Pomsets with Preconditions*

A Simple Model of Relaxed Memory

RADHA JAGADEESAN, DePaul University, USA
ALAN JEFFREY, Mozilla Research and the Servo Project, USA
JAMES RIELY, DePaul University, USA

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 ARM8. We show that the resulting model (1) supports compositional reasoning for temporal safety properties, (2) supports all expected sequential compiler optimizations, (3) satisfies the SC-DRF criterion, and (4) compiles to x64 and ARM8 microprocessors without requiring extra fences on relaxed accesses.

CCS Concepts: • **Theory of computation → Parallel computing models**; *Preconditions*; *Denotational semantics.*

Additional Key Words and Phrases: Concurrency, Relaxed Memory Models, Multi-Copy Atomicity, ARMv8, Pomsets, Preconditions, Temporal Safety Properties, Thin-Air Reads, Compiler Optimizations

ACM Reference Format:
Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with Preconditions: A Simple Model of Relaxed Memory. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 194 (November 2020), 30 pages. https://doi.org/10.1145/3428262

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 to "ease of use." First, programs should support *compositional* and *local* reasoning; in this paper, we emphasize temporal safety properties [Abadi and Lamport 1993; Misra and Chandy 1981; Pnueli 1984; Stark 1985]. Second, relaxed memory should be not change the behavior of correctly synchronized programs; this property is known as *sequential consistency for data race free programs* (SC-DRF) [Adve and Hill 1990, 1993].

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

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.

*This paper has been greatly improved by the comments of the anonymous reviewers. It 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 NSF.

Authors' addresses: Radha Jagadeesan, DePaul University, Chicago, USA; Alan Jeffrey, Mozilla Research and the Servo Project, Chicago, USA; James Riely, DePaul University, Chicago, USA.

2475-1421/2020/11-ART194
https://doi.org/10.1145/3428262

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 1996]. The resulting model mostly satisfies the desiderata. We sacrifice only implementability on "non-MCA" processors, such as POWER and ARM7. 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 difficulties involved, consider that the existence of an execution in a relaxed memory model may depend on code that was *not* executed. Let r–s be registers and b, x–z be shared memory locations. Consider the following program, where all memory locations are initialized to 0:

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

Most programmers would be surprised to learn that this program allows an execution that sets z to 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:

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

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:

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

Then z is 1 at then end of an execution where the first thread is interleaved immediately after executing $x := 1$. Without the conditional in (*), it is obvious that the program should not write 1:

$$y := x \parallel r := y;\ x := r;\ z := r \qquad (\text{OOTA1})$$

In OOTA1, the constant 1 arises "Out Of Thin Air" (OOTA) [Batty et al. 2015]. As a result, any model of relaxed memory that supports common compiler optimizations, as above, must take into account code that was not executed. This is why many models of relaxed memory include some form of speculative execution, with the goal of allowing the outcome z=1 for (*), but not OOTA1.

The control flow variant of OOTA1 is:

$$\mathtt{if}(x)\{y := 1\} \parallel \mathtt{if}(y)\{x := 1;\ z := 1\} \qquad (\text{OOTA2})$$

This program is data-race-free. Thus, allowing an execution that writes 1 would violate SC-DRF.

OOTA behaviors can be quite subtle. Consider the following variants of (*):

$$y := x \parallel r := y;\ \mathtt{if}(r)\{x := r;\ z := r\}\,\mathtt{else}\,\{x := 2\} \qquad (\text{OOTA3})$$

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

Following the reasoning above, OOTA3 has an execution where z=2, but it does not have an execution where z=1. Neither does OOTA4. In this case, it not sound to assume that 1 is written on both sides of the conditional, invalidating the first optimization given for (*) above.

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?

$$(r_1 := x;\ s := y;\ r_2 := x;\ C) \qquad\qquad (r_1 := x;\ r_2 := r_1;\ s := y;\ C)$$

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

Lochbihler's monumental study of the JMM revealed a surprising limitation. Consider the following program [Lochbihler 2013, Fig. 8], where again all memory locations are initialized to 0:

$$y := x \parallel r := y;\ \mathtt{if}(b)\{r := \mathtt{new\,D};\ x := r;\ z := r\}\,\mathtt{else}\,\{s := \mathtt{new\,C};\ x := r\} \parallel b := 1 \qquad (\text{OOTA5})$$

OOTA5 "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." The JMM allows $(r := y)$ to see the object created by new, by bouncing it through (x:=r) and (y:=x). In the *commitment order* of the JMM, this allows the address of the allocated object to be read (r:=y) before its type is determined (if(b)).

This type of *bait-and-switch* behavior 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. Even partitioning memory to achieve type safety, there are implications for the Java security architecture [Lochbihler 2013, §5.4].

In both OOTA4 and OOTA5, the OOTA outcome occurs by *baiting* with the else branch, then *switching* to the then branch, based on a coin flip (if(b)). 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; Manson et al. 2005] all allow OOTA behaviors of OOTA4.[1] Due to the similarity of OOTA4 and OOTA5, it is reasonable to conclude 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 [Boehm 2007]. C11 also allows OOTA behaviors, exploiting causality cycles. Undefined/OOTA behavior is antithetical to the goals of safe languages.

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. All require fences after relaxed reads, even on ARM8.

Our Model. In our approach, a program is a set of executions. Each execution is a *pomset*: a partial order over a set of read and write events. The order is intended to be read as a *dependency* relation. The dependency relation is dynamic, varying between executions. Events that are not related in an execution are *independent* and can be seen by a sequential observer in either order.

Cross thread dependencies arise from conflicting actions on the same variable: Roughly, we order any two actions on the same location, at least one of which is a write. In the parlance of hardware memory models [Alglave et al. 2014]: coe, fre, and rfe are included in the global 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]. As envisioned in [Alglave 2010, §3.3], this allows us to capture cross-thread dependencies in a single partial order.

Our key insight is that MCA *permits a single, global notion of time, manifest in the pomset order.*

Within a thread, the dependency calculation can be viewed as the computation of *preserved program order*, called ppo in hardware models. In our software model, ppo captures the *essential dependencies* between events in the same thread. Consider the following program fragments:

$$\begin{array}{l} C_1 : x\!:=\!1;\ y\!:=\!1 \\ C_2 : r\!:=\!x;\ \texttt{if}(r)\{\ y\!:=\!1\}\,\texttt{else}\,\{y\!:=\!1\} \\ C_3 : x\!:=\!1;\ r\!:=\!x;\ \texttt{if}(r)\{y\!:=\!1\} \end{array}$$

Each of these fragments satisfy the Hoare triple $\{\texttt{true}\}\ C_i\ \{y = 1\}$; 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.

Our key insight is to that *logic is better than syntax* to capture such dependencies.

[1] Call the threads s, t, and u. To get the result in the promising semantics, first execute u to get message <b:1@1>. Then t promises <x:1@1>, which it can fulfill by reading b=0. Then execute s to get message <y:1@1>. Then execute t, reading b=1 and y=1 and fulfill the promise by writing <x:1@1>. The execution is exactly the same in our speculative semantics [Jagadeesan et al. 2010], removing timestamps and replacing the word *promise* by *speculation*.

The logical perspective provides a clear intuition as to why certain compiler transformations should be valid. Such intuitions are not always readily available in relaxed memory models. For example, *value range analysis*—used in the discussion of ($\ast$)—is difficult in many models. As another example, models such as ARM8 distinguish *internal* reads, which are fulfilled by a write of the same thread, from *external* ones, which are fulfilled cross-thread. Unlike external reads, internal reads are not necessarily recorded in the dependency relation. As exemplified by C_3, this allows a compiler to reorder the fulfilling write with subsequent code that depends on the read. Neither value range analysis nor internal reads require special treatment in our model.

In §2 and §4, we define the model. We show that the model:

- validates expected litmus cases and compiler optimizations (§3-4).
- captures all C11 concurrency features (§5),
- allows compositional reasoning for temporal safety, disallowing OOTA behavior (§6),
- is implementable on ARM8/TSO *without* extra synchronization for relaxed access (§7), and
- satisfies the *local* SC-DRF criterion [Dolan et al. 2018] (§8).

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

Batty [2017] observed that "the current crop of relaxed memory models can only be used to calculate the behavior of a whole program…" and argued that instead, we should "consider a program as an aggregate of components over different models, composed together." Our work is inspired by this call for *compositionality* in models of relaxed concurrency.

Our model is compositional in the normal sense of a denotational semantics: for example, the denotation $\llbracket C_1 \parallel C_2 \rrbracket$ is computed from $\llbracket C_1 \rrbracket$ and $\llbracket C_2 \rrbracket$. The model obeys laws such as scope extrusion—$\llbracket C \parallel \mathtt{var}\,x;\, D \rrbracket = \llbracket \mathtt{var}\,x;\,(C \parallel D) \rrbracket$ when $x \notin \mathsf{id}(C)$—and case analysis—$\llbracket C \rrbracket = \llbracket \mathtt{if}(M)\,\{C\}\,\mathtt{else}\,\{C\} \rrbracket$. This kind of algebraic reasoning is not supported by current models.

Our model also supports compositional reasoning about *data races* (§8), isolating races in space and time. Spatial separation ensures that a race on one location does not invalidate SC-DRF at other locations. Temporal separation ensures that SC-DRF can be applied within a properly synchronized region, unaffected by races that precede or follow.

Finally, our model supports compositional reasoning about *temporal safety properties* (§6). Consider that each thread of OOTA4 satisfies the following invariant: *A write of* 1 *to y must be preceded by a read of* 1 *from x, and if* 1 *is written to z then a write of* 1 *to x must be preceded by a read of* 1 *from y.* Compositionality allows us to conclude that whole program satisfies this property. As noted above, this reasoning *fails* in models based on promises, speculations, or commitments. Compositionality for temporal safety is a *verifiable* criterion for claiming that a model rejects OOTA executions.

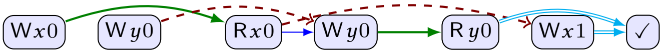
2 THE BASIC MODEL

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 simplified: as discussed in §4, it fails to validate some important optimizations. We give the full semantics in §4. In §5, we define extensions that incorporate fences, read-modify-write operations, and address computation.

