathsf{R}}\}]\!]$ include:

$$d\colon \boxed{\mathsf{R}\,x\,0} \quad e\colon \boxed{\mathsf{R}\,x\,0}$$

Note that the precondition on the reads are weaker than one would expect. This is not a problem for reads, since they must also be fulfilled—allowing more reads actually increases the obligations of fulfillment. The same solution would not work for writes—as we discussed at the end of §3, allowing more writes is simply unsound. Fortunately, this problem does not occur when prefixing a write in front of another write, due to the order required by 5b.

Irrelevant Read Introduction. Compilers sometimes introduce reads in order to lift code. Consider the following example [Ševčík 2008, §1.4.5]:

$$[\![\texttt{if}(r)\{s := x;\ y := s\}]\!] \not\supseteq [\![s := x;\ \texttt{if}(r)\{y := s\}]\!]$$

The right-hand program is derived from the left by introducing an irrelevant read in the else-branch, then moving the common code out of the conditional. Definition 4.1 does *not* validate this rewrite.

Read introduction is only valid "modulo irrelevant reads." We capture this idea using *read saturation*. Read saturation allows us to add actions of the form $(\mathsf{R}\,x\,v)$ to the left-hand pomsets and $(r \neq 0 \mid \mathsf{R}\,x\,v)$ to the right, thus equating them.

Let $\mathsf{read}(\mathcal{P})$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ when $\exists P \in \mathcal{P}$ and there exist $E'' \subset E$ and D such that $E' = E'' \uplus D$, $\leq' \supseteq \leq|_{E''}$, $\lambda' \supseteq \lambda|_{E''}$, and for every $d \in D$ there are x and v such that $\mathcal{A}'(d) = (\mathsf{R}\,x\,v)$. Note that if $\mathcal{P}' \supseteq \mathcal{P}$, then $\mathsf{read}(\mathcal{P}') \supseteq \mathsf{read}(\mathcal{P})$.

Read introduction (RI) is valid under the saturated semantics.

$$\mathsf{read}[\![C]\!] \supseteq \mathsf{read}[\![r := x;\ C]\!] \qquad\qquad \text{if } r \notin \mathsf{id}(C) \tag{RI}$$

With RI, the model satisfies all of the transformations of Ševčík [2008, §5.3-4] except redundant write after read elimination (which is unsound) and reordering with external actions (which we do not model.)

5 EXTENSIONS

We extend the model to include additional features: fences, read-modify-write actions (RMWs), and address calculation. The proofs given later in the paper extend to include these features.

Fences. Syntactic fences "$\mathsf{F}^{v}; C$" have corresponding actions: (F^{v}). The *syntactic fence mode* ($v ::= \mathsf{rel} \mid \mathsf{acq} \mid \mathsf{sc}$) is either *release*, *acquire*, or *sequentially-consistent*. $(\mathsf{F}^{\mathsf{rel}})$ is a release. $(\mathsf{F}^{\mathsf{acq}})$ is an acquire. $(\mathsf{F}^{\mathsf{sc}})$ is both a release and an acquire.

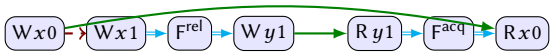
$$[\![\mathsf{F}^{v}; C]\!] \triangleq (\mathsf{F}^{v}) \Rightarrow [\![C]\!]$$

Syntactic fences require additional order to simulate ra/sc-accesses. We add the following rules to Definition **??** of *prefixing*:

(5e) if d reads, and e is an acquiring fence, then $d <' e$, and
(5f) if d is a releasing fence, and e writes, then $d <' e$.

Consider the following variant of Pub1:

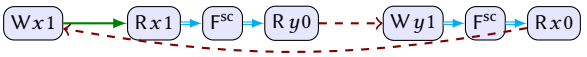
$$x := 0;\ x := 1;\ \mathsf{F}^{\mathsf{rel}};\ y := 1 \ \parallel\ r := y;\ \mathsf{F}^{\mathsf{acq}};\ s := x \qquad \text{(Pub2)}$$



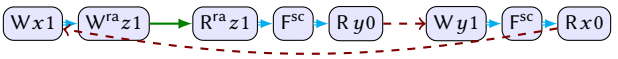
5f requires that $(\mathsf{F}^{\mathsf{rel}}) \le (\mathsf{W}y1)$. 5e requires that $(\mathsf{R}y1) \le (\mathsf{F}^{\mathsf{acq}})$. The attempted execution is *invalid*: the stale read $(\mathsf{R}x0)$ violates the last requirement of fulfillment (Definition 2.5).

Our semantics does not suffer the weaknesses of C11 fences, noted by Lahav et al. [2017, Figs. 5 and 6]. We omit 0-initialization in these examples:

$$x := 1 \ \parallel\ r := x;\ \mathsf{F}^{\mathsf{sc}};\ r := y \ \parallel\ y := 1;\ \mathsf{F}^{\mathsf{sc}};\ r := x \qquad \text{(SC2)}$$



$$x := 1;\ z^{\mathsf{ra}} := 1; \ \parallel\ r^{\mathsf{ra}} := z;\ \mathsf{F}^{\mathsf{sc}};\ r := y \ \parallel\ y := 1;\ \mathsf{F}^{\mathsf{sc}};\ r := x \qquad \text{(SC3)}$$



The executions are disallowed, due to cycles. While these results are immediate in our model, it is worth noting that they are anything but immediate in the various models of C11. Lahav et al. [2017] devote an entire paper to debugging the model of SC access in C11.

