Pomsets with Preconditions

A Simple Model of Relaxed Memory

ANONYMOUS AUTHOR(S)

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

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

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

ACM Reference Format:
Anonymous Author(s). 2020. Pomsets with Preconditions: A Simple Model of Relaxed Memory. 1, 1 (May 2020), 30 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

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

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

There are also two aspects to "ease of use." First, the *data race free-sequentially consistent* (DRF-SC) criterion [Adve and Hill 1990, 1993] permits the programmer to forget about relaxed memory for correctly synchronized programs. Second, all programs—even those with data races—should support compositional reasoning on temporal safety properties [Abadi and Lamport 1993; Misra and Chandy 1981; Pnueli 1985; Stark 1985].

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

In this paper, we combine two ideas that humans can understand: *preconditions* [Hoare 1969] and *labelled partial orders* (aka *pomsets*) [Gischer 1988; Plotkin and Pratt 1997]. The resulting model mostly satisfies the desiderata. We sacrifice only implementability on "non-MCA" processors, such as POWER and ARMv7. As a result, however, there is only one order relation to visualize.

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

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

A note.

XXXX-XXXX/2020/5-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

that $r_2 \neq s$, is it valid to transform the program on the left to that on the right?

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

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

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

$$\begin{array}{l} z := 1 \ \|\ y := x \\ \|\ r := y;\ \texttt{if}(z)\{s := \texttt{new C}\}\ \texttt{else}\ \{r := \texttt{new D}\};\ x := r \end{array} \tag{OOTA1}$$

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

The JMM, the promising semantics [Kang et al. 2017], and related models [Chakraborty and Vafeiadis 2019; Jagadeesan et al. 2010] all invalidate compositional reasoning, as in OOTA1. See §A for confirmation from the authors themselves. This means that these models *cannot support both type safety and realistic memory reclamation.*

The C++ 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). C++ *does* allow the transformation for *plain* access, but this comes with the threat of *undefined behavior* should any plain access ever possibly engage in a data race. Thus the folklore belief that "every substantial C++ program has undefined behavior."

Strong models, including Sequential Consistency (SC) [Lamport 1979], RC11 [Lahav et al. 2017], and others [Boehm and Demsky 2014; Dolan et al. 2018; Jeffrey and Riely 2016; Lahav et al. 2017] support compositional reasoning. However, all of these models invalidate reordering of independent statements. Several require fences after relaxed reads, even on ARMv8.

Our approach has two key ingredients.

First, we focus on *multi-copy atomicity* (MCA), which holds that 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. The key insight is that MCA *permits a single, global notion of time, manifest in the pomset order.* This is a dramatic simplification over other models.

Second, we weaken the program-order within a thread to capture only *essential dependencies.* These are represented in the pomset order. This reduction from program order to pomset order is similar to the reduction to *preserved* program order (ppo) in hardware models. However, rather than calculating dependencies syntactically, we compute them using classical Hoare logic. The key insight is that *logic is better than syntax.* Consider the following program fragments:

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

All these fragments satisfy $\{\text{true}\}\ C_i\ \{y = 1\}$; thus, in each case, the write of y is independent of any code that precedes it in program order. This allows a compiler or processor to reorder the write with respect to the code that precedes it.

We show that the model:

- captures all C11 concurrency features (§3),
- allows compositional reasoning for safety (§4),

- compiles to ARMv8 and TSO *without* extra synchronization for relaxed-atomic access (§5),
- satisfies the *local* DRF-SC criterion [Dolan et al. 2018] (§6), and
- validates single-threaded compiler optimizations (§7).

We conclude by discussing relating work (§8) and limitations (§9).

2 THE MODEL

We define the model and give the semantics of a concurrent language. We layer the presentation, beginning with a simple language that supports only read and write operations. In §3, we define extensions that incorporate address computation, fences, and read-modify-write operations.

2.1 Data models

A *data model* consists of:

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

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

We require that data models satisfy the following:

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

We use expressions as formulae, coercing M to $M \neq 0$.

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

We say that a is a *read* if $a \in \mathrm{dom}(\mathsf{Rd})$ and a is a *write* if $a \in \mathrm{dom}(\mathsf{Wr})$. When $\mathsf{Rd}(a) = (x, v)$, we say that a *reads* v *from* x, and similarly for writes.

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.

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

For the formulae of the data model, we say that ϕ is *independent of* x when, for every v, $\phi \vDash \phi[v/x] \vDash \phi$. We say that ϕ is *dependent on* x otherwise. We say that ϕ is *location independent* if it is independent of every location.

We say that ϕ *implies* ψ when $\phi \vDash \psi$, that ϕ is a *tautology* when $\mathsf{true} \vDash \phi$, and that ϕ is *unsatisfiable* when $\phi \vDash \mathsf{false}$.

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

2.2 Example Language

Our example language includes actions of the form $(\checkmark)$, which is a *termination*, $(\mathrm{R}^{\mu} x v)$, which *reads* v from x and $(\mathrm{W}^{\mu} x v)$, 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 elide the rlx-mode annotation in examples. (Note that C-style *plain* access is the same as relaxed access for race-free programs; for racy programs, plain access results in undefined behavior.)

We define the language by prefixing individual reads and writes.