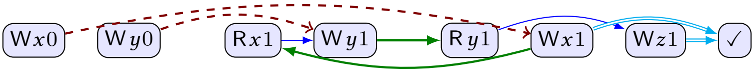
The model is based on *partially ordered multisets* [Gischer 1988; Plotkin and Pratt 1996], where events are labelled with reads and writes, and the partial order tracks dependencies, which arise within threads due to local dependencies and across threads due to reads and coherence.

For example the semantics of ($\ast$) contains the expected pomset (where ✓ indicates termination):

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



but also the unexpected one:

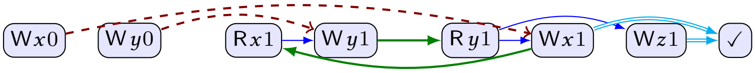


(†)

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

An attempt to replicate this execution with OOTA3 fails, since it introduces a cycle:

$$x{:=}0;\ y{:=}0;\ (y{:=}x \parallel r{:=}y;\ \mathtt{if}(r)\{x{:=}r;\ z{:=}r\}\,\mathtt{else}\,\{x{:=}2\})$$



(OOTA3)

In this case, $(\mathsf{W}x1)$ only happens on one side of the conditional, causing a control dependency from $(\mathsf{R}y1)$ to $(\mathsf{W}x1)$. Similar cycles arise when attempting to write z=1 in OOTA1–OOTA4.

In the diagrams above, there is only one order—color plays no formal role. We use color only to help the reader see where the order comes from:

- $(\mathsf{R}y1)$ → $(\mathsf{W}x1)$ is a *local* requirement, relating reads to writes that depend on them.
- $(\mathsf{W}x1)$ → $(\mathsf{R}x1)$ is a *reads-from* requirement, relating writes to reads they fulfill.
- $(\mathsf{W}x0)$ ⇢ $(\mathsf{W}x1)$ is a *coherence* requirement, relating actions that touch the same location.
- $(\mathsf{W}^{\mathsf{ra}}z1)$ ⇒ (✓) is a *fencing* requirement, involving fences and synchronization actions.

Each pomset event is labeled with a *precondition*, in addition to the actions shown above. Whereas read actions represent an obligation that must be *fulfilled* concurrently by a matching write (Def. 2.7), 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 $(\mathsf{W}x1)$ during the calculation of (†). First consider the else-branch of the conditional: the semantics of "$\mathtt{if}(\neg r)\{x{:=}1\}$" contains $(r{=}0 \mid \mathsf{W}x1)$, indicating the control dependency. The then-branch is more complex: the semantics of "$\mathtt{if}(r)\{x{:=}r\}$" contains $(r{\neq}0 \land r{=}1 \mid \mathsf{W}x1)$ indicating both a control and a data dependency. This can be simplified to $(r{=}1 \mid \mathsf{W}x1)$. Combining the two branches of the conditional, we have $(r{=}0 \lor r{=}1 \mid \mathsf{W}x1)$. Prepending $r{:=}y$ substitutes $[y/r]$, resulting in $(y{=}0 \lor y{=}1 \mid \mathsf{W}x1)$. Prepending the initializer $y{:=}0$ substitutes $[0/y]$, resulting in $(0{=}0 \lor 0{=}1 \mid \mathsf{W}x1)$. This is a tautology, which we write as $(\mathsf{W}x1)$. We repeat this calculation in §2.6, after giving the formal definitions.

The same calculation fails for the write to x in OOTA1–OOTA4. In OOTA3, for example, the writes to x on either side of the conditional cannot be combined, since one side writes 1 and the other side writes 2. Thus, the semantics of the conditional contains $(r{=}1 \mid \mathsf{W}x1)$, rather than $(r{=}0 \lor r{=}1 \mid \mathsf{W}x1)$. As we shall see (Def. 2.6), in OOTA3 it is only possible to weaken this precondition by introducing order from $(\mathsf{R}y1)$ to $(\mathsf{W}x1)$.

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, y, z, and b
- a set of *actions* $\mathcal{A}$, ranged over by a, 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,[2] 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 $\mathsf{Wr} : \mathcal{A} \rightarrow (\mathcal{X} \times \mathcal{V})$, and there are subsets of $\mathcal{A}$: Acq, Rel, SC, and Term, such that $\mathsf{dom}(\mathsf{Rd}) \cap \mathsf{SC} \subseteq \mathsf{Acq}$, $\mathsf{dom}(\mathsf{Wr}) \cap \mathsf{SC} \subseteq \mathsf{Rel}$, and $\mathsf{Term} \subseteq \mathsf{Rel}$.

- We say that action a is a *read* if $a \in \mathsf{dom}(\mathsf{Rd})$. We say that a is a *write* if $a \in \mathsf{dom}(\mathsf{Wr})$. 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. If OOTA executions are disallowed for C11 relaxed access, then C11 plain access is the same as relaxed access for data race free programs; data races on plain access result in undefined behavior [Boehm 2007].

Logical formulae include equations over locations and registers, such $(x{=}1)$ and $(r{=}s{+}1)$. We use expressions as formulae, coercing M to $M \neq 0$.

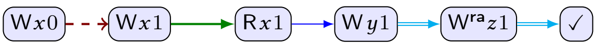
Formulae are *open*, in that occurrences of register names and memory locations are subject to substitutions of the form $\phi[x/r]$ and $\phi[N/x]$. Actions are not subject to substitution.

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 $\phi \vDash \mathsf{false}$.

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.

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



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

[2] 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 $\phi[M/r] = \phi[x/r][M/x]$.

dependent on the read of x; *control* and *address* dependencies are also local. $(\mathsf{W}y1) \Rightarrow (\mathsf{W}^{\mathsf{ra}}z1)$ and $(\mathsf{W}^{\mathsf{ra}}z1) \Rightarrow (\checkmark)$ are *fencing/synchronization* requirements: $(\mathsf{W}^{\mathsf{ra}}z1)$ and $(\checkmark)$ are *release* actions.

Definition 2.2. A *pomset with preconditions* is a tuple $(E, \le, \lambda)$, such that

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

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

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

Preconditions are linked to pomset order via *causal strengthening*, which requires that formulae do not weaken over time, as measured by $\le$. Example ADDR1 (§5) requires causal strengthening.

Let $\mathcal{P}$ range over sets of pomsets. The semantics of a program is given as a set of pomsets $\mathcal{P}$ that is closed with respect to *downsets* (which are similar to prefixes for strings), to *augmentation* (which may add order), and to *implication* (which may strengthen formulae).

Definition 2.3. P' is a *downset* of P if $E \supseteq E' \supseteq \{d \in E \mid \exists e \in E'.\ d \le e\}$, $\le' = \le|_{E'}$, and $\lambda' = \lambda|_{E'}$.
We say that P' is an *augment* of P if $E' = E$, $\lambda' = \lambda$, and $\le' \supseteq \le$.
We say that P' *implies* P if $E' = E$, $\le' = \le$, $\mathcal{A}' = \mathcal{A}$, and $\Phi'(e)$ implies $\Phi(e)$.

In examples, we draw pomsets that are *augmentation-minimal* and *implication-minimal.*

A pomset is *completed* if it contains a unique termination action, ordered after all other events. Note that, by causal strengthening, the precondition of the termination event of a completed pomset must imply the preconditions of all other events.

The semantics of a program includes only completed pomsets and their downsets. We systematically elide the termination event in diagrams, unless it is relevant to the discussion.

2.3 Example Language

We define the language by prefixing individual reads and writes.

$$\begin{aligned} C, D \ ::=&\ \mathtt{skip} \mid r\mathtt{:=}M;C \mid r\mathtt{:=}x^{\mu};C \mid x^{\mu}\mathtt{:=}M;C \\ &\mid C \parallel D \mid \mathtt{var}\,x;C \mid \mathtt{if}(M)\{C\}\,\mathtt{else}\,\{D\} \end{aligned}$$

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 $y\mathtt{:=}\mathbb{M};C$ is shorthand for $r\mathtt{:=}x;\ y\mathtt{:=}\mathbb{M}[r/x];C$. Each occurrence of x in an extended expression corresponds to an separate read.

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

The semantic function $[\![-]\!]$ takes a command and yields a set of pomsets.

2.4 Composition and Concurrency

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. As in our previous work [Disselkoen et al. 2019], composition is used to define concurrency and conditionals. Here, we also use it to define address calculation (§5).

Definition 2.4. 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 P^1 is completed exactly when P^2 is completed, there is at most one termination in E', $E' = E^1 \cup E^2$, $\leq' \supseteq \leq^1 \cup \leq^2$, and for all $e \in E'$, either:

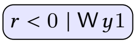
$$\begin{gathered} e \notin E^2,\ \mathcal{A}'(e) = \mathcal{A}^1(e) \text{ and } \Phi'(e) \text{ implies } \Phi^1(e), \\ e \notin E^1,\ \mathcal{A}'(e) = \mathcal{A}^2(e) \text{ and } \Phi'(e) \text{ implies } \Phi^2(e), \text{ or} \\ \mathcal{A}'(e) = \mathcal{A}^1(e) = \mathcal{A}^2(e) \text{ and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

We then define:

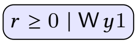
$$[\![C \parallel D]\!] \triangleq [\![C]\!] \parallel [\![D]\!]$$

Consider the following pomsets:

$\texttt{if}(r<0)\{y\texttt{:=}1\}$



$\texttt{if}(r\geq0)\{y\texttt{:=}1\}$



The parallel composition of these programs includes pomsets with either one of the two events, but not both. Including both would violate consistency (Def. 2.2). However, Definition 2.4 allows events with the same label to *coalesce*, taking the disjunction of their preconditions. Thus, the semantics of the combined program also includes $(r<0 \vee r\geq 0 \mid \mathsf{W}y1)$. As discussed in the next subsection, coalesced events inherit order from both sides.

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'.

2.5 Conditional, Register Assignment, and Skip

Conditional execution is defined using parallel composition and *filtering*: $(\phi \rhd \mathcal{P})$ selects the subset of pomsets in $\mathcal{P}$ whose preconditions all imply ϕ. Register assignment is defined using substitution: $(\mathcal{P}\sigma)$ performs the substitution σ on every *formula* in $\mathcal{P}$—*actions* are not subject to substitution. The semantics of $\texttt{skip}$ is defined using singleton pomsets with label ✓.