Read-Modify-Write. We discuss RMW operations that work on a single location in memory, such as *fetch-and-add* (FADD) and *compare-and-swap* (CAS). These operations can be modeled using read/write actions or using an additional relation between events. The second approach is more general and less obvious, therefore we explain it here.

In Definition 2.2, we require that a *(memory model) pomset* be a tuple $(E, \le, \lambda, \mathsf{rmw})$, where $\mathsf{rmw} \subseteq \le$ relates the two events of a successful RMW. Additionally, we require that:

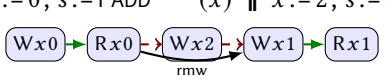
- If c, e write the same x, $c \le e$ and $d \xrightarrow{\mathsf{rmw}} e$ then $c \le d$.
- If c, e write the same x, $d \le c$ and $d \xrightarrow{\mathsf{rmw}} e$ then $e \le c$.

In Definition 2.3, we require that downsets are RMW closed: $E' \subseteq \{d \in E \mid \exists e \in E'.\ e \xrightarrow{\mathsf{rmw}} d\}$.

Other than these two changes, nothing else changes. In particular, RMWs require no special treatment in Definition 2.7: the constituent events of an RMW may coalesce with other events as a result of parallel composition. We elide the obvious and tedious semantic rules that generate rmw.

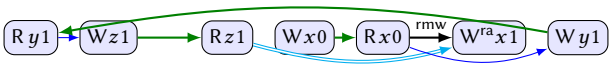
This definition ensures atomicity, disallowing executions such as [Podkopaev et al. 2019, Ex. 3.2]:

$$x := 0;\ s := \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x) \ \parallel\ x := 2;\ s := x \qquad \text{(RMW1)}$$



By using two actions rather than one, the definition allows examples such as the following, which is allowed by ARMv8 [Podkopaev et al. 2019, Ex. 3.10]:

$$r := y;\ z := r \ \|\ r := z;\ x := 0;\ s := \mathsf{FADD}^{\mathsf{rlx,ra}}(x);\ y := s{+}1 \qquad \text{(RMW2)}$$



Address Calculation. In the definition of a data model, we require that locations have the form $x ::= [\ell]$, where ℓ is a value. Expressions may include neither memory locations nor the operator $[L]^{\mu}$. In our example language, we update the syntax of commands:

$$C \ ::= \ \cdots \ \mid \ r := [L]^{\mu};\ C \ \mid \ [L]^{\mu} := M;\ C$$

Address calculation can be encoded using the conditional. We give the semantics simply by expanding this encoding. Applying this technique to Candidate 2.9, we arrive at the following:

$$[\![r := [L]^{\mu};\ C]\!] \triangleq \|_{\ell}\ (L = \ell) \rhd \left(\bigcup_{v}\ (\mathsf{R}^{\mu}[\ell]v) \Rightarrow [\![C]\!][[\ell]/r]\right)$$
$$[\![[L]^{\mu} := M;\ C]\!] \triangleq \|_{\ell}\ (L = \ell) \rhd \left(\bigcup_{v}\ (M = v \mid \mathsf{W}^{\mu}[\ell]v) \Rightarrow [\![C]\!][M/[\ell]]\right)$$

The same technique to can be applied to Definition 4.1—we elide the lengthy but obvious definition. For degenerate programs that include only constant references (every expression $[L]^{\mu}$ satisfies $L = \ell$, for some ℓ), the resulting definition produces exactly the same executions as before.

The rewrites listed in §3-4 remain valid, with the following generalization: For address expressions $[M]$ and $[N]$, replace $x = y$ by provable equality of M and N, and $x \neq y$ by provable inequality.

In Definition 4.1, we were able to ensure disjunction closure by performing targeted, local weakening via $\mathsf{weaken}_{\mathsf{R}xv}$ and $\mathsf{weaken}_{\mathsf{W}xM}$. This is much more difficult to do with address calculation, since a single bit of syntax can refer to multiple locations. Consider that $[\![[r] := 0;\ [0] := !r]\!]$ includes both of the following pomsets ("!" is logical negation: "$!M$" evaluates to 1 if M is 0, and 0 otherwise):

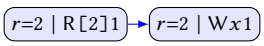
Thus, the disjunction closure also includes both of the following:

In this example, the d events that coalesce correspond to different statements in the syntax.

Because we do not enforce order between reads, there is some danger that address calculations could introduce anomalous behaviors that arise *out of thin air* (OOTA) [Batty et al. 2015]. Consider the following program, where initially $x = 0$, $y = 0$, $[0] = 0$, $[1] = 2$, and $[2] = 1$. It should only be possible to read 0, disallowing the attempted execution below:

$$r := y;\ s := [r];\ x := s \ \|\ r := x;\ s := [r];\ y := s \qquad \text{(OOTA2)}$$

Although no order is enforced between reads, the read-to-write order induced by the semantics is sufficient to prohibit this OOTA behavior. Note the intermediate state of $[\![s := [r];\ x := s]\!]$:



The precondition on $(\mathsf{W}x1)$ is required by causal strengthening (Definition 2.2).