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

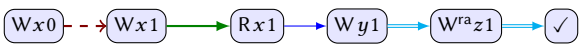
We use common syntax sugar, such as *extended expressions*, $\mathbb{M}$, which include memory locations. For example, if $\mathbb{M}$ includes a single occurrence of x, then $y := \mathbb{M}; C$ is shorthand for $r := x; y := \mathbb{M}[r/x]; C$. Each occurrence of x in an extended expression corresponds to an separate read.

We write $\mathtt{if}(M)\{C\}$ 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 $\mathtt{if}(M)\{C^1; D\}\mathtt{else}\{C^2; D\}$.

2.3 Semantic domain

We model *executions* as *completed pomsets with preconditions*, ranged over by P. We call these *pomsets*, for short. These extend the well-known model of partially ordered multisets [Gischer 1988] with formulae and a termination event. We model *programs* (notation $[\![C]\!]$) as *sets* of completed pomsets with preconditions, ranged over by $\mathcal{P}$.

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



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

Although we use multiple arrows, we emphasize that they are all part of the same $\leq$ relation.

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

The *preconditions* associated with events provide a *sequential requirement* for the event to execute. For example, the following commands gives rise to the pomset below them.

(∗)

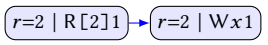
The pomsets on the left says that $y := r$ can produce the action $\mathrm{W}y1$ if $r = 1$, capturing the *data dependency* between register r and location y. The pomsets on the right says that $\mathtt{if}(r < 0)\{y := 1\}$ can produce the action $\mathrm{W}y1$ if $r < 0$, capturing the *control dependency* between r and y.

Pomsets are subject to two *top-level* restrictions: Every read must be *fulfilled*, and every precondition must be *satisfied*. Reads are fulfilled by writes, which may be concurrent (Definition 2.5). Preconditions can only be satisfied by substitutions, which can only derive from prefixing in the same thread (Definition 2.12). Thus reads give rise to *inter-thread* dependencies, whereas preconditions give rise to *intra-thread* dependencies.

There is a mapping between Hoare triples $\{\phi\}\ C\ \{\psi\}$ and the pomsets of C. The postcondition ψ represents a set of write actions, and ϕ represents their preconditions. For example, interpreting the pomset on the left this way yields the valid Hoare triple $\{r = 1\}\ y{:=}r\ \{y = 1\}$.

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

Preconditions are linked to pomset order via *causal strengthening*, which requires that formulae do not weaken over time, as measured by $\leq$. For example, the semantics of $s{:=}[r];\ x{:=}s$ includes:



The precondition on $\mathrm{W}x1$ is required by the data dependency induced by s (see page 13). Note that, by causal strengthening, the termination event of a completed pomset must imply all other events.

Definition 2.1. A *pomset with preconditions* is a tuple $(E, \leq, \lambda)$:

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

We write pairs in $(\Phi \times \mathcal{A})$ as $(\phi \mid a)$, eliding ϕ when it is a tautology.

We write $d < e$ when $d \leq e$ and $d \neq e$.

We lift terminology from logical formulae and actions to events, saying, for example, that e is *unsatisfiable* if $\Phi(e)$ is unsatisfiable, and that e is a *termination* when $\mathcal{A}(e)$ is a termination. We often elide explicit universal quantifiers in phrases such as "*for all* e, $\Phi'(e)$ implies $\Phi(e)$."

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

Definition 2.2. P' is a *downset* of P if $E \supseteq E' \supseteq \{d \in E \mid \exists e \in E'.\ d \leq e\}$, $\leq' = \leq|_{E'}$, and $\lambda' = \lambda|_{E'}$.

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

Definition 2.3. We say that P' is an *augment* of P if $E' = E$, $\lambda' = \lambda$, and $\leq' \supseteq \leq$.
We say that P' *implies* P if $E' = E$, $\leq' = \leq$, $\mathcal{A}' = \mathcal{A}$, and $\Phi'(e)$ implies $\Phi(e)$.

The semantics is also closed with respect to *disjunction*, which weakens pomsets by taking the disjunction of the formulae in their common downset. For example, since $[\![x{:=}1{+}r{*}r{-}r]\!]$ includes $(r{=}0 \mid \mathrm{W}x1)$ and $(r{=}1 \mid \mathrm{W}x1)$, it must also include $(r{=}0 \vee r{=}1 \mid \mathrm{W}x1)$ (see page 8). Likewise, $[\![[r]{:=}0;\ [0]{:=}!r]\!]$ must include $(r{=}0 \vee r{=}1 \mid \mathrm{W}[0]0)$ despite the fact that $(r{=}0 \mid \mathrm{W}[0]0)$ is generated by $[r]{:=}0$ and $(r{=}1 \mid \mathrm{W}[0]0)$ is generated by $[0]{:=}!r$ (see page 13).

Definition 2.4. We say that P is an *disjunct* of P^i $(i \in I)$ if there is some $k \in I$ and some downset P^i_∇ of each P^i such that (1) $E = E^k$, $\leq = \leq^k$, and $\mathcal{A} = \mathcal{A}^k$, (2) for every $i \in I$, $E^k_\nabla = E^i_\nabla$, ${\leq^k}_\nabla = {\leq^i}_\nabla$, and $\mathcal{A}^k_\nabla = \mathcal{A}^i_\nabla$, (3) for each $e \in (E^k \setminus E^k_\nabla)$: $\Phi(e)$ implies $\Phi^k(e)$, and (4) for each $e \in E^k_\nabla$: $\Phi(e)$ implies $\bigvee_i \Phi^i_\nabla(e)$.