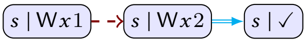
Definition 2.5. Let $P \in (\phi \rhd \mathcal{P})$ when $P \in \mathcal{P}$ and $\Phi(e)$ implies ϕ. Let $P' \in (\mathcal{P}\sigma)$ when there is $P \in \mathcal{P}$ such that $E' = E$, $\leq' = \leq$, $\mathcal{A}' = \mathcal{A}$, and $\Phi'(e) = \Phi(e)\sigma$. Let $P \in$ SKIP when E has one element labelled with action ✓. We then define:

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

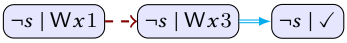
Substitution applies to formulae, not actions. For example, $(x{=}1 \mid \mathsf{W}x2)[0/x] = (0{=}1 \mid \mathsf{W}x2)$.

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

$\texttt{if}(s)\{x\texttt{:=}1;\,x\texttt{:=}2\}$



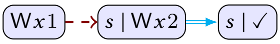
$\texttt{if}(\neg s)\{x\texttt{:=}1;\,x\texttt{:=}3\}$

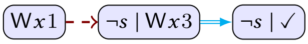


(‡)

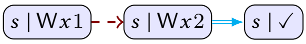
Putting these together, we can coalesce the $(\mathsf{W}x1)$ events:

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

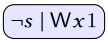




Let us focus on the left pomset above. It is derived from the composition:



$\parallel$



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

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

2.6 Prefixing

Prefixing adds a new read or write event to the beginning of a pomset. The candidate definition given here fails to validate some compiler optimizations. We give the final definition in §4.

Maintaining downset closure complicates the definition in uninteresting ways; therefore, we perform this closure explicitly. Let $\nabla\mathcal{P} = \{P' \mid P' \text{ is a downset of some } P \in \mathcal{P}\}$.

Candidate 2.6. 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 (P1) $E' = E \cup \{d\}$, (P2) $\leq' \supseteq \leq$, (P3A) $\mathcal{A}'(e) = \mathcal{A}(e)$, (P3B) $\mathcal{A}'(d) = a$,

(P4A) if $d \in E$ then $\Phi'(d)$ implies $\phi \vee \Phi(d)$, otherwise $\Phi'(d)$ implies ϕ,
(P4B) if d does not read then either $e = d$ or $\Phi'(e)$ implies $\Phi(e)$,
(P4C) if d reads v from x then either $e = d$ or $\Phi'(e)$ implies $\Phi(e)[v/x]$,
(P5A) if d reads and e writes then either $e = d$ or $\Phi'(e)$ implies $\Phi(e)$ or $d \leq' e$,
(P5B) if d and e are actions in conflict then $d \leq' e$,
(P5C) if d is an acquire or e is a release then $d \leq' e$, and
(P5D) if d is an SC write and e is an SC read then $d \leq' e$.

We then define:

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

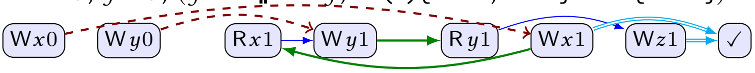
The main work happens in the definition of prefixing ($\Rightarrow$). P1 introduces a "new" event. Coalescing is allowed, as it was for composition (§2.4): the new event d may coalesce with an "old" one in E. P2 ensures that no order is removed from old events. P3 specifies the actions labelling the events. P4 specifies the preconditions (along with P5A). P5 specifies the preserved program order.

The semantics of read and write introduce a separate pomset for each possible value read or written. The value is fixed in each pomset. For writes, the dependence on M appears in the precondition $(M = v)$. This precondition must be satisfied using the substitutions in the semantic rules and P4C. Whereas writes introduce a precondition that must be satisfied sequentially, reads introduce a fulfillment requirement (§2.7) that must be satisfied concurrently.

The possibilities for fulfillment are limited by the program order that is preserved by P5. Requirement P5A connects the sequential semantics of write to the concurrent semantics of read; it requires order from read to write when the precondition of the write is *weakened* using P4C. The other requirements in P5 are standard. P5B captures the extended coherence requirement on sequential actions that touch the same location. P5C imposes the order required by acquire and release actions. P5D imposes the additional order required by SC actions. (Recall that SC reads are acquires, and that SC writes and termination actions are releases.)

We explain the concurrent semantics in the next section using standard litmus tests. In this subsection, we focus on the sequential semantics. Let us revisit (†):

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



(†)

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

$r{=}0 \mid \mathsf{W}x0$ $r{=}0 \mid \mathsf{W}z0$ $r{=}1 \mid \mathsf{W}x1$ $r{=}1 \mid \mathsf{W}z1$ $r{=}0 \mid \mathsf{W}x0$ $r{=}1 \mid \mathsf{W}z1$

Consistency (Def. 2.2) rules out the rightmost pomset, since the conjunction of preconditions is unsatisfiable. No order is required between the writes. Combining the middle pomset with $(r{=}0 \mid \mathsf{W}x1)$, the conditional $[\![\texttt{if}(r)\{x\,{:=}\,r;\ z\,{:=}\,r\}\,\texttt{else}\,\{x\,{:=}\,1\}]\!]$ contains:

$$\boxed{r{=}1 \lor r{=}0 \mid \mathsf{W}x1} \quad \boxed{r{=}1 \mid \mathsf{W}z1}$$

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

$$\boxed{y{=}1 \lor y{=}0 \mid \mathsf{W}x1} \quad \boxed{y{=}1 \mid \mathsf{W}z1}$$

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

$$\boxed{\mathsf{R}y1} \quad \boxed{(y{=}1 \lor y{=}0) \land (1{=}1 \lor 1{=}0) \mid \mathsf{W}x1} \quad \boxed{(y{=}1) \land (1{=}1) \mid \mathsf{W}z1}$$

The second conjunct in each event is required by P4C. P4C also prevents inconsistent reads such as:

$$\boxed{\mathsf{R}y2} \quad \boxed{(y{=}1 \lor y{=}0) \land (2{=}1 \lor 2{=}0) \mid \mathsf{W}x1} \quad \boxed{(y{=}1) \land (1{=}1) \mid \mathsf{W}z1}$$

P4C also allows predicates to weaken, in which case P5A requires order:

$$\begin{tikzcd}
\boxed{\mathsf{R}y1} \arrow[rr, bend left] & \boxed{(y{=}1 \lor y{=}0) \land (1{=}1 \lor 1{=}0) \mid \mathsf{W}x1} & \boxed{(1{=}1) \land (1{=}1) \mid \mathsf{W}z1}
\end{tikzcd}$$

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

$$\begin{tikzcd}
\boxed{0{=}0 \mid \mathsf{W}y0} \arrow[r, dashed] & \boxed{\mathsf{R}y1} \arrow[rr, bend left] & \boxed{(0{=}1 \lor 0{=}0) \land (1{=}1 \lor 1{=}0) \mid \mathsf{W}x1} & \boxed{(1{=}1) \land (1{=}1) \mid \mathsf{W}z1}
\end{tikzcd}$$

Simplifying the tautologies, we have:

$$\begin{tikzcd}
\boxed{\mathsf{W}y0} \arrow[r, dashed] & \boxed{\mathsf{R}y1} \arrow[rr, bend left] & \boxed{\mathsf{W}x1} & \boxed{\mathsf{W}z1}
\end{tikzcd}$$

As discussed in the introduction of §1, this reasoning fails in OOTA1–OOTA4.

For OOTA1, $[\![x\,{:=}\,r;\ z\,{:=}\,r]\!]$ contains:

$$\boxed{r{=}1 \mid \mathsf{W}x1} \quad \boxed{r{=}1 \mid \mathsf{W}z1}$$

It is only possible to satisfy the precondition $(r{=}1)$ using P4C when prefixing $(\mathsf{R}y1)$. This forces a dependency from read to write via P5A. OOTA2 is similar.

For OOTA3, $[\![\texttt{if}(r)\{x\,{:=}\,r;\ z\,{:=}\,r\}\,\texttt{else}\,\{x\,{:=}\,2\}]\!]$ contains:

$$\boxed{r{=}1 \mid \mathsf{W}x1} \quad \boxed{r{=}1 \mid \mathsf{W}z1} \qquad\qquad \boxed{r{=}2 \lor r{=}0 \mid \mathsf{W}x2} \quad \boxed{r{=}2 \mid \mathsf{W}z2}$$

Coalescing is possible when writing 2, as on the right above, but it is not possible when writing 1, as on the left, since $(\mathsf{W}x1)$ and $(\mathsf{W}x2)$ are different actions. When writing 1, this is no different than OOTA1, and thus P5A forces a dependency from the read of y to the writes.

For OOTA4, $[\![\texttt{if}(b)\{x\,{:=}\,r;\ z\,{:=}\,r\}\,\texttt{else}\,\{x\,{:=}\,1\}]\!]$ contains:

$$\boxed{\mathsf{R}b1} \quad \boxed{\big((r{=}1 \land b{\neq}0) \lor b{=}0\big) \land \big((r{=}1 \land 1{\neq}0) \lor 1{=}0\big) \mid \mathsf{W}x1} \quad \boxed{(r{=}1 \land b{\neq}0) \land (r{=}1 \land 1{\neq}0) \mid \mathsf{W}z1}$$

In an execution that reads $b{=}1$, P5A again forces a dependency from the read of y to the writes.

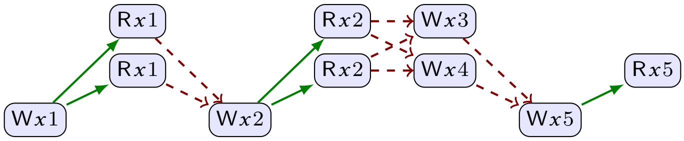
2.7 Fulfillment, Local Declarations, and Top-Level Pomsets

As in our previous work [Disselkoen et al. 2019], at the point that x is bound, we require that every read of x be *fulfilled*. Fulfillment plays the role that *reads-from* (rf) and *coherence* (co) play in other relaxed memory models. Unlike the acyclicity requirements involving rf and co in other models, however, fulfillment is defined *compositionally*—see example BLOCKER, below.

Definition 2.7. 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 $e \leq c$.