6 UNDERSTANDING "THIN AIR READS" USING TEMPORAL LOGIC

A significant challenge for a software memory model is to relax order enough to allow efficient implementation without admitting anomalous behaviors—called *out of thin air* (OOTA) in the literature [Batty et al. 2015; Boehm 2018; McKenney et al. 2016]. The most famous example is:

$$(y := x \parallel r := y;\ x := r) \qquad \mathrm{R}x1 \to \mathrm{W}y1 \to \mathrm{R}y1 \to \mathrm{W}x1 \qquad \text{(OOTA3)}$$

Although Java does not allow OOTA behaviors of OOTA3, Lochbihler [2013] showed that it does allow OOTA behaviors of OOTA1, from §1. Jeffrey and Riely [2016] described a logic that rules out OOTA3 but not OOTA1. In this section, we provide a more accurate test of OOTA behaviors by enhancing their logic with temporal features.

On first read, we suggest that readers skip to the examples and the discussion that follows, coming back to the details of the logic as necessary. Example 6.2 discusses the canonical OOTA example OOTA3; the analysis is trivial and well-known [Jeffrey and Riely 2019; Kang et al. 2017]. Example 6.3 is more interesting. It discusses a variant of Lochbihler's example OOTA1, from the introduction. The logic given here is not meant to be definitive; on page 26, we discuss OOTA examples that require non-trivial extensions [Chakraborty and Vafeiadis 2019; Svendsen et al. 2018].

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[7]:

$$\begin{array}{ll}
P, e \models \mathrm{W}xv & \text{if } \mathcal{A}(e) = \mathrm{W}xv \text{ and true implies } \Phi(e) \\
P, e \models \mathrm{R}xv & \text{if } \mathcal{A}(e) = \mathrm{R}xv \text{ and true implies } \Phi(e) \\
P, e \models \varrho \wedge \vartheta & \text{if } P, e \models \varrho \text{ and } P, e \models \vartheta \\
P, e \models \mathsf{true} & \\
P, e \models \neg\varrho & \text{if } P, e \not\models \varrho \\
P, e \models \boxminus\varrho & \text{if } \forall d < e.\ P, d \models \varrho \\
P, e \models \diamondminus\varrho & \text{if } \exists d < e.\ P, d \models \varrho
\end{array}$$

Let $P \models \varrho$ if $P, e \models \varrho$, for all $e \in E$.
Let $\mathcal{P} \models \varrho$ if $P \models \varrho$, for all $P \in \mathcal{P}$.
Let $\varrho, \mathcal{P} \models \vartheta$ if $\{P \mid P \models \varrho\} \parallel \mathcal{P} \models \vartheta$.
Let ϱ be *downclosed* when $\{P \mid P \models \varrho\}$ is.

The past operators do not include the current instant, and so do *not* satisfy $(\boxminus\varrho \implies \diamondminus\varrho)$[8]. However, the following hold:

$$\begin{array}{ll}
P \models (\boxminus\varrho \implies \varrho) \implies \varrho & \text{(Induction)} \\
P \models (\varrho \implies \diamondminus\varrho) \implies \neg\varrho & \text{(Coinduction)} \\
P \models (\varrho \implies \diamondminus\vartheta) \implies (\diamondminus\varrho \implies \diamondminus\vartheta) & \text{(Weakening)}
\end{array}$$

We present two proof rules. The first provides a logical view of x-*closure* (Definition 2.5):

$$\frac{\varrho \text{ is independent of } x \qquad P \models (\mathrm{R}xv \implies \diamondminus\mathrm{W}xv) \implies \varrho}{\nu x\,.\,P \models \varrho}$$

The second rule describes concurrent composition, in the style of Abadi and Lamport [1993]. To simplify the presentation, we consider the special case with a single invariant.

Proposition 6.1. *Let ϱ be downclosed. Let $\mathcal{P}_1, \mathcal{P}_2$ be augmentation-closed. Then:*

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

[7] Let false, $\vee$, $\implies$ and $\diamondminus$ as usual; for example, $\diamondminus\varrho = \neg(\boxminus\neg\varrho)$.
[8] The order-minimal elements always validate $\boxminus\varrho$ and invalidate $\diamondminus\varrho$.

Proof sketch. We will show that all downsets in the downset closures of $\mathcal{P}_1 \parallel \mathcal{P}_2$ satisfy the required property. Proof proceeds by induction on downsets of $P \in \mathcal{P}_1 \parallel \mathcal{P}_2$. The case for empty downset follows from assumption that ϱ is downset 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 downset 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 downset of P' and $P' \models \varrho$ by induction hypothesis, we deduce that $P_2 \models \varrho$. Since $P_1 \in \mathcal{P}_1$, by assumption $\varrho, \mathcal{P}_1 \models \varrho$ we deduce that $P \models \varrho$. □

Example 6.2. With all variables initialized to 0, we show that OOTA3 satisfies $\neg \mathrm{W}x1$.
We start with the invariant:

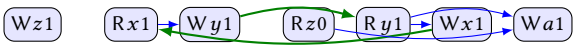
$$[\mathrm{W}x1 \Rightarrow \diamondsuit \mathrm{R}y1] \wedge [\mathrm{W}y1 \Rightarrow \diamondsuit \mathrm{R}x1]$$

This invariant holds for each thread; thus, it holds for the aggregate program by composition. Closing y yields $\mathrm{R}y1 \Rightarrow \diamondsuit \mathrm{W}y1$. Weakening the right conjunct: $\diamondsuit \mathrm{W}y1 \Rightarrow \diamondsuit \mathrm{R}x1$. Chaining these together: $\mathrm{R}y1 \Rightarrow \diamondsuit \mathrm{R}x1$. Weakening: $\diamondsuit \mathrm{R}y1 \Rightarrow \diamondsuit \mathrm{R}x1$. Chaining into the left conjunct: $\mathrm{W}x1 \Rightarrow \diamondsuit \mathrm{R}x1$. Closing x, weakening, then chaining: $\mathrm{W}x1 \Rightarrow \diamondsuit \mathrm{W}x1$. By coinduction, $\neg \mathrm{W}x1$.