2.4 Semantics of the Example Language

In the remainder of §2, we explain the semantics of our example language.

By far the most complex operators are the prefixing operators—read and write—which introduce new actions. We build the definition of prefixing from first principles, starting in §2.5. The final definition of prefixing appears in §2.7.

Of the other operators, it will not be surprising to students of concurrency theory that the most interesting are local declarations ($\mathsf{var}\, x\,;\, C$) and parallel composition ($C \parallel D$).

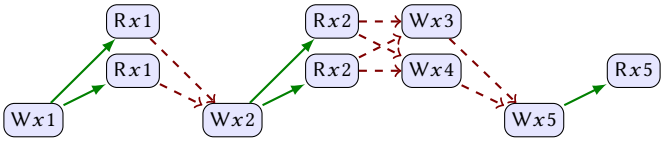
At the point that x is bound, we can require that every read of x be *fulfilled*. Fulfillment plays the role that *reads-from* and *coherence* play in other relaxed memory models, yet it is not the same.

Parallel composition is roughly pomset union, allowing that some events may *coalesce*, with the resulting precondition being the disjunction of the precondition taken from the two sides. Composition is used define conditionals (as in [Disselkoen et al. 2019]). We also use it to define write prefixing (§2.5). In both cases, the use of composition is required to ensure disjunction closure (Definition 2.4).

Definition 2.5. Two actions *conflict* if one writes a location and the other either reads or writes the same location. We say d *fulfills* e *(on* x*)* if (1) d writes v to x, (2) e reads v from x, (3) $d < e$, and (4) for every conflicting write c, either $c \leq d$ or $e \leq c$.

Item (3) requires that a write d is ordered before any read e it fulfills; we draw order required by this item (aka *reads-from*) with bold green arrows. Item (4) requires that any conflicting write c is ordered before d or after e; we draw order required by this item with dashed red arrows (aka *extended coherence*). In our model, these are not separate orders. We include the *reason* for the order only to improve readability. As an example, consider:

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



A write is *relevant* if it is read from. In order to fulfill all of the reads on x in the example, we pick a total order on the relevant writes: in this case, $(\mathrm{W}x1) \leq (\mathrm{W}x2) \leq (\mathrm{W}x5)$. The reads slot between these, immediately after their fulfilling write. Reads are not necessarily ordered with respect to each other, even if they come from the same thread, as do the reads here. Irrelevant writes also float relative to each other, as do $(\mathrm{W}x3)$ and $(\mathrm{W}x4)$. But irrelevant writes must be ordered with respect to relevant writes and reads. The resulting order is somewhat weaker than traditional extended coherence, which requires a total order on the writes, regardless of whether they are relevant. We discuss coherence further on page 10.

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

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

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

The definition of location binding validates *scope extrusion* [Milner 1999]: if C does not mention x then $[\![C \parallel \mathsf{var}\, x\,;\, D]\!] = [\![\mathsf{var}\, x\,;\, (C \parallel D)]\!]$. However, the definition 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.

Concurrent composition is roughly union. Because of consistency (Definition 2.1), we do not include events with contradictory preconditions. Consider:

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

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

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

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

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

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

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

Conditional execution is defined using parallel composition and *filtering*: $(\phi \triangleright \mathcal{P})$ selects the subset of pomsets in $\mathcal{P}$ that imply ϕ. Register assignment is defined using substitution: $(\mathcal{P}\sigma)$ performs the substitution σ on every pomset in $\mathcal{P}$. The semantics of `skip` is defined using singletons: STOP is the set of pomsets that contain a single termination event.

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

$$[\![\texttt{if}(M)\{C\}\,\texttt{else}\,\{D\}]\!] \triangleq (M \triangleright [\![C]\!]) \parallel (\neg M \triangleright [\![D]\!]) \qquad [\![r := M;\, C]\!] \triangleq [\![C]\!][M/r] \qquad [\![\texttt{skip}]\!] \triangleq \mathsf{STOP}$$

2.5 Program Order Prefixing

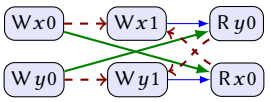
We present several candidate definitions for prefixing before giving the final definition. The candidates are progressively more general and less ordered. We begin by considering programs with trivial expressions, including all of program order in pomset order.

Candidate 2.9. Let $(\phi \mid a) \overset{\text{triv}}{\Longrightarrow} \mathcal{P}$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ when there is $P \in \mathcal{P}$ such that P' adds a new event that precedes all of the events in P.

$$[\![r := x^\mu;\, C]\!] = \bigcup_v (\mathrm{R}^\mu x v) \overset{\text{triv}}{\Longrightarrow} [\![C]\!]$$
$$[\![x^\mu := v;\, C]\!] = \mathrm{W}^\mu x v \overset{\text{triv}}{\Longrightarrow} [\![C]\!]$$

The definition ensures that program order is included in the pomset order. Due to the requirements of fulfillment, we also have that eco is included in pomset order. As a result, all executions are sequentially consistent. For example, consider the *store buffering* litmus test:

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



The read to write order is required by the definition of fulfillment. This candidate execution is *not* a pomset due to the resulting cycle; thus it is disallowed by Candidate 2.9.