F3 requires that a write d is ordered before any read e that it fulfills; this order is typically called *reads from*. 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:

$$x := 1 \parallel x := 2 \parallel x := 3 \parallel x := 4 \parallel x := 5 \parallel r := x; r := x; r := x; r := x; r := x$$



(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, $(\mathsf{W}x1) \le (\mathsf{W}x2) \le (\mathsf{W}x5)$. 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 $(\mathsf{W}x3)$ and $(\mathsf{W}x4)$. But irrelevant writes must be ordered with respect to relevant writes and reads. The resulting order is somewhat weaker than traditional extended coherence (eco), which requires a total order on writes, regardless of whether they are relevant. We discuss coherence further in §3.1.

In order to be x-closed, a pomset must be "done" with x, in both the concurrent and the sequential semantics. The concurrent semantics requires that all reads of x be fulfilled. The sequential semantics requires that all formulae be *independent of* x (Def. 2.1): $\forall v.\ \phi \vDash \phi[v/x] \vDash \phi$.

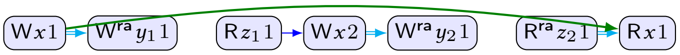
Definition 2.8. A pomset is x*-closed* if every read on x is fulfilled, and every formula is independent of x. Let $P \in (\nu x.\mathcal{P})$ when $P \in \mathcal{P}$ and P is x-closed. We define:

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

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

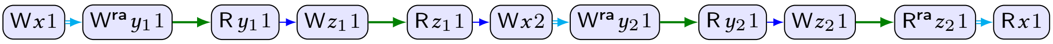
Together with F4, the definition of x-closed disallows the following execution:

$$\mathsf{var}\, x;\ (x := 1;\ y_1^{\mathsf{ra}} := 1 \parallel \mathsf{if}(z_1)\{x := 2\};\ y_2^{\mathsf{ra}} := 1 \parallel r := z_2^{\mathsf{ra}};\ s := x)$$



(BLOCKER)

In order to close x, we must choose whether $(\mathsf{W}x2)$ is preceding $(\mathsf{W}x2 \dashrightarrow \mathsf{W}x1)$ or following $(\mathsf{R}x1 \dashrightarrow \mathsf{W}x2)$. This prevents $(\mathsf{W}x2)$ from blocking the read after parallel composition. For example, if BLOCKER were placed in the context $b := a \parallel d := c \parallel [-]$, we would have:



This violates the conventional, weaker statement of F4: there is no conflicting write c such that $d < c < e$. By requiring order on $(\mathsf{W}x2)$ we forbid this blocker and validate scope extrusion (§3.2).

3 PROPERTIES OF THE BASIC MODEL

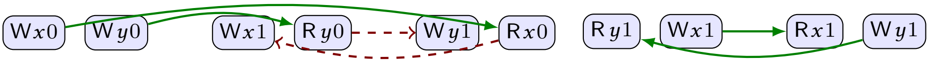
It is amazing how much the semantics of §2 "gets right" out of the box, including value range analysis, internal reads, and SC access, all of which can be complex in other models. In this section, we walk through several litmus tests, valid rewrites and invalid rewrites. The examples show that P5A–P5D and F3–F4 are understandable as *general principles*. The interaction of these principles is limited to a single, global, pomset order. We discuss tweaks to the semantics in §4.

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 language. Our model also agrees with the OOTA examples of 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:

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

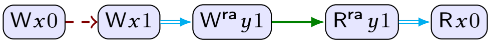


(SB/LB)

Because there are no intra-thread dependencies, the desired outcomes are allowed, as shown.

Publication. F3–F4 and P5B–P5C ensure correct publication, prohibiting stale reads:

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



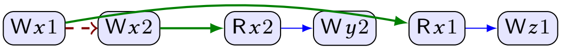
(PUB1)

This pomset is disallowed, since $(\mathsf{R}x0)$ fails to satisfy F4: $(\mathsf{W}x0) < (\mathsf{W}x1) < (\mathsf{R}x0)$. Attempting to satisfy this requirement, one might order $(\mathsf{R}x0)$ before $(\mathsf{W}x1)$, but this would create a cycle.

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 SC-DRF theorem (unlike Java).

Since reads are not ordered by P5B, we allow the following unintuitive behavior. C11 includes read-read coherence between relaxed atomics in order to forbid this:

$x := 1;\ x := 2 \parallel y := x;\ z := x$

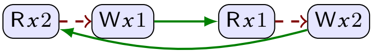


(CO2)

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 without requiring aliasing analysis.

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

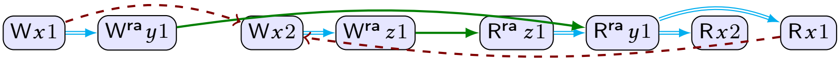
$r := x;\ x := 1 \parallel s := x;\ x := 2$



(TC16)

We also forbid the following, which Java allows:

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

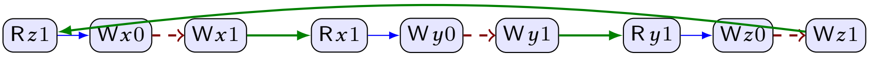


(CO3)

The order from $(\mathsf{R}x1)$ to $(\mathsf{W}x2)$ is required to fulfill $(\mathsf{R}x1)$. The outcome is disallowed due to the cycle. If this outcome were allowed, then racing writes would be visible, even after a full synchronization; this would invalidate local reasoning about data races (§8).

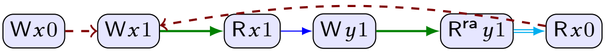
MCA. We present a few examples that are hallmarks of MCA architectures.

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



(MCA1)

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



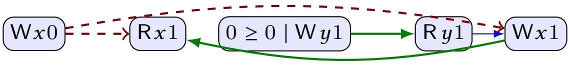
(MCA2)

These candidate executions are invalid, due to cycles. MCA1 is an example of *write subsumption* [Pulte et al. 2018, §3]. In MCA2, $(\mathsf{W}x1)$ is delivered to the second thread, but not the third; this is similar to the well know IRIW (Independent Reads of Independent Writes) litmus test, which is also disallowed by MCA architectures if the reads within each thread are ordered.

If y^{ra} is changed to y^{rlx} in MCA2, 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 example ADDR2 (§10) for further discussion.

Internal Reads and Value Range Analysis. 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 execution should be allowed, as it is in our model:

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

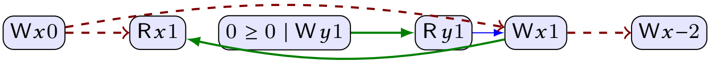


(TC1)

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

Our semantics is robust with respect to the introduction of concurrent writes, as in TC9:

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



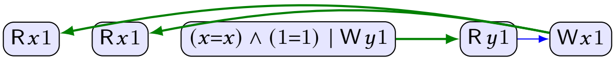
(TC9)

The calculation of this pomset is unchanged from TC1.

Examples such as TC9 present substantial difficulties in other models. When thought of in terms of compiler optimizations, TC9 is justified by global value analysis in collusion with the thread scheduler. This execution is disallowed by our event structure model [Jeffrey and Riely 2016]. It is allowed by Pichon-Pharabod and Sewell [2016], at the cost of introducing *dead reads*.

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

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

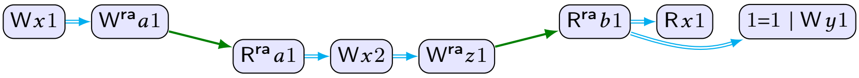


(TC2)

Note that in $[\![s:=x;\ \mathtt{if}(r=s)\{y:=1\}]\!]$, the precondition on $(\mathsf{W}y1)$ must imply $(r=x \wedge r=1)$. The first is imposed by P5A, the second by P4C, ensuring that the two reads see the same value.

Using ARM8 terminology, these executions involve *internal reads*, which are fulfilled by a sequentially preceding write. Read actions always generate an event that must be fulfilled, and therefore cannot be ignored, even if they are unused. This fact prevents internal reads from ignoring concurrent blocking writes.

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

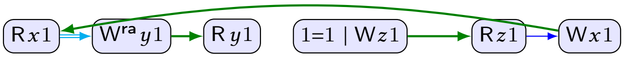


(INTERNAL1)

Here $(\mathsf{R}x1)$ violates F4. The precondition $(1=1)$ is imposed by P4C. The pomset becomes inconsistent if we change $(\mathsf{R}x1)$ to $(\mathsf{R}x2)$, since the precondition would change to $(2=1)$.

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

$$r := x;\ y^{\mathsf{ra}} := 1;\ s := y;\ z := s \parallel x := z \qquad \text{(INTERNAL2)}$$



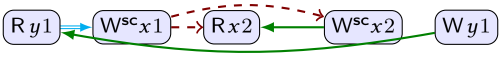
This behavior is allowed in our model, as it is in ARM8. Note that $[\![z := s]\!]$ includes $(s{=}1 \mid \mathsf{W}z1)$. Prepending a read, $[\![s := y;\ z := s]\!]$ may update the precondition to $(y{=}1 \mid \mathsf{W}z1)$ without introducing order. Further prepending $(\mathsf{W}^{\mathsf{ra}}y1)$ 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 ARM8 model has to jump through some hoops to ensure that internal reads are handled correctly. ARM8 takes the symmetric approach: rather than dropping order *out of* an internal read, ARM8 drops the order *into* it. This difference complicates the proof of correctness for implementing our semantics on ARM8 (§7).

SC access. P5D ensures that program order between SC operations is always preserved. Combined with F3–F4, 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 all actions are sc. It is also immediate that SC actions can be totally ordered, using any linearization of pomset order. Just as SC access in ARM8 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, since there is no order from $(\mathsf{W}^{\mathsf{sc}}x2)$ to $(\mathsf{W}y1)$—recall that SC writes are *releases*.

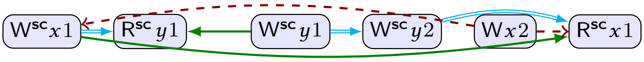
$$r := y;\ x^{\mathsf{sc}} := 1;\ s := x \parallel x^{\mathsf{sc}} := 2;\ y := 1 \qquad \text{(SC1)}$$



This execution is disallowed by Dolan et al. [2018, §8.2], preventing them from using `stlr` to implement volatile writes on ARM8. Our implementation strategy does use `stlr` for SC writes, as is standard. For further discussion, see examples PAST and FUTURE in §8.