Example 6.3. Our language lacks object creation; therefore, we consider a variant of OOTA1:

$$z{:=}1 \parallel y{:=}x \parallel \texttt{if}(z)\{x{:=}1\}\,\texttt{else}\,\{r{:=}y;\ x{:=}r;\ a{:=}r\} \qquad \text{(OOTA4)}$$

The essential temporal property of OOTA1 is "allocation at type C is preceded by a read of 1 for z." OOTA4 retains this structure: "any write to location a is preceded by a read of 1 for z." As a warmup, note that attempting to write 1 to a results in a cycle:



We prove the formula $\neg \mathrm{W}a1$, starting with invariant:

$$[\diamondsuit \mathrm{W}y1 \Rightarrow \diamondsuit \mathrm{R}x1] \wedge [\mathrm{W}a1 \Rightarrow (\diamondsuit \mathrm{R}y1 \wedge \boxminus(\mathrm{W}x1 \Rightarrow \diamondsuit \mathrm{R}y1))]$$

Closing y and chaining into the left conjunct: $\diamondsuit \mathrm{R}y1 \Rightarrow \diamondsuit \mathrm{R}x1$. Chaining into the right conjunct:

$$\mathrm{W}a1 \Rightarrow (\diamondsuit \mathrm{R}x1 \wedge \boxminus(\mathrm{W}x1 \Rightarrow \diamondsuit \mathrm{R}x1))$$

Closing x: $\mathrm{W}a1 \Rightarrow (\diamondsuit \mathrm{W}x1 \wedge \boxminus(\mathrm{W}x1 \Rightarrow \diamondsuit \mathrm{W}x1)$. Applying coinduction to the right conjunct:

$$\mathrm{W}a1 \Rightarrow (\diamondsuit \mathrm{W}x1 \wedge \boxminus(\neg \mathrm{W}x1))$$

Simplifying: $\mathrm{W}a1 \Rightarrow$ false, as required.

Many examples are superficially similar, but in fact have fewer dependencies, such as (∗) from §1. Boehm's [2018] RFUB example presents another potential form of OOTA behavior, in the context of compiler optimization. Our analysis shows that there is no OOTA behavior in RFUB, instead Boehm's analysis has a false dependency:

$$[\![r{:=}y;\ x{:=}r]\!] \not\supseteq [\![r{:=}y;\ \texttt{if}(r{\neq}1)\{z{:=}1;\ r{:=}1\};\ x{:=}r]\!] \qquad \text{(RFUB)}$$

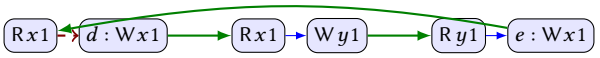
The left command is half of OOTA3. The right command is dubbed RFUB, for *Register assignment From an Unexecuted Branch*. Boehm observes that in the context $x{:=}y \parallel [-]$, these programs have different behaviors. Yet the OOTA example on the left never writes 1. Why should the unexecuted branch change that? As it turns out, both branches of the conditional in RFUB can execute, since the write to x is independent of the read from y. It useful to considering the Hoare logic formulas satisfied by the two threads above: we have $\{\text{true}\}$ RFUB $\{x = 1\}$, but not $\{\text{true}\}$ OOTA $\{x = 1\}$. The change in the thread from OOTA3 to RFUB is not a valid refinement under Hoare logic; as a result, it is expected that RFUB may have additional behaviors.

Understanding OOTA behavior is notoriously difficult, even for the greatest minds in the field! We believe that *logic* is the only tool that can cut the horrible knot that semanticists have tied themselves in. Preconditions provide a *natural* solution to working out these dependencies.

7 EFFICIENT IMPLEMENTATION ON ARMV8

We show that our semantics compiles efficiently to the ARMv8 model of Pulte et al. [2018]. Efficient compilation is not possible for [Flur et al. 2016], referenced in [Lahav and Vafeiadis 2016, Fig. 4]:

$$r := x;\ x := 1 \parallel y := x \parallel x := y$$



This type of "big detour" [Alglave et al. 2014] is outlawed by ARMv8[9].

We consider the fragment of our language where concurrent composition occurs only at top level and there are no location declarations. Using the translation strategy of Podkopaev et al. [2019], we show that any *consistent* ARMV8 execution graph for this sublanguage can be considered a top-level execution of our semantics. The key step is constructing the order for the derived pomset candidate. We would like to take $\leq = (\mathsf{ob} \cup \mathsf{eco})^*$, where ob is the ARMV8 acyclicity relation, and eco is the ARMV8 extended coherence order, as discussed after Definition 2.5. But this does not quite work.