For programs with general expressions, we must introduce preconditions. We write the definition of the prefixing operator more carefully this time, and highlight some of the changes in the candidate semantics:

Definition 2.10. Let $(\phi \mid a) \overset{\text{sc}}{\Longrightarrow} \mathcal{P}$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ when there is $P \in \mathcal{P}$ such that (1) $E' = E \uplus \{d\}$, (2) $\leq' \supseteq \leq$, (3a) $\mathcal{A}'(d) = a$, (3b) $\Phi'(d)$ implies ϕ, (4a) $\mathcal{A}'(e) = \mathcal{A}(e)$,

(4b) if d reads v from x then $\Phi'(e)$ implies $\Phi(e)[v/x]$,
(4c) if d does not read then $\Phi'(e)$ implies $\Phi(e)$, and
(5) $d <' e$,

Candidate 2.11.
$$[\![r := x^{\mu};\, C]\!] = \bigcup_v (\mathrm{R}^{\mu} x v) \overset{\text{sc}}{\Longrightarrow} [\![C]\!]\ [x/r]$$
$$[\![x^{\mu} := M;\, C]\!] = \|_v (M = v \mid \mathrm{W}^{\mu} x v) \overset{\text{sc}}{\Longrightarrow} [\![C]\!]$$

Item 1 introduces a new event. Item 2 ensures that no order is removed from old events. Item 3 describes the label of the new event, which must imply ϕ. Item 4 describes the labels of old events, as discussed below. Item 5 ensures that program order is included for the new event.

For writes, item 4 is simple: For old events, the new precondition in P' must imply old the precondition in P. (This is very similar to the treatment of the new event in item 3.)

The semantics of write introduces a write action for each possible value of the expression M. To ensure that at most one write is enabled, these are given disjoint preconditions.

The semantics is again driven by Hoare logic—for the preconditions of writes, the relevant rule is left disjunction:

$$\frac{\{\phi^1\}\ C\ \{\psi\} \quad \{\phi^2\}\ C\ \{\psi\}}{\{\phi^1 \vee \phi^2\}\ C\ \{\psi\}}$$

Note that $[\![x := 1 + r * r - r]\!]$ includes both $(r{=}0 \mid \mathrm{W}x1)$ and $(r{=}1 \mid \mathrm{W}x1)$. By using $\|_v$ in the definition of write, it also includes: $(r{=}0 \vee r{=}1 \mid \mathrm{W}x1)$. An alternate definition using $\cup_v$ would exclude this pomset.

Given that the conditional is defined using $\|$, the use of $\|$ in the definition of write is necessary to validate *case analysis*: $[\![C]\!] = [\![\texttt{if}(M)\{C\}\,\texttt{else}\,\{C\}]\!]$.

For reads, item 4b allows some preconditions to weaken and requires others to strengthen. Recall the pomsets given previously (∗) for $y := r$ and $\texttt{if}(r < 0)\{y := 1\}$. Prepending $r := x$ first causes the substitution $[x/r]$:

$$r := x;\ y := r \qquad\qquad r := x;\ \texttt{if}(r < 0)\{y := 1\}$$
$$(\mathrm{R}x1) \overset{\text{sc}}{\Longrightarrow} (x = 1 \mid \mathrm{W}y1) \qquad\qquad (\mathrm{R}x1) \overset{\text{sc}}{\Longrightarrow} (x < 0 \mid \mathrm{W}y1)$$

Item 4b then the substitutes the chosen value $[1/x]$:

$$(\mathrm{R}x1) \to (1 = 1 \mid \mathrm{W}y1) \qquad\qquad (\mathrm{R}x1)\ \ \cancel{(1 < 0 \mid \mathrm{W}y1)} \qquad (\dagger)$$

On the right, $(\mathrm{W}y1)$ has become impossible; this is no longer a pomset due to *inconsistency*. On the left, it has become causally dependent on the read. By prefixing a read event, the precondition $x = 1$ has moved from the sequential realm of Hoare logic to the concurrent memory model. Rather than a precondition that must be *satisfied*, the resulting pomset has a read event that must be *fulfilled*.

Whereas a write action introduces a precondition—satisfied sequentially—on the value to be written; a read introduces a pomset requirement—fulfilled concurrently—on the value to be read. Since reads of different values do not have disjoint preconditions, it is important that a read introduce at most one event per pomset. Thus, we use $\cup_v$ to combine pomsets for different read values, rather than $\|_v$.

2.6 General Prefixing

We now relax item 5 so that only some program order is preserved. The final definition has a small delta with respect to Definition 2.10. A pomset must preserve program order in four cases: 5a weakening a dependent precondition, 5b conflict (aka, coherence), 5c ra-fences, and 5c sc-fences.

It is amazing how much this definition "gets right" out of the box, including "internal reads", which greatly complicate other models [Pulte et al. 2018]. We walk through several litmus tests.