Watt et al. [2020, §3.1] noticed a similar difficulty in Javascript [ECMA International 2019, §27]:

$$x^{\mathsf{sc}} := 1;\ r := y^{\mathsf{sc}} \parallel y^{\mathsf{sc}} := 1;\ y^{\mathsf{sc}} := 2;\ x := 2;\ s := x^{\mathsf{sc}} \qquad \text{(SC2)}$$



This execution is allowed both by our semantics and by ARM8 (using `stlr` for SC writes and `ldar` for SC reads). However, it is not allowed by Javascript 2019. In Javascript, the rules relating SC and relaxed access are subtle. As result of these interactions, Javascript 2019 fails to satisfy SC-DRF [Watt et al. 2019]. The rules are even more complex in C11; see SC3 and SC4 in §5 for a discussion of SC fences in C11. In our model, only P5D is required to explain SC access.

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 $\mathsf{id}(C)$ be the set of locations and registers that occur in C.

The semantics validates many peephole optimizations. Most apply only to relaxed access.

$$[\![r := x;\ s := y;\ C]\!] = [\![s := y;\ r := x;\ C]\!] \qquad \text{if } r \neq s \qquad \text{(RR)}$$

$$[\![x := M;\ y := N;\ C]\!] = [\![y := N;\ x := M;\ C]\!] \qquad \text{if } x \neq y \qquad \text{(WW)}$$

$$[\![x := M;\ s := y;\ C]\!] = [\![s := y;\ x := M;\ C]\!] \qquad \text{if } x \neq y \text{ and } s \notin \mathsf{id}(M) \qquad \text{(RW)}$$

P5 imposes no order between events in RR–RW. Using augmentation closure, P5 also validates roach-motel reorderings [Sevčík 2008]. For example, on read/write pairs:

$$[\![x^{\mu} := M;\ s := y;\ C]\!] \supseteq [\![s := y;\ x^{\mu} := M;\ C]\!] \qquad \text{if } x \neq y \text{ and } s \notin \mathsf{id}(M) \tag{ROACH1}$$

$$[\![x := M;\ s := y^{\mu};\ C]\!] \supseteq [\![s := y^{\mu};\ x := M;\ C]\!] \qquad \text{if } x \neq y \text{ and } s \notin \mathsf{id}(M) \tag{ROACH2}$$

Redundant load elimination (RL) follows from P1, taking $d \in E$, regardless of the access mode:

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

Since P5B 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:

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

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

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

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

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]:

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

Invalid Rewrites. The definition of location binding does not validate renaming of locations: if $x \neq y$ then $[\![\mathsf{var}\,y;\ C]\!] \neq [\![\mathsf{var}\,x;\ C[x/y]]\!]$, 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 given execution is impossible if the first thread is split, as in $[\![r := x;\ y^{\mathsf{ra}} := 1 \parallel s := y;\ z := s \parallel x := z]\!]$. The write in the first thread cannot discharge the precondition in the second, now separate.

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

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

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: $[-] \parallel r := x;\ x := 2;\ s := x;\ \mathtt{if}(r{=}s)\{z := 1\}$.

4 CASE ANALYSIS, ACCESS ELIMINATION AND READ INTRODUCTION

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

We consider the following optimizations on relaxed access: case analysis (CA), dead store elimination (DS), store forwarding (SF), read elimination (RE), and irrelevant read introduction (RI). We do not attempt to validate rewrites that eliminate ra/sc accesses, beyond those already given.

Definition 4.1. Extend the definition of prefixing (Cand. 2.6) to require:

(P6) if d is a release, e_1 is an acquire, $e_1 \leq e_2$, then $\Phi(e_2)$ is location independent.

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

Let $P' \in (\mathsf{weaken}_{\mathsf{R}xv} \mathcal{P})$ when there is $P \in \mathcal{P}$ such that $E' = E$, $\leq' = \leq$, $\mathcal{A}' = \mathcal{A}$, and either $\Phi'(e)$ implies $\Phi(e)$ or e is $\leq$-minimal[3] and $\mathcal{A}(e) = \mathsf{R}xv$.

$$[\![r := x^{\mathsf{rlx}}; C]\!] \triangleq [\![C]\!][x/r] \cup \textstyle\bigcup_v (\mathsf{R}^{\mathsf{rlx}} x v) \Rightarrow \mathsf{weaken}_{\mathsf{R}xv} [\![C]\!][x/r]$$

$$[\![x^{\mathsf{rlx}} := M; C]\!] \triangleq \mathsf{cover}_x [\![C]\!][M/x] \cup \textstyle\bigcup_v (M = v \mid \mathsf{W}^{\mathsf{rlx}} x v) \Rightarrow [\![C]\!][M/x]$$

$$[\![r := x^{\mu}; C]\!] \triangleq \textstyle\bigcup_v (\mathsf{R}^{\mu} x v) \Rightarrow [\![C]\!][x/r] \qquad \text{if } \mu \neq \mathsf{rlx}$$

$$[\![x^{\mu} := M; C]\!] \triangleq \textstyle\bigcup_v (M = v \mid \mathsf{W}^{\mu} x v) \Rightarrow [\![C]\!][M/x] \qquad \text{if } \mu \neq \mathsf{rlx}$$

There are four changes in the definition: To validate read elimination, we include $[\![C]\!][x/r]$. To ensure that read elimination does not allow stale reads, we require P6. To validate write elimination, we include $\mathsf{cover}_x [\![C]\!][M/x]$. To validate case analysis, we apply $\mathsf{weaken}_{\mathsf{R}xv}$ before prefixing a read.

We close this section with a read-enriched semantics that validates irrelevant read introduction.

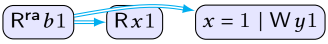
Read Elimination and Store Forwarding. In our work on microarchitecture [Disselkoen et al. 2019], read actions could be observed using cache effects. Candidate 2.6 maintains this perspective—for example, it distinguishes $[\![r := x]\!]$ and $\supseteq [\![\mathtt{skip}]\!]$ even though there is no context in the language of this paper that can distinguish these programs. If one accepts that these programs should be equated at an architectural level, then one would expect the semantics to validate read elimination (RE) and store forwarding (SF).

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

$$[\![x^{\mu} := M; r := x; C]\!] \supseteq [\![x^{\mu} := M; r := M; C]\!] \tag{SF}$$

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 internal read examples from §3.1. With read elimination, the read action $(\mathsf{R}y1)$ 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 this non-execution into a valid execution, violating SC-DRF. The addition of P6 to the definition of prefixing prevents this outcome. When computing $[\![x := 1; a^{\mathsf{ra}} := 1; \mathtt{if}(b^{\mathsf{ra}})\{y := x\}]\!]$, P6 prevents prefixing $(\mathsf{W}^{\mathsf{ra}} a1)$ in front of:



In order to satisfy P6, the precondition of $(\mathsf{W}y1)$ must be location independent.

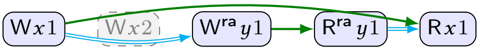
Dead Store Elimination. Dead store elimination (DS) is symmetric to redundant load elimination.

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

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 "$[-] \parallel z := x$".

Using cover_x, DS is validated by Definition 4.1. A write may only be removed if it is *covered* by a following write. This restriction is sufficient to prevent bad executions such as:

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



In this diagram, we have included a "non-event"—dashed border—to mark the eliminated write. In general, there may need to be many following writes, one for each subsequent release.

[3] e is *$\leq$-minimal* if there is no d such that $d \leq e$.

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''.

Disjunction closure is sufficient to establish case analysis (CA):

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

Candidate 2.6 is not disjunction closed. For example, consider the two sides of the composition defined by the conditional, where $C_{\mathsf{R}} = r\!:=\!x;\ \mathtt{if}(M)\{s\!:=\!x\}$.

$$\mathtt{if}(N)\{C_{\mathsf{R}}\} \qquad\qquad \mathtt{if}(\neg N)\{C_{\mathsf{R}}\}$$

$$d\colon \boxed{N \mid \mathsf{R}x0} \quad e\colon \boxed{N \wedge M \mid \mathsf{R}x0} \qquad\qquad e\colon \boxed{\neg N \mid \mathsf{R}x0} \quad d\colon \boxed{\neg N \wedge M \mid \mathsf{R}x0}$$

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

$$\mathtt{if}(N)\{C_{\mathsf{R}}\}\,\mathtt{else}\,\{C_{\mathsf{R}}\}$$

$$d\colon \boxed{N \vee (\neg N \wedge M) \mid \mathsf{R}x0} \quad e\colon \boxed{(N \wedge M) \vee \neg N \mid \mathsf{R}x0}$$

which is:

$$d\colon \boxed{N \vee M \mid \mathsf{R}x0} \quad e\colon \boxed{\neg N \vee M \mid \mathsf{R}x0}$$

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

$$d\colon \boxed{\mathsf{R}x0} \quad e\colon \boxed{\mathsf{R}x0}$$

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 *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 P5B.

If P5B is strengthened to include read-read coherence, then disjunction closure holds without $\mathsf{weaken}_{\mathsf{R}xv}$. In this case, however, CSE fails. This compromise may be reasonable for C11 atomics, which are meant to be used sparingly. It is less attractive for safe languages, like Java.

Irrelevant Read Introduction. A compiler may introduce reads in order to lift code. Consider the following example [Sevčík 2008, §1.4.5]:

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

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 $(\mathsf{R}xv)$ to the left-hand side, validating the inclusion.

Let $P' \in \mathsf{read}(\mathcal{P})$ when $\exists P \in \mathcal{P}$ and $\exists D$ such that $E' = E \uplus D$, $\leq' \supseteq \leq$, $\lambda' \supseteq \lambda$, and $\forall d \in D.\ \exists x.\ \exists v.\ \mathcal{A}'(d) = (\mathsf{R}xv)$. 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.

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

With RI, the model satisfies all of the transformations of Sevčík [2008, §5.3-4] except redundant write after read elimination (see §10) and reordering with external actions, which we do not model.

5 FENCES, READ-MODIFY-WRITE, AND ADDRESS CALCULATION

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 "$\mathsf{F}^{\nu}; C$" have corresponding actions: (F^{ν}). The *syntactic fence mode* ($\nu ::= \mathsf{rel} \mid \mathsf{acq} \mid \mathsf{sc}$) 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.