The definition is complicated by ARMV8's *internal reads*, manifest in rfi, which relates reads to writes that are fulfilled by the same thread. ARMV8 drops ob-order *into* an internal read. As discussed in §2, however, our semantics drops pomset order *out of* an internal read. To accommodate this, we drop these dependencies from the ARMV8 *dependency order before* (dob) relation. The relation dob′ is defined from dob by restricting the order into and out of a read that is in the codomain of the rfi relation. More formally, let $d \xrightarrow{\mathsf{dob}'} e$ when $d \xrightarrow{\mathsf{dob}} e$ and $d \notin \mathrm{codom}(\mathsf{rfi}), e \notin \mathrm{codom}(\mathsf{rfi})$. Let ob′ be defined as for ob, simply replacing dob with dob′.

For pomset order, we then take $\leq = (\mathsf{ob}' \cup \mathsf{eco})^*$.

THEOREM 7.1. *For any consistent* ARMV8 *execution graph, the constructed candidate is a top-level memory model pomset.*

The proof for compilation into TSO is very similar. The necessary properties hold for TSO, where ob is replaced by (the transitive closure of) the TSO propagation relation [Alglave et al. 2014].

8 LOCAL DATA RACE FREEDOM AND SEQUENTIAL CONSISTENCY

We adapt Dolan et al.'s [2018] notion of *Local Data Race Freedom (LDRF)* to our setting.

The result requires that locations are properly initialized. We assume a sufficient condition: that programs have the form "$x_1 := v_1;\ \cdots x_n := v_n;\ C$" where every location mentioned in C is some x_i.

We make two further restrictions to simplify the exposition. To simplify the definition of *happens-before*, we ban fences and RMWS. To simplify the proof, we assume all locations are top-level; that is, there are no local declarations of the form $(\mathtt{var}\ x;\ C)$.

To state the theorem, we require several technical definitions. The reader unfamiliar with [Dolan et al. 2018] may prefer to skip to the examples in the proof sketch, referring back as needed.

Data Race. Data races are defined using *program* order (po), not *pomset* order ($\leq$). In SB, for example, $(\mathsf{R}x0)$ has an x-race with $(\mathsf{W}x1)$, but not $(\mathsf{W}x0)$, which is po-before it.

It is obvious how to enhance the semantics of prefixing and most other operators to define po. When combining pomsets using the conditional, the obvious definition may result in cycles, since po-ordered reads may coalesce. In this case we include a separate pomset for each way of breaking these cycles.

[9] There is either a cycle $\mathsf{R}x1 \xrightarrow{\mathsf{poloc}} d \xrightarrow{\mathsf{coe}} e \xrightarrow{\mathsf{rfe}} \mathsf{R}x1$ or $d \xrightarrow{\mathsf{rfe}} \mathsf{R}x1 \xrightarrow{\mathsf{data}} \mathsf{W}y1 \xrightarrow{\mathsf{rfe}} \mathsf{R}y1 \xrightarrow{\mathsf{data}} e \xrightarrow{\mathsf{coe}} d$.

Because we ignore the features of §5, we can adopt the simplest definition of *synchronizes-with* (sw): Let $d \xrightarrow{\text{sw}} e$ exactly when d fulfills e, d is a release, e is an acquire, and $\neg(d \xrightarrow{\text{po}} e)$.

Let $\text{hb} = (\text{po} \cup \text{sw})^+$ be the *happens-before* relation. In Pub1, for example, $(\mathrm{W}x1)$ happens-before $(\mathrm{R}x0)$, but this fails if either ra access is relaxed.

Let $L \subseteq \mathcal{X}$ be a set of locations. We say that d *has an L-race with* e (notation $d \overset{L}{\rightsquigarrow} e$) when they conflict (Definition 2.5) at some location in L, but are unordered by hb: neither $d \xrightarrow{\text{hb}} e$ nor $e \xrightarrow{\text{hb}} d$.

Generators. We say that P' *generates* P if either P augments P' or P implies P'. For example, the unordered pomset $(\mathrm{R}x1)$ $(\mathrm{W}y1)$ generates the ordered pomset $(\mathrm{R}x1) \to (r = 1 \mid \mathrm{W}y1)$.

We say that P is a *generation-minimal* in $\mathcal{P}$ if $P \in \mathcal{P}$ and there is no $P \neq P' \in \mathcal{P}$ that generates P.

Let $\text{gen}[\![C]\!] = \{P \in [\![C]\!] \mid P \text{ is top-level and generation-minimal in } [\![C]\!]\}$.

Extensions. We say that P' *C-extends* P if $P \neq P' \in \text{gen}[\![C]\!]$ and P is a downset of P'.

Similarity. We say that P' *is e-similar to* P if they differ at most in (1) pomset order adjacent to e and (2) the value associated with event e, if it is a read. Formally: $E' = E$, $\Phi' = \Phi$, $\leq'|_{E\setminus\{e\}} = \leq|_{E\setminus\{e\}}$, if e is not a read then $\mathcal{A}' = \mathcal{A}$, and if e is a read then $\mathcal{A}'|_{E\setminus\{e\}} = \mathcal{A}|_{E\setminus\{e\}}$ and $\mathcal{A}'(e) = \mathcal{A}(e)[v'/v]$, for some v', v.

Stability. We say that P is *L-stable in C* if (1) $P \in \text{gen}[\![C]\!]$, (2) P is po-convex (nothing missing in program order), and (3) there is no C-extension of P with a *crossing* L-race: that is, there is no $d \in E$, no P' C-extending P, and no $e \in E' \setminus E$ such that $d \overset{L}{\rightsquigarrow} e$. The empty pomset is L-stable.