Definition 2.12. Let $(\phi \mid a) \Rightarrow \mathcal{P}$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ when there is some $P \in \mathcal{P}$ that satisfies items 1-4 of Definition 2.10 such that:

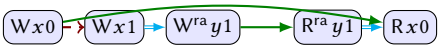
(5a) if e writes then either $d <' e$ or $\Phi'(e)$ implies $\Phi(e)$,
(5b) if d and e are actions in conflict, then $d <' e$,
(5c) if d is an acquire or e is a release, then $d <' e$, and
(5d) if d is an SC write and e is an SC read, then $d <' e$.

Candidate 2.13. $[\![r := x^{\mu}; C]\!] = \bigcup_v (\mathrm{R}^{\mu} x v \Rightarrow [\![C]\!][x/r]$
$[\![x^{\mu} := M; C]\!] = \|_v (M = v \mid \mathrm{W}^{\mu} x v \Rightarrow [\![C]\!]\,[M/x]$

Item 5a captures *read to write dependency*[2]. It only requires order from read to write when the precondition of the write is *weakened* using 4b. Item 5b captures the extended coherence requirement on actions that touch the same location. Item 5c imposes the order required by acquire and release actions[3]. Item 5d imposes the additional order required by SC actions[4].

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

$$x := 0; x := 1; y^{\mathrm{ra}} := 1 \;\|\; r := y^{\mathrm{ra}}; s := x \qquad \text{(Pub1)}$$

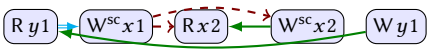


By Definition 2.5, $(\mathrm{R}x0)$ is *unfulfilled* in this pomset. It fails the last requirement of the definition, since $(\mathrm{W}x0) \le (\mathrm{W}x1) < (\mathrm{R}x0)$. In order to satisfy this requirement, $(\mathrm{R}x0)$ must be ordered before $(\mathrm{W}x1)$, but this creates a cycle.

Items 5d ensures that program order between SC operations is always preserved. Combined with the requirements for fulfillment, this is sufficient to establish that programs with only SC access have only SC executions; for example, execution candidate SB is banned when the actions of the two threads are all sc (but allowed with less order otherwise, as discussed below). It is also immediate that SC actions can be totally ordered, using any linearization of pomset order. Just as SC access in ARMV8 is simplified by MCA, it is simplified here by the global pomset order.

Unlike [Dolan et al. 2018, §8.2], our model allows:

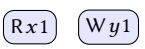
$$r := y; x^{\mathrm{sc}} := 1; s := x \;\|\; x^{\mathrm{sc}} := 2; y := 1 \qquad \text{(SC1)}$$



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

We relax program order on non-SC accesses in order to allow outcomes like that of execution candidate SB. Order is relaxed between reads, between writes to different locations, and from a read to an independent write:

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

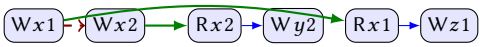


The existence of this pomset is justified by the Hoare triple $\{\mathrm{true}\}\ C\ \{y = 1\}$.

[2]When d is not a read, 4c trivially implies 5a.
[3]Recall that termination actions are releases.
[4]Recall that SC reads are acquires and SC writes are releases.

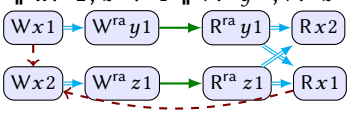
Unordered actions can be scheduled freely. As a result, our model of coherence is similar to that of Dolan et al. [2018]. Since reads are not ordered by 5b, we allow the following, which C11 forbids:

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



Conversely, we forbid the following, which Java allows:

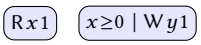
$$x := 1;\ y^{\mathsf{ra}} := 1 \parallel x := 2;\ z^{\mathsf{ra}} := 1 \parallel r := y^{\mathsf{ra}};\ r := z^{\mathsf{ra}};\ r := x;\ r := x \tag{Co3}$$



The order from $(\mathrm{R}x1)$ to $(\mathrm{W}x2)$ is required to fulfill $(\mathrm{R}x1)$.

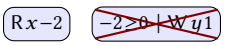
Item 5a imposes order from read to write when weakening the precondition of the write via 4b, as on the left hand side of (†). Item 4b *allows* a precondition to weaken, but does not *require* it. Item 5a only requires order when the precondition weakens. Thus, no order is required in:

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



The existence of this pomset is justified by the Hoare triple $\{x \geq 0\}\ C\ \{y = 1\}$. It is not justified by the value of the read action.