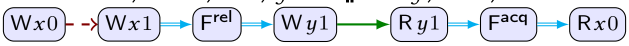
$$\llbracket \mathsf{F}^{\nu}; C \rrbracket \triangleq (\mathsf{F}^{\nu}) \Rightarrow \llbracket C \rrbracket$$

With no further changes, syntactic fences would impose exactly the same order as synchronization actions. This is sufficient to simulate sc accesses, since sc fences are very strong. However, it is insufficient to simulate ra accesses. Thus we add the following requirements to Definition 2.6:

(P5E) if d reads, and e is an acquiring fence, then $d \leq' e$, and
(P5F) if d is a releasing fence, and e writes, then $d \leq' e$.

Consider the following variant of PUB1:

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



(PUB2)

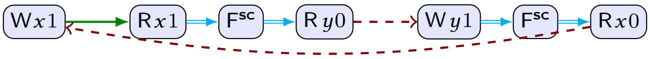
P5F requires that $(\mathsf{F}^{\mathsf{rel}}) \leq (\mathsf{W}y1)$. P5E requires that $(\mathsf{R}y1) \leq (\mathsf{F}^{\mathsf{acq}})$. The other order involving fences is required by P5C. The attempted execution is *invalid*: the stale read $(\mathsf{R}x0)$ violates F4.

As for RL, redundant fence elimination (RF) follows from P1, regardless of the access mode.

$$\llbracket \mathsf{F}^{\nu}; \mathsf{F}^{\nu}; C \rrbracket \supseteq \llbracket \mathsf{F}^{\nu}; C \rrbracket \tag{RF}$$

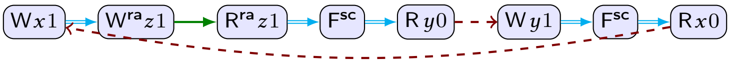
Our semantics does not suffer from overly weak fencing (SC3) or a lack of fence cumulativity (SC4). The following examples, from Lahav et al. [2017, Figs. 5 and 6], are allowed by both the original C11 and the model of Batty et al. [2016]. We omit 0-initialization in these examples:

$$x := 1 \parallel r := x;\ \mathsf{F}^{\mathsf{sc}};\ r := y \parallel y := 1;\ \mathsf{F}^{\mathsf{sc}};\ r := x$$



(SC3)

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



(SC4)

The executions are disallowed, due to the evident 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.

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.

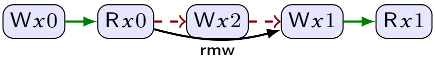
In Definition 2.2, we require that a *(memory model) pomset* be a tuple $(E, \leq, \lambda, \mathsf{rmw})$, where $\mathsf{rmw} \subseteq\ \leq$ relates the two events of a successful RMW. Additionally, we require that:

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

Other than these additions, nothing else changes. In particular, RMWs require no special treatment in Definition 2.4: 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]:

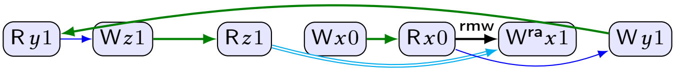
$$x := 0;\ s := \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x) \parallel x := 2;\ s := x$$



(RMW1)

By using two actions rather than one, the definition allows examples such as the following, which is allowed by ARM8 [Podkopaev et al. 2019, Ex. 3.10]:

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



(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:

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

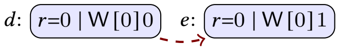
Address calculation can be encoded using the conditional. We give the semantics simply by expanding this encoding. Applying this technique to Candidate 2.6, we arrive at the following:

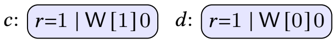
$$\llbracket r := [L]^{\mu};\ C \rrbracket \triangleq \|_{\ell}\ (L = \ell) \rhd \left(\bigcup_{v}\ (\mathsf{R}^{\mu}[\ell]v) \Rightarrow \llbracket C \rrbracket[[\ell]/r]\right)$$
$$\llbracket [L]^{\mu} := M;\ C \rrbracket \triangleq \|_{\ell}\ (L = \ell) \rhd \left(\bigcup_{v}\ (M = v \mid \mathsf{W}^{\mu}[\ell]v)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

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.

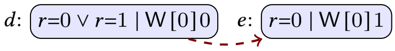
The rewrites listed in §3-4 remain valid, with the following generalization: For address expressions $[M]$ and $[N]$, replace $x = y$ by provable equality of M and N, and $x \neq y$ by provable inequality.

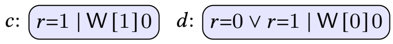
In Definition 4.1, we were able to ensure disjunction closure by performing targeted, local weakening via $\mathsf{weaken}_{\mathsf{R}xv}$. This is much more difficult to do with address calculation, since a single bit of syntax can refer to multiple locations. Consider that $\llbracket [r] := 0;\ [0] := !r \rrbracket$ 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:

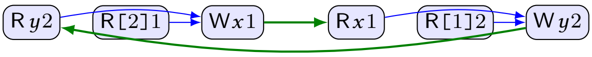




In this example, the d events that coalesce come from inconsistent executions. This is possible because the d events originate from different commands.

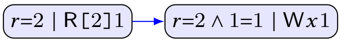
Because we do not enforce order between reads, there is some danger that address calculations could introduce anomalous behaviors that arise *out of thin air* (OOTA). Consider the following program, where initially $x = 0$, $y = 0$, $[0] = 0$, $[1] = 2$, and $[2] = 1$. It should only be possible to read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

$$r := y;\ s := [r];\ x := s \parallel r := x;\ s := [r];\ y := s$$



(ADDR1)

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 $[\![s := [r]\,;\ x := s]\!]$:

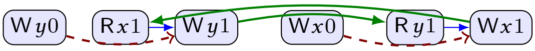


The precondition (r=2 | Wx1) is required by causal strengthening (Def. 2.2).

6 UNDERSTANDING "OUT OF THIN AIR" 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 OOTA1 from §1. Here we inline initialization in order to fit the format of our proof rules:

$y := 0;\ y := x \parallel x := 0;\ r := y;\ x := r$



(OOTA1)

Although Java does not allow OOTA behaviors of OOTA1, Lochbihler [2013] showed that it does allow OOTA behaviors of OOTA5, also from §1. In [Jeffrey and Riely 2016], we described a logic that rules out OOTA1 but not OOTA5 or its variant OOTA4. In this section, we provide a more accurate test of OOTA behaviors by enhancing our previous logic with temporal features.

On first read, we suggest that readers skip to the examples and the discussion that follows, coming back to the definitions as necessary. Example 6.2 discusses the canonical OOTA example OOTA1; the analysis is trivial and well-known [Jeffrey and Riely 2016; Kang et al. 2017]. Example 6.3 is more interesting. There, we discuss OOTA4, which is a variant of Lochbihler's OOTA5.

The logic given here is not meant to be definitive; in §10, we discuss OOTA examples that appear to 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:

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

Define false, $\vee$, and $\Rightarrow$ as usual.
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 $(\boxminus\varrho \Rightarrow \diamondminus\varrho)$. The order-minimal elements always validate $\boxminus\varrho$ and invalidate $\diamondminus\varrho$. However, we can prove the following:

$$P \models (\boxminus\varrho \Rightarrow \varrho) \Rightarrow \varrho \qquad \text{(Induction)}$$
$$P \models (\varrho \Rightarrow \diamondminus\varrho) \Rightarrow \neg\varrho \qquad \text{(Coinduction)}$$
$$P \models (\varrho \Rightarrow \diamondminus\vartheta) \Rightarrow (\diamondminus\varrho \Rightarrow \diamondminus\vartheta) \qquad \text{(Weakening)}$$

We present two additional proof rules. The first provides a logical view of x-*closure* (Def. 2.7):

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

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:*

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

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 $P \models \varrho$. □

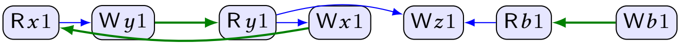
Example 6.2. With all variables initialized to 0, we show that OOTA1 satisfies $\neg \mathsf{W}x1$.
We start with the invariant:

$$[\mathsf{W}x1 \Rightarrow \diamondminus \mathsf{R}y1] \land [\mathsf{W}y1 \Rightarrow \diamondminus \mathsf{R}x1]$$

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

Example 6.3. Because our language lacks object creation, we cannot consider Lochbihler's example (OOTA5) directly. Instead we study OOTA4, which has the same temporal structure. The essential temporal property of OOTA4 is: *A write of* 1 *to y must be preceded by a read of* 1 *from x, and if* 1 *is written to z then a write of* 1 *to x must be preceded by a read of* 1 *from y.* We show an attempted execution that violates this invariant, eliding initialization:

```
y:=x ∥ r:=y; if(b){x:=r; z:=r}else{x:=1} ∥ b:=1
```



(OOTA4)

As we discussed in §2.6 there is a dependency from $(\mathsf{R}y1)$ to $(\mathsf{W}x1)$; thus, the outcome is disallowed. This outcome is also disallowed by our event structures model [Jeffrey and Riely 2019, §9], although the logic given in that paper is insufficient to establish this fact. The outcome is *allowed* by Manson et al. [2005], Jagadeesan et al. [2010], Kang et al. [2017], and Chakraborty and Vafeiadis [2019].

To establish that this outcome is disallowed here, we prove $\neg \mathsf{W}z1$, starting with invariant:

$$[\diamondminus \mathsf{W}y1 \Rightarrow \diamondminus \mathsf{R}x1] \land [\mathsf{W}z1 \Rightarrow (\diamondminus \mathsf{R}y1 \land \boxminus(\mathsf{W}x1 \Rightarrow \diamondminus \mathsf{R}y1))]$$

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

$$\mathsf{W}z1 \Rightarrow (\diamondminus \mathsf{R}x1 \land \boxminus(\mathsf{W}x1 \Rightarrow \diamondminus \mathsf{R}x1))$$

Closing x: $\mathsf{W}z1 \Rightarrow (\diamondminus \mathsf{W}x1 \land \boxminus(\mathsf{W}x1 \Rightarrow \diamondminus \mathsf{W}x1)$. Applying coinduction to the right conjunct:

$$\mathsf{W}z1 \Rightarrow (\diamondminus \mathsf{W}x1 \land \boxminus(\neg \mathsf{W}x1))$$

Simplifying: $\mathsf{W}z1 \Rightarrow \mathsf{false}$, as required.

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

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

The left command is half of OOTA1. 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? Because of the conditional, the write to x in RFUB is independent of the read from y. It useful to considering the Hoare logic formulas satisfied by the two threads above: we have $\{\mathsf{true}\}$ RFUB $\{x = 1\}$ for the right thread of RFUB, but not $\{\mathsf{true}\}$ OOTA1 $\{x = 1\}$ for the right thread of OOTA1. The change in the thread from OOTA1 to RFUB is not a valid refinement under Hoare logic; thus, it is expected that RFUB may have additional behaviors.