Sequentiality. Let $\lessdot_L = <_L \cup \text{po}$, where $<_L$ is the restriction of $<$ to events that access locations in L. We say that P' is *L-sequential after* P if P' is po-convex and $\lessdot_L$ is acyclic in $E' \setminus E$.

Theorem 8.1. *Let P be L-stable in C. Let P' be a C-extension of P that is L-sequential after P. Let P'' be a C-extension of P' that is po-convex, such that no subset of E'' satisfies these criteria. Then either (1) P'' is L-sequential after P or (2) there is some C-extension P''' of P' and some $e \in (E'' \setminus E')$ such that (a) P''' is e-similar to P'', (b) P''' is L-sequential after P, and (c) $d \overset{L}{\rightsquigarrow} e$, for some $d \in (E'' \setminus E)$.*

The theorem provides an inductive characterization of *Local Data-Race Freedom (LDRF)*: Any extension of a L-stable pomset is either L-sequential, or is e-similar to a L-sequential extension that includes a race involving e.

Proof Sketch. In order to develop a technique to find P''' from P'', we analyze pomset order in generation-minimal top-level pomsets. First, we note that $\leq_*$ (the transitive reduction $\leq$) can be decomposed into three disjoint relations. Let $\text{ppo} = (\leq_* \cap \text{po})$ denote *preserved* program order, as required by prefixing (Definition **??**). The other two relations are cross-thread subsets of $(\leq_* \setminus \text{po})$, as required by fulfillment (Definition 2.5): wr orders writes before reads, satisfying fulfillment requirement F3; xw orders read and write accesses before writes, satisfying requirement F4[10].

Using this decomposition, we can show the following.

Lemma 8.2. *Suppose $P'' \in \text{gen}[\![C]\!]$ has a read e that is maximal in $(\text{ppo} \cup \text{wr})$ and such that every po-following read is also $\leq$-following ($e \xrightarrow{\text{po}} d$ implies $e \leq d$, for every read d). Further, suppose there is an e-similar P''' that satisfies the requirements of fulfillment. Then $P''' \in \text{gen}[\![C]\!]$.*

The proof of the lemma follows an inductive construction of $\text{gen}[\![C]\!]$, starting from a large set with little order, and pruning the set as order is added: We begin with all pomsets generated by the semantics without imposing the requirements of fulfillment (including only ppo). We then prune reads which cannot be fulfilled, starting with those that are minimally ordered. This proof is simplified by precluding local declarations.

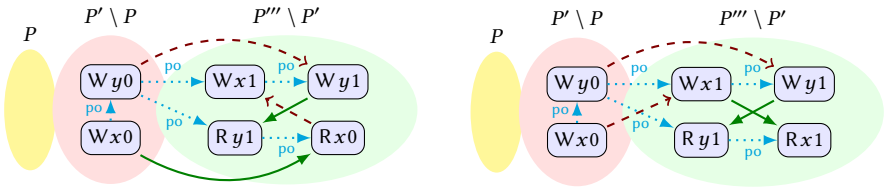
[10] Within a thread, F3 and F4 follow from prefixing requirement 5b, which is included in ppo.

We can prove a similar result for (po ∪ wr)-maximal read and write accesses.

Turning to the proof of the theorem, if P'' is L-sequential after P, then the result follows from (1). Otherwise, there must be a $\lessdot_L$ cycle in P'' involving all of the actions in $(E'' \setminus E')$: If there were no such cycle, then P'' would be L-sequential; if there were elements outside the cycle, then there would be a subset of E'' that satisfies these criteria.

If there is a (po ∪ wr)-maximal access, we select one of these as e. If e is a write, we reverse the outgoing order in xw; the ability to reverse this order witnesses the race. If e is a read, we switch its fulfilling write to a "newer" one, updating xw; the ability to switch witnesses the race. For example, for P'' on the left below, we choose the P''' on the right; e is the read of x, which races with $(\mathrm{W}x1)$.

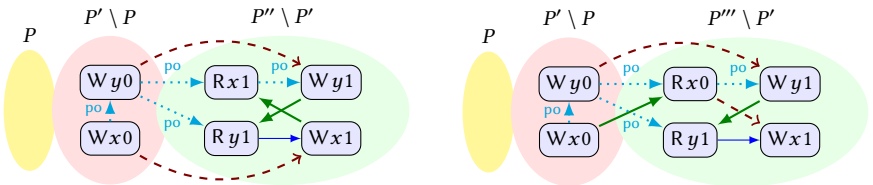
$$x:=0;\ y:=0;\ (x:=1;\ y:=1 \parallel \texttt{if}(y)\{r:=x\}$$



It is important that e be (po ∪ wr)-maximal, not just (ppo ∪ wr)-maximal. The latter criterion would allow us to choose e to be the read of y, but then there would be no e-similar pomset: if an execution reads 0 for y then there is no read of x, due to the conditional.

If there is no (po ∪ wr)-maximal access, then all cross-thread order must be from wr. In this case, we select a (ppo ∪ wr)-maximal read, switching its fulfilling write to an "older" one. As an example, consider the following; once again, e is the read of x, which races with $(\mathrm{W}x1)$.