Nonetheless, item 4b requires that the value of the read action must be *consistent* with subsequent formulae, via $\Phi(e)[v/x]$. In this example, the pomset becomes inconsistent if -2 is read for x:



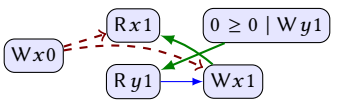
This holds for all preceding reads, unless the precondition is discharged by introducing order. Thus, $[\![s := x;\ C]\!]$ contains the pomset on the left below, but not that inconsistent "pomset" on the right:

Like item 4b, which substitutes $[v/x]$ during a read, Candidate 2.13 substitutes $[M/x]$ during a write. Like 4b, this affects subsequent preconditions, either allowing them to weaken, or requiring them to strengthen. For write prefixing, however, there is no rule corresponding to item 5a. Unlike a read event, order is *not* imposed from a write event to the subsequent events whose precondition it weakens:

$x := 0;\ C$ $\qquad\qquad$ $x := -2;\ C$

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

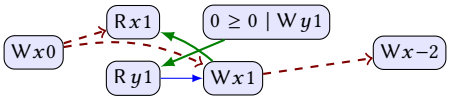
$$x := 0;\ (r := x;\ \mathtt{if}(r \geq 0)\{y := 1\} \parallel x := y) \tag{TC1}$$



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

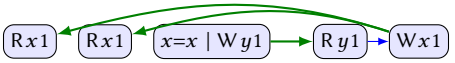
This execution is an example of an *internal read*, using ARMv8 terminology [Pulte et al. 2018]. Unlike [Jeffrey and Riely 2016], our semantics is robust with respect to the introduction of concurrent writes, as in TC9:

$$x{:=}0;\ (r{:=}x;\ \texttt{if}(r \geq 0)\{y{:=}1\} \parallel x{:=}y \parallel x{:=}-2) \tag{TC9}$$



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

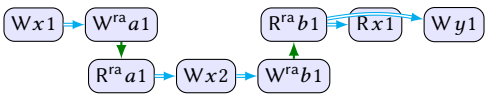
$$r{:=}x;\ s{:=}x;\ \texttt{if}(r{=}s)\{y{:=}1\} \parallel x{:=}y \tag{TC2}$$



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

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

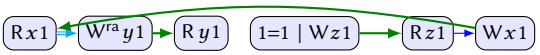
$$x{:=}1;\ a^{\mathrm{ra}}{:=}1;\ \texttt{if}(b^{\mathrm{ra}})\{y{:=}x\} \parallel \texttt{if}(a^{\mathrm{ra}})\{x{:=}2;\ b^{\mathrm{ra}}{:=}1\} \tag{Internal1}$$



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

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

$$r{:=}x;\ y^{\mathrm{ra}}{:=}1;\ s{:=}y;\ z{:=}s \parallel x{:=}z \tag{Internal2}$$



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

Our model drops order into actions that depend on a read that can be fulfilled *internally*, by a prefixed write. This is natural consequence of substitution. The ARMv8 model has to a jump through some hoops to ensure that internal reads are handled correctly. ARMv8 takes the symmetric approach: rather than dropping order *out of* an internal read, ARMv8 drops the order *into* it. This difference complicates the proof for ARMv8 (§5).

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

2.7 Relaxed Write Elimination

We discuss compiler optimization in §7. Irrelevant reads have no effect in our model, thus we can define correctness with respect to pomsets that have been saturated with arbitrary irrelevant reads. The same does not hold for writes.

In our final definition of the semantics, we allow for the possibility of relaxed write elimination:

Definition 2.14. Let $(\text{cover}^{\text{rlx}}\ x \triangleright \mathcal{P})$ be the set $\mathcal{P}' \subseteq \mathcal{P}$ such that $P' \in \mathcal{P}'$ when for every release $e' \in E'$, there is some $d' \in E'$ such that $d' \leq e'$ and d' writes x.

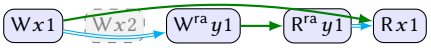
Let $(\text{cover}^{\text{ra}}\ x \triangleright \mathcal{P}) = (\text{cover}^{\text{sc}}\ x \triangleright \mathcal{P}) = \emptyset$.

$$[\![r := x^{\mu};\ C]\!] = \bigcup_v\ (\mathsf{R}^{\mu} x v) \Rightarrow [\![C]\!][x/r]$$

$$[\![x^{\mu} := M;\ C]\!] = \ \|_v\ \ (M = v \mid \mathsf{W}^{\mu} x v) \Rightarrow [\![C]\!][M/x] \cup\ (\text{cover}^{\mu}\ x \triangleright [\![C]\!][M/x])$$

The use of cover^{μ} in the definition prevents the eliminated write rule from applying immediately before a release. This prevents bad executions such as:

$$x := 1;\ x := 2;\ y^{\text{ra}} := 1 \ \|\ r := y^{\text{ra}};\ s := x \qquad \text{(Cover)}$$

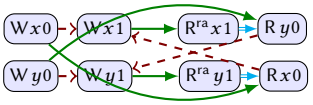


In this drawing, we have included a "non-event"—dashed border—to mark the eliminated write.

2.8 Litmus Tests

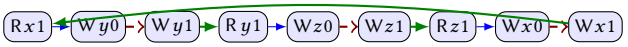
Our model gives the desired results for the test cases of Pugh [2004], Ševčík [2008, §5.3], and Batty et al. [2015, §4]. It also agrees with the "surprising and controversial behaviors" of Manson et al. [2005, §8]. We present two examples that are hallmarks of MCA architectures. The analysis follows from a few simple principles.

$$x := 0;\ x := 1 \ \|\ r := x^{\text{ra}};\ s := y \\ \|\ y := 0;\ y := 1 \ \|\ r := y^{\text{ra}};\ s := x \qquad \text{(MCA1)}$$



In this variant of IRIW (Independent Reads of Independent Writes), order is imposed by *coherence* (between the writes), *fulfillment* (between read and write), and *fencing* (from acquiring read to relaxed read). Given the evident cycle, the candidate execution is invalid. It is also impossible for all threads to read 1 in the following, due to *control dependencies*, *coherence*, and *fulfillment*.

$$\texttt{if}(x)\{y := 0\};\ y := 1 \ \|\ \texttt{if}(y)\{z := 0\};\ z := 1 \ \|\ \texttt{if}(z)\{x := 0\};\ x := 1 \qquad \text{(MCA2)}$$



In either example, the execution is allowed if the cycle is broken—for example, by changing x^{ra} to x^{rlx} in IRIW.