Understanding OOTA behavior is notoriously difficult, even for the greatest minds in the field! This example shows the wisdom of using existing tools, such as preconditions and Hoare logic, to model new problems, such as relaxed memory.

7 EFFICIENT IMPLEMENTATION ON ARM8

We show that our semantics compiles efficiently to ARM8 [Deacon 2017; Pulte et al. 2018] using the translation strategy of Podkopaev et al. [2019], which was extended to SC access by Moiseenko et al. [2019, §5]: Relaxed access is implemented using `ldr`/`str`, non-relaxed access using `ldar`/`stlr`, acquire and other fences using `dmb.ld`/`dmb.sy`.

We consider the fragment of our language where concurrent composition occurs only at top level and there are no local declarations of the form $(\mathsf{var}\, x;\, C)$. We show that any *consistent* ARM8 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 ARM8 acyclicity relation, and eco is the ARM8 extended coherence order. But this does not quite work.

The definition is complicated by ARM8's *internal reads*, manifest in rfi, which relates reads to writes that are fulfilled by the same thread. ARM8 drops ob-order *into* an internal read. As discussed in §3.1, however, our semantics drops pomset order *out of* an internal read. To accommodate this, we drop these dependencies from the ARM8 *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 $d \notin \mathsf{codom}(\mathsf{rfi})$, $e \notin \mathsf{codom}(\mathsf{rfi})$. Let ob' be defined as for ob, simply replacing dob with dob'.

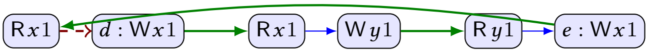
For pomset order, we then take $\leq\ = (\mathsf{ob}' \cup \mathsf{eco})^*$.

Theorem 7.1. *For any consistent* ARM8 *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].

It is worth noting that efficient compilation is not possible for the earlier Flowing and Pop model [Flur et al. 2016], referenced in [Lahav and Vafeiadis 2016, Fig. 4], which allows the following:

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



This type of "big detour" [Alglave et al. 2014] is outlawed by ARM8.[4]

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

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 there are no local declarations of the form $(\mathsf{var}\,x\,;\,C)$.

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 ($\le$). 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—see the discussion of CA in §4. In this case we include a separate pomset for each way of breaking these cycles.

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

Let $\mathsf{hb} = (\mathsf{po} \cup \mathsf{sw})^+$ be the *happens-before* relation. In PUB1, for example, $(\mathsf{W}x1)$ happens-before $(\mathsf{R}x0)$, 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 $d \overset{L}{\leftrightsquigarrow} e$) when they *conflict* (Def. 2.7) at some location in L, but are unordered by hb: neither $d \xrightarrow{\mathsf{hb}} e$ nor $e \xrightarrow{\mathsf{hb}} d$.

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

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 $\mathrm{gen}[\![C]\!] = \{P \in [\![C]\!] \mid P \text{ is } \textit{top-level} \text{ (Def. 2.8) and generation-minimal in } [\![C]\!]\}$.

Extensions. We say that P' *C-extends* P if $P \neq P' \in \mathrm{gen}[\![C]\!]$ 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: $E' = E$, $\Phi' = \Phi$, $\le'|_{E\setminus\{e\}} = \le|_{E\setminus\{e\}}$, 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) $P \in \mathrm{gen}[\![C]\!]$, (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 $d \overset{L}{\leftrightsquigarrow} e$. The empty pomset is L-stable.

Sequentiality. Let $\lessdot_L = <_L \cup\, \mathsf{po}$, 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) $d \overset{L}{\leftrightsquigarrow} e$, for some $d \in (E'' \setminus E)$.*

The theorem provides an inductive characterization of *Sequential Consistency for Local Data-Race Freedom (SC-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 $\le_*$ (the transitive reduction $\le$) can be decomposed into three disjoint relations. Let ppo $= (\le_* \cap$ po$)$ denote *preserved* program order, as required by prefixing (Def. 2.6). The other two relations are cross-thread subsets of $(\le_* \setminus$ po$)$, as required by fulfillment (Def. 2.7): wr orders writes before reads, satisfying fulfillment requirement F3; xw orders read and write accesses before writes, satisfying requirement F4. (Within a thread, F3 and F4 follow from prefixing requirement P5B, which is included in ppo.)

Using this decomposition, we can show the following.

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

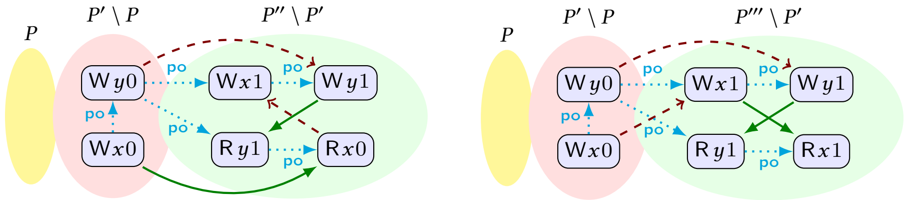
The proof of the lemma follows an inductive construction of gen$[\![C]\!]$, starting from a large set with little order, and pruning the set as order is added: We begin with all pomsets generated by the semantics without imposing the requirements of fulfillment (including only ppo). We then prune reads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

We can prove a similar result for (po $\cup$ 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 $<_L$ 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 $\cup$ 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 (Wx1).

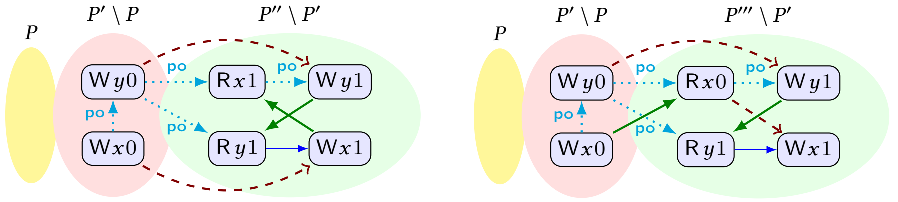
$$x := 0;\ y := 0;\ (x := 1;\ y := 1 \parallel \texttt{if}(y)\{r := x\})$$



It is important that e be (po $\cup$ wr)-maximal, not just (ppo $\cup$ 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 $\cup$ wr)-maximal access, then all cross-thread order must be from wr. In this case, we select a (ppo $\cup$ 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 (Wx1).

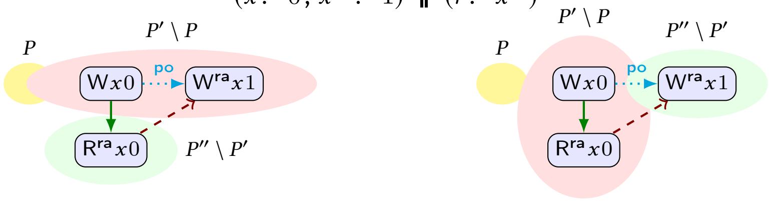
$$x := 0;\ y := 0;\ (r := x;\ y := 1 \parallel s := y;\ x := s)$$



This example requires $(\mathsf{W}x0)$. 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.

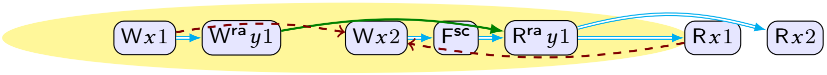
$$(x := 0;\ x^{\mathsf{ra}} := 1)\ \parallel\ (r := x^{\mathsf{ra}})$$


P
$P' \setminus P$
$\mathsf{W}x0$
po
$\mathsf{W}^{\mathsf{ra}}x1$
$\mathsf{R}^{\mathsf{ra}}x0$
$P'' \setminus P'$
P
$P' \setminus P$
$P'' \setminus P'$
$\mathsf{W}x0$
po
$\mathsf{W}^{\mathsf{ra}}x1$
$\mathsf{R}^{\mathsf{ra}}x0$

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, as in co3. As another example, consider the following, which is disallowed here, but allowed by Java [Dolan et al. 2018, Ex. 2]. We include an SC fence here to mimic the behavior of volatiles in the JMM.

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

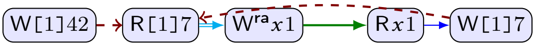

$\mathsf{W}x1$
$\mathsf{W}^{\mathsf{ra}}y1$
$\mathsf{W}x2$
F^{sc}
$\mathsf{R}^{\mathsf{ra}}y1$
$\mathsf{R}x1$
$\mathsf{R}x2$

(PAST)

The highlighted events are L-stable. The order from $(\mathsf{R}x1)$ to $(\mathsf{W}x2)$ is required by fulfillment, causing the cycle. If the fence is removed, 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, Ex. 3]:

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



(FUTURE)

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/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—in the inductive step, writes that are introduced can be ignored by existing reads, and reads that are introduced can be fulfilled, for some value, by some preceding write.

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]. Pulte et al. [2019] present an operational model of ARM8 in the style of [Kang et al. 2017].

In our previous work [Disselkoen et al. 2019], we introduced the notion of pomsets with preconditions. In 2019, we studied *micro-architecture*, where failed speculative execution is visible via cache effects. Here we presented an *architectural* model, which allows us to impose *consistency*, ignoring failed speculative execution, and *causal strengthening*. In 2019, we used 3-valued pomsets, as opposed to the simple pomsets used here; see §10 for further discussion. The previous paper was not focused on memory models, and thus did not prove the soundness of compiler optimizations, the absence of thin air reads, efficient implementability, or SC-DRF.

Our model shares important structural elements with that of Paviotti et al. [2020], who provide a fix for the OOTA problem in C11 relaxed access. Like us, they use true concurrency semantics to identify independencies 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. We only compose downsets of completed pomsets—this is mirrored by their condition on $\leq^{\mathcal{X}}$ during coproduct (discussed in their §6.3). Nonetheless, the papers have different goals, leading to different outcomes. For example, their model can be implemented efficiently on non-MCA architectures; our model clearly can not! Conversely, our model provides an intrinsic characterization of 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]. See [Jeffrey and Riely 2019, §8] for a discussion. A partial order approach to weak memory was sketched by Brookes [2016] and fleshed out for TSO by Kavanagh and Brookes [2019]. Their action labels include buffers, encoding the operational behavior of TSO inside the pomsets themselves.