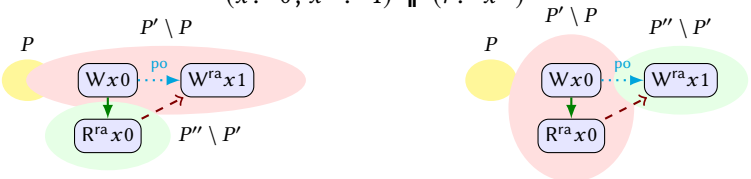
$$x:=0;\ y:=0;\ (r:=x;\ y:=1 \parallel s:=y;\ x:=s)$$



This example requires $(\mathrm{W}x0)$. Proper initialization ensures the existence of such "older" writes. □

The premises of the theorem allow us to avoid the complications caused by "mixed races" in [Dongol et al. 2019]. In the left pomset below, P'' is not an extension of P', since P' is not a downset of P''. When considering this pomset, we must perform the decomposition on the right.

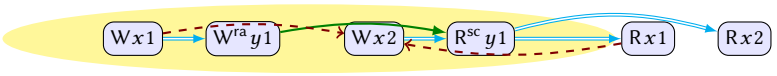
$$(x:=0;\ x^{\mathrm{ra}}:=1) \parallel (r:=x^{\mathrm{ra}})$$



This affects the inductive order in which we move across pomsets, but does not affect the set of pomsets that are considered. This simplification is enabled by denotational reasoning.

In our language, past races are always resolved at a stable point. Consider the following outcome, which is disallowed here, but allowed by Java [Dolan et al. 2018, Example 2]:

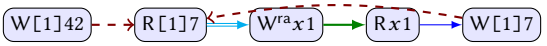
$$(x := 1;\ y^{\mathsf{ra}} := 1) \parallel (x := 2;\ \mathtt{if}(y^{\mathsf{sc}})\{r := x;\ s := x\})$$



The highlighted events are L-stable. The order from $(\mathsf{R}x1)$ to $(\mathsf{W}x2)$ is required by fulfillment, causing the cycle. If we changed y^{sc} to y^{ra}, there would be no order from $(\mathsf{W}x2)$ to $(\mathsf{R}^{\mathsf{ra}}y1)$, the highlighted events would no longer be L-stable, and the execution would be allowed. This more relaxed notion of "past" is not expressible using Dolan et al.'s synchronization primitives.

The notion of "future" is also richer here. Consider [Dolan et al. 2018, Example 3]:

$$(r := 1;\ [r] := 42;\ s := [r];\ x^{\mathsf{ra}} := r) \parallel (r := x;\ [r] := 7)$$



There is no interesting stable point here. The execution is disallowed because of a read from the causal future. If we changed x^{ra} to x^{rlx}, then there would be no order from $(\mathsf{R}[1]7)$ to $(\mathsf{W}^{\mathsf{rlx}}x1)$, and the execution would be allowed. The distinction between "causal future" and "temporal future" is not expressible in Dolan et al.'s operational semantics.

Our definition of L-sequentiality does not quite correspond to SC executions, since actions may be elided by read or write elimination (§4). However, for any properly initialized L-sequential pomset that uses elimination, there is larger L-sequential pomset that does not use elimination. This can be shown inductively: writes that are introduced this way can be ignored by existing reads; reads that are introduced this way can be fulfilled by some preceding write, using its value.

9 OTHER RELATED WORK

We survey related work not discussed previously.

A memory consistency model for a shared-memory multiprocessor defines the values that a read may return. For a survey of hardware models, see [Alglave 2010]. For software models, see [Batty 2015; Lochbihler 2013]. For an attempt to bridge the two, see [Podkopaev et al. 2019].

In our previous work [Disselkoen et al. 2019], we introduced the notion of pomsets with preconditions. This was to study *micro-architecture*—specifically, speculative execution. We have presented an *architectural* model, leading to some formal differences between the models. Chief among these: we now require pomsets to satisfy *consistency* requirement, which simplifies the model, though it can no longer reason about executions where speculative behaviour can become visible, for example by cache timing attacks. There are many less fundamental differences. For example, the previous use of *three-valued pomsets* means that we allowed MCA2, but disallowed its two-location variant. This odd behavior stems from *semi-transitivity*, inherited from [Lamport 1986].

Both [Disselkoen et al. 2019] and this paper are inspired by Batty's [2017] call for a compositional approach to the semantics of relaxed memory concurrency.

Our model shares important structural elements with that of Paviotti et al. [2020], who provide a fix for the thin-air problem in C11. Like us, they use true concurrency semantics to identify (in)dependence in an execution and thus calculate the preserved program order explicitly. Our definition of parallel composition allows events to coalesce, taking preconditions via disjunction—this is mirrored by Paviotti et al.'s definition of coproduct. Their condition on $\leq^X$ (discussed in their §6.3) is mirrored by our requirement that events that coalesce must have the same downset. Nonetheless, the papers have different goals, leading to different outcomes. For example, their model compiles efficiently to non-MCA architectures; our model clearly does not! Conversely, our

model provides an intrinsic characterization optimizations, such as redundant read elimination, which only hold in their model up to observational refinement.

True concurrency techniques have been applied to relaxed memory by Cenciarelli et al. [2007], Castellan [2016], Pichon-Pharabod and Sewell [2016], and Chakraborty and Vafeiadis [2017]. Partial order models have been developed by Brookes [2016] and Kavanagh and Brookes [2019].

There is also a rich literature on the use of transformations over SC executions to model relaxed memory: 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. Demange et al. [2013] developed BMM, which permits reordering of a relaxed write with a following relaxed read. BMM is designed as a restriction of the JMM that compiles efficiently to TSO. It requires fencing on other architectures.

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 and RAW together with 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."