3 EXTENSIONS

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

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

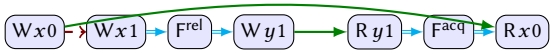
$$[\![\mathsf{F}^{\nu};\ C]\!] = (\mathsf{F}^{\nu}) \Rightarrow [\![C]\!]$$

Syntactic fences require additional order to simulate ra/sc-accesses. We add the following rules to Definition 2.12 of *prefixing*:

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

Consider the following variant of Pub1:

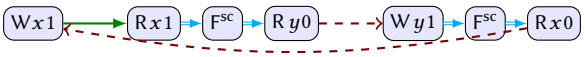
$$x := 0;\ x := 1;\ \mathsf{F}^{\mathsf{rel}};\ y := 1 \ \parallel\ r := y;\ \mathsf{F}^{\mathsf{acq}};\ s := x \qquad \text{(Pub2)}$$



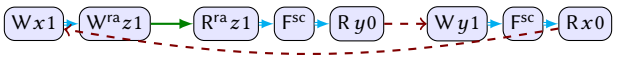
5f requires that $(\mathsf{F}^{\mathsf{rel}}) \le (\mathsf{W}y1)$. 5e requires that $(\mathsf{R}y1) \le (\mathsf{F}^{\mathsf{acq}})$. The attempted execution is *invalid*: the stale read $(\mathsf{R}x0)$ violates the last requirement of fulfillment (Definition 2.5).

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

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



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



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

Address Calculation. In the definition of a data model, 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:

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

For degenerate programs that include only constant references (every expression $[L]^{\mu}$ satisfies $L = \ell$, for some ℓ), the following semantics produces exactly the same executions as before.

$$[\![r := [L]^{\mu};\ C]\!] = \bigcup_{\ell,v} \big(\ L = \ell \mid \mathsf{R}^{\mu}[\ell]v\big) \Rightarrow [\![C]\!][[\ell]/r]$$

$$[\![[L]^{\mu} := M;\ C]\!] = \parallel_{\ell,v} \big(\ L = \ell \wedge M = v \mid \mathsf{W}^{\mu}[\ell]v\big) \Rightarrow [\![C]\!][M/[\ell]] \ \cup \parallel_{\ell} \big(\mathsf{cover}^{\mu}\,[\ell] \rhd [\![C]\!][M/[\ell]]\big)$$

Let us revisit the discussion of the use of $\parallel$ in Candidate 2.11. Note that $[\![[r] := 0;\ [0] := !r]\!]$ includes both of the following pomsets ("$!M$" evaluates to 1 if M is 0, and 0 otherwise):

By using $\parallel$, it also includes:

(‡)

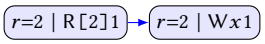
In this example, the events that coalesce correspond to different statements in the syntax.

Because we do not enforce order between reads, there is some danger that address calculations could introduce anomalous behaviors that arise *out of thin air* (OOTA) [Batty et al. 2015]. Consider the following attempted execution, where all memory addresses are initialized to 0, except that $[2]$ is 1 and $[1]$ is 2:

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

Although there is no order enforced between the reads, the read-to-write order induced by the semantics is sufficient to prohibit this OOTA behavior. Note the intermediate state:

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



The precondition on the write is required by causal strengthening in Definition 2.1: if $d \le e$ then $\Phi(e)$ implies $\Phi(d)$.

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

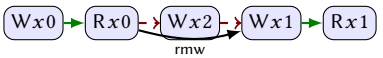
- If c, e write the same x, $c \le e$ and $d \xrightarrow{\mathsf{rmw}} e$ then $c \le d$.
- If c, e write the same x, $d \le c$ and $d \xrightarrow{\mathsf{rmw}} e$ then $e \le c$.

In Definition 2.2, require that downsets are RMW closed: $E' \subseteq \{d \in E \mid \exists e \in E'.\ e \xrightarrow{\mathsf{rmw}} d\}$.

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

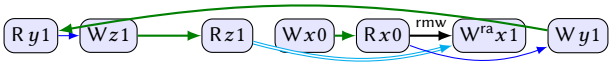
This definition ensures atomicity, disallowing executions such as [Podkopaev et al. 2019, Ex. 3.2]:

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



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

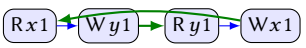
$$r := y\,;\; z := r \;\parallel\; r := z\,;\; x := 0\,;\; s := \mathsf{FADD}^{\mathsf{rlx},\mathsf{ra}}(x)\,;\; y := s{+}1 \qquad \text{(RMW2)}$$



4 UNDERSTANDING "THIN AIR READS" USING TEMPORAL LOGIC

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

$$(y := x \;\parallel\; r := y\,;\; x := r) \qquad \text{(OOTA3)}$$



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

On first read, we suggest that readers skip to the examples and the discussion that follows, coming back to the details of the logic as necessary. Example 4.2 discusses the canonical OOTA example OOTA3; the analysis is trivial and well-known [Jeffrey and Riely 2019; Kang et al. 2017]. Example 4.3 is more interesting. It discusses a variant of Lochbihler's example OOTA1, from the introduction. The logic given here is not meant to be definitive; on page 23, we discuss OOTA examples that require non-trivial extensions [Chakraborty and Vafeiadis 2019; Svendsen et al. 2018].