There is 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—SC-DRF holds, but Sevčík [2011] discovered that this model 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-acquire accesses 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 have not modeled loops or recursive functions. These introduce complexities—such as liveness and continuity—that are orthogonal to the main topic of the paper.

We have not modeled general sequencing of the form $(C; C')$. The definition of prefixing (§2.6) is relatively simple since we only prepend one action at a time.

We have not modeled mixed-size access [Flur et al. 2017; Watt et al. 2020]. ARM8 captures multibyte access using multiple events related by *same instruction* (si) [Alglave 2019]. To capture the use of si when defining *locally ordered before* (lob), it is likely sufficient to modify P5B. To capture the use of si when defining *observation* (ob), it is likely sufficient to modify F3 and F4.

We have only considered peephole optimizations—other optimizations may be valid. For example, the model does *not* validate redundant write after read elimination as a peephole optimization. Consider this example from [Sevčík 2008, §5.3.1]:

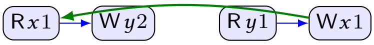
$$[\![r:=x;\ x:=r;\ s:=x;\ \texttt{if}(r{\neq}r)\{y:=1\}]\!] \not\supseteq [\![r:=x;\ s:=x;\ \texttt{if}(r{\neq}r)\{y:=1\}]\!]$$

In the JMM, these are distinguished by the context $x:=1;\ a^{\mathsf{ra}}:=1 \parallel x:=2;\ z^{\mathsf{ra}}:=1 \parallel \texttt{if}(a^{\mathsf{ra}} \land z^{\mathsf{ra}})$ $\{[-]\}$. That is not case here, however, due to local SC-DRF. Nonetheless, completed pomsets from the left include a write to x, which is not found on the right. This write to x cannot be eliminated using cover_x (Def. 4.1) because there is no following write. At top-level, however, the read of x must be fulfilled by *some* write—assuming that these writes can coalesce, the optimization can be validated.

We have not attempted validate all program transformations that involve synchronization, fences or RMWs [Morisset 2017; Vafeiadis et al. 2015]. Nonetheless, our semantics validates roach motel (ROACH1, ROACH2), redundant load (RL), fence removal (RF), and store forwarding (SF). We expect that dead store elimination (DS) generalizes to non-relaxed access by generalizing cover_x. Some transformations are not sound in the model, but we expect them to be provable as metaproperties. For example, access-mode strengthening (such as replacing rlx by ra) is valid up to an equivalence that ignores access modes on actions. Other transformations worth studying include commuting adjacent acquires, commuting adjacent releases, and implementing non-relaxed access using relaxed access and fences. Lock elision and access-mode weakening are also interesting, although these are only sound in certain contexts. We expect all of these transformations to be valid, given an appropriate notional of validity.

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:

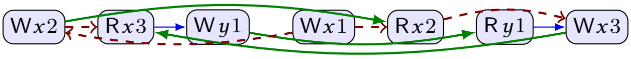
$$(y\mathbin{:=}x+1 \parallel x\mathbin{:=}y)$$



(OOTA6)

The attempted execution is *not* allowed by our semantics, since there is no write to fulfill (Ry1). As another example, consider the following, from Chakraborty and Vafeiadis [2019, Fig. 3]:

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



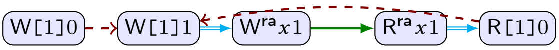
(OOTA7)

The attempted execution is *not* allowed by our semantics, due to the evident cycle. 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. Surprisingly, this outcome is allowed by the promising semantics [Kang et al. 2017]. Chakraborty and Vafeiadis developed WEAKESTMO to address this example; however, WEAKESTMO does not address OOTA4.

Our model realizes *multi-copy atomicity* (MCA). Thus it will not compile efficiently to non-MCA architectures, such as POWER and ARM7. To do so, one cannot include the order required by F4 in pomset order. In [Disselkoen et al. 2019], we modeled programs using *3-valued pomsets*, using the *weak* relation ($\nsucc$) for F4 and P5B. This does *not* provide a suitable model for non-MCA behavior, however, since it disallows MCA2.[5]

In the discussion of MCA2, 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. Consider:

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



(ADDR2)

If we changed x^{ra} to x^{rlx}, then there would be no order from (Rx1) to (R[1]0), and the execution would be allowed. In order to allow this relaxation in certain cases without invalidating the publication of the "field initializer" (W[1]1), it may be desirable to distinguish address dependencies from other dependencies—doing so would likely require two separate preconditions for each event.

[5] Following [Lamport 1986], 3-valued pomset require: (1) if $d \leq e$ then $d \nsucc e$, (2) if $d \leq e$ and $e \nsucc d$ then $d = e$, and (3) if $c \leq d \nsucc e$ or $c \nsucc d \leq e$ then $c \nsucc e$. MCA2 is disallowed by (2). Top-level 3-valued pomsets also require that $\nsucc$ is a partial order per-location; a sufficient condition is (4) if $d \nsucc e$ and $e \nsucc d$ then d and e do not touch the same location. Although MCA1 is allowed, its two-thread variant is not, due to the combination of *semi-transitivity (3)* and *partial coherence (4).*

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

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

Jade Alglave. 2019. This commit adds the Armv8 memory model for mixed-size accesses. https://github.com/herd/herdtools7/commit/95785c747750be4a3b64adfab9d5f5ee0ead8240.

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. 2015. *The C11 and C++11 concurrency model.* Ph.D. Dissertation. University of Cambridge, UK.

Mark Batty. 2017. Compositional relaxed concurrency. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (Oct. 2017), 13 pages. https://doi.org/10.1098/rsta.2015.0406

Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC atomics in C11 and OpenCL. 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, 634–648. https://doi.org/10.1145/2837614.2837637

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 283–307. https://doi.org/10.1007/978-3-662-46669-8_12

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* (Austin, Texas, USA) *(POPL '11)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

Hans-J. Boehm. 2007. Memory Model Rationales. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2176.

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

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* (Edinburgh, United Kingdom) *(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.pdf. 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)*. HAL-Inria, Saint-Malo, France, 39–53. 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, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4421)*, Rocco De Nicola (Ed.). Springer, 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*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 100–110. https://doi.org/10.5555/3049832.3049844

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

Will Deacon. 2017. Formal memory model for Armv8.0 application level. https://github.com/herd/herdtools7/commit/daa126680b6ecba97ba47b3e05bbaa51a89f27b7.

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*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 329–342. https://doi.org/10.1145/2429069.2429110

Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019.* IEEE, 1238–1255. 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* (Philadelphia, PA, USA) *(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

ECMA International. 2019. ECMAScript 2019 Language Specification. https://www.ecma-international.org/ecma-262/10.0/.

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

Shaked Flur, Susmit Sarkar, Christopher Pulte, Kyndylan Nienhuis, Luc Maranget, Kathryn E. Gray, Ali Sezgin, Mark Batty, and Peter Sewell. 2017. Mixed-size concurrency: ARM, POWER, C/C++11, and SC. 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, 429–442. https://doi.org/10.1145/3009837.3009839

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, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 307–326. https://doi.org/10.1007/978-3-642-11957-6_17

Alan Jeffrey and James 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), 25 pages. https://doi.org/10.23638/LMCS-15(1:33)2019

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek 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. http://dl.acm.org/citation.cfm?id=3009850

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2018. Private correspondence.

Ryan Kavanagh and Stephen Brookes. 2019. A Denotational Semantics for SPARC TSO. *Logical Methods in Computer Science* 15, 2 (2019), 23 pages. 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 (Lecture Notes in Computer Science, Vol. 9995)*, John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer, 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

Leslie 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. https://doi.org/10.5555/648065.747612

Andreas 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.

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

Jayadev Misra and K. Mani Chandy. 1981. Proofs of Networks of Processes. *IEEE Trans. Software Eng.* 7, 4 (1981), 417–426. https://doi.org/10.1109/TSE.1981.230844

Evgenii Moiseenko, Anton Podkopaev, Ori Lahav, Orestis Melkonian, and Viktor Vafeiadis. 2019. Reconciling Event Structures with Modern Multiprocessors. *CoRR* abs/1911.06567 (2019), 34 pages. arXiv:1911.06567 http://arxiv.org/abs/1911.06567 To appear in ECOOP 2020.

Robin Morisset. 2017. *Compiler optimisations and relaxed memory consistency models.* Ph.D. Dissertation. PSL Research University, Paris, France. https://tel.archives-ouvertes.fr/tel-01823521

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, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). 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* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 622–633. https://doi.org/10.1145/2837614.2837616

Gordon D. Plotkin and Vaughan R. Pratt. 1996. Teams can see pomsets. In *Partial Order Methods in Verification, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, July 24-26, 1996 (DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 29)*, Doron A. Peled, Vaughan R. Pratt, and Gerard J. Holzmann (Eds.). DIMACS/AMS, 117–128. https://doi.org/10.1090/dimacs/029/07

Amir Pnueli. 1984. In Transition From Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems - Conference proceedings, Colle-sur-Loup (near Nice), France, 8-19 October 1984 (NATO ASI Series, Vol. 13)*, Krzysztof R. Apt (Ed.). Springer, 123–144. https://doi.org/10.1007/978-3-642-82453-1_5

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL (2019), 69:1–69:31. https://doi.org/10.1145/3290382

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

William 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

Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 1–15. https://doi.org/10.1145/3314221.3314624

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* (San Jose, California, USA) *(PPoPP '07)*. ACM, New York, NY, USA, 161–172. https://doi.org/10.1145/1229428.1229469

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

Jaroslav Sevčík. 2011. Private correspondence.

Eugene W. Stark. 1985. A Proof Technique for Rely/Guarantee Properties. In *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings (Lecture Notes in Computer Science, Vol. 206)*, S. N. Maheshwari (Ed.). Springer, 369–391. https://doi.org/10.1007/3-540-16042-6_21

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, 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

Conrad Watt, Christopher Pulte, Anton Podkopaev, Guillaume Barbier, Stephen Dolan, Shaked Flur, Jean Pichon-Pharabod, and Shu-yu Guo. 2020. Repairing and mechanising the JavaScript relaxed memory model. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 346–361. https://doi.org/10.1145/3385412.3385973

Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening WebAssembly. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 133:1–133:28. https://doi.org/10.1145/3360559