10 LIMITATIONS

Our work has several limitations, each of which provides an opportunity for future research.

We do not include loops or general sequencing of the form $C\,;\,C'$. Loops introduce complexities —such as liveness and continuity—that are orthogonal to the main topic of the paper. We also do not include types, allocation, garbage collection, etc.

Except for roach motel (RelW, AcqW), we have not attempted to validate transformations that involve synchronizations and fences [Vafeiadis et al. 2015]. For example, the model does not support elimination of redundant synchronization accesses.

The logic we presented in §6 is only strong enough to prove a few examples. Svendsen et al. [2018] presented a different a logic, capable of showing that the following program cannot write 2:

$$(y := x + 1 \parallel x := y) \qquad \mathsf{R}x1 \to \mathsf{W}y2 \to \mathsf{R}y1 \to \mathsf{W}x1 \qquad \text{(OOTA5)}$$

The attempted execution is *not* allowed by our semantics since there is no write to fulfill $(\mathsf{R}y1)$. Proving this requires the ability to reason about values.

As another example, consider the following, from Chakraborty and Vafeiadis [2019, Fig. 3].

$$x := 2;\ \mathtt{if}(x \neq 2)\{y := 1\} \parallel x := 1;\ r := x;\ \mathtt{if}(y)\{x := 3\} \qquad \text{(OOTA6)}$$

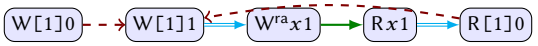
$$\mathsf{W}x2 \to \mathsf{R}x3 \to \mathsf{W}y1 \to \mathsf{W}x1 \to \mathsf{R}x2 \to \mathsf{R}y1 \to \mathsf{W}x3$$

The attempted execution is *not* allowed by our semantics, due to the evident cycle. Surprisingly, this outcome is allowed by the promising semantics [Kang et al. 2017]. Chakraborty and Vafeiadis developed WEAKESTMO to address this example. Intuitively, it is not possible for the left thread to read 3 for x when the right thread reads 2. Proving this may require a logic with modalities to deal with intervening writes and coherence.

Our model realizes *multi-copy atomicity* (MCA). Thus it will not compile efficiently to non-MCA architectures, such as POWER and ARMV7. To do so, one cannot include the order required by F4 in pomset order. It may be sufficient to use a weaker second order. Disselkoen et al. [2019] attempted to define such a weaker order, but their model is anomalous on MCA2—as we discussed in §9.

In the discussion of MCA1, we noted that our model does not enforce order between reads dues to address and control dependencies. This has implications for Java's final field semantics.

$$(r := 1;\ [r] := 0;\ [r] := 1;\ x^{\mathsf{ra}} := r) \ \parallel\ (r := x^{\mathsf{ra}};\ s := [r])$$



If we changed x^{ra} to x^{rlx}, then there would be no order from $(\mathsf{R}^{\mathsf{ra}}\, x\, 1)$ to $(\mathsf{R}\,[1]\,0)$, and the execution would be allowed. It may be desirable to distinguish address dependencies from other dependencies—doing so would likely require two separate preconditions for each event.

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, 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 Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

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

Mark John Batty. 2017. Compositional relaxed concurrency. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017).

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

Stephen Brookes. 2016. A denotational semantics for weak memory concurrency. http://www.cs.bham.ac.uk/~pbl/mgs2016/brookesslides.pdfhttp://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1780r0.html. Midlands Graduate School in the Foundations of Computing Science.

Simon Castellan. 2016. Weak memory models using event structures. In *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*. Saint-Malo, France. https://hal.inria.fr/hal-01333582

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

Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*. ACM, 100–110.

Soham Chakraborty and Viktor Vafeiadis. 2018. Private correspondence.

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *PACMPL* 3, POPL (2019), 70:1–70:28. https://doi.org/10.1145/3290383

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 *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

Brijesh Dongol, Radha Jagadeesan, and James Riely. 2019. Modular transactions: bounding mixed races in space and time. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 82–93. https://doi.org/10.1145/3293883.3295708

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 608–621. https://doi.org/10.1145/2837614.2837615

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

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

Alan Jeffrey and James Riely. 2019. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. *Logical Methods in Computer Science* 15, 1 (2019). https://doi.org/10.23638/LMCS-15(1:33)2019

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

J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. 2018. Private correspondence.

Ryan Kavanagh and Stephen Brookes. 2019. A Denotational Semantics for SPARC TSO. *Logical Methods in Computer Science* 15, 2 (2019). https://doi.org/10.23638/LMCS-15(2:10)2019

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

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

Paul E. McKenney, Alan Jeffrey, Ali Sezgin, and Tony Tye. 2016. 0422R0: Out-of-thin-air execution is vacuous. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0422r0.html.

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.

Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science)*, Peter Müller (Ed.), Vol. 12075. Springer, 599–625. https://doi.org/10.1007/978-3-030-44914-8_22

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

William Pugh. 1999. Fixing the Java Memory Model. In *Proceedings of the ACM 1999 Conference on Java Grande, JAVA '99, San Francisco, CA, USA, June 12-14, 1999*, Geoffrey C. Fox, Klaus E. Schauser, and Marc Snir (Eds.). ACM, 89–98. https://doi.org/10.1145/304065.304106

W. Pugh. 2004. Causality Test Cases. https://perma.cc/PJT9-XS8Z

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

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

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, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. 2015. Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 209–220. https://doi.org/10.1145/2676726.2676995

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