We adapt past linear temporal logic (PLTL) [Lichtenstein et al. 1985] to pomsets by dropping the previous instant operator and adopting strict versions of the temporal operators. The atoms of our logic are write and read events. Given a pomset P and event e, define[5]:

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

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 $\{Q \mid Q \models \varrho\} \parallel \mathcal{P} \models \vartheta$.
Let ϱ be *downclosed* when $\{Q \mid Q \models \varrho\}$ is.
The past operators do not include the current instant, and so do *not* satisfy $(\boxminus\varrho \Rightarrow \diamondminus\varrho)$[6]. However, the following hold:

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

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

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

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 4.1. *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 $P \models \varrho$. □

The logic is defined with respect to downclosed sets, but $[\![C]\!]$ includes only completed pomsets. For reasoning in the logic, we downclose the semantics, considering pomsets that may not be completed. Let $\nabla[\![C]\!] = \{P' \mid P' \text{ is a downset of some } P \in [\![C]\!]\}$.

Example 4.2. With all variables initialized to 0, we show that OOTA3 satisfies $\neg\mathsf{W}x1$.
We start with the invariant:

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

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

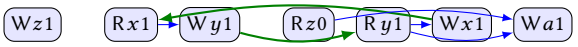
This invariant holds for each thread; thus, it holds for the aggregate program by composition. Closing y yields $\mathsf{R}y1 \Rightarrow \Diamond\mathsf{W}y1$. Weakening the right conjunct: $\Diamond\mathsf{W}y1 \Rightarrow \Diamond\mathsf{R}x1$. Chaining these together: $\mathsf{R}y1 \Rightarrow \Diamond\mathsf{R}x1$.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

Example 4.3. Our language lacks object creation; therefore, we consider a variant of OOTA1:

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

OOTA4 retains the essential temporal aspects of Lochbihler's example, discussed in §1. In this case, all threads satisfy the invariant "any write to location a is preceded by a reading 1 for z," rather than "allocation at type C is preceded by reading 1 for z"

As a warmup, note that attempting to write 1 to a results in a cycle:



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

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

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

$$\mathsf{W}a1 \Rightarrow (\Diamond\mathsf{R}x1 \wedge \boxminus(\mathsf{W}x1 \Rightarrow \Diamond\mathsf{R}x1))$$

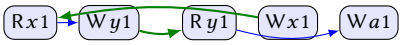
Closing x: $\mathsf{W}a1 \Rightarrow (\Diamond\mathsf{W}x1 \wedge \boxminus(\mathsf{W}x1 \Rightarrow \Diamond\mathsf{W}x1)$. Applying coinduction to the right conjunct:

$$\mathsf{W}a1 \Rightarrow (\Diamond\mathsf{W}x1 \wedge \boxminus(\neg\mathsf{W}x1))$$

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

Many examples are superficially similar, but in fact have fewer dependencies. A referee for a previous version of this paper expected that the following example is "the same":

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



In this execution, $\mathsf{W}x1$ is independent of $\mathsf{R}y1$, thus there is no OOTA behavior.

On page 21, we discuss Boehm's [2018] RFUB example, which presents another potential form of OOTA behavior, in the context of compiler optimization. Our analysis shows that there is no OOTA behavior in RFUB, instead Boehm's analysis has a false dependency.

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

5 EFFICIENT IMPLEMENTATION ON ARMV8

We consider the fragment of our language where concurrent composition occurs only at top level and there are no location declarations. Using the translation strategy of Podkopaev et al. [2019], we show that any *consistent* ARMv8 execution graph for this sublanguage can be considered a top-level execution of our semantics. Consistency is defined by Pulte et al. [2018]. The key step is constructing the order for the derived pomset candidate. We would like to take $\leq = (\mathsf{ob} \cup \mathsf{eco})^*$, where ob is the ARMv8 acyclicity relation, and eco is the ARMv8 extended coherence order, as discussed after Definition 2.5. But this does not quite work.

The definition is complicated by ARMv8's *internal reads*, manifest in rfi, which relates reads to writes that are fulfilled by the same thread. ARMv8 drops ob-order *into* an internal read. As discussed in §2, however, our semantics drops pomset order *out of* an internal read. To accommodate this,

we drop these dependencies from the ARMv8 *dependency order before* (dob) relation. The relation dob′ is defined from dob by restricting the order into and out of a read that is in the codomain of the rfi relation. More formally, let $d \xrightarrow{\text{dob}'} e$ when $d \xrightarrow{\text{dob}} e$ and $d \notin \text{codom}(\text{rfi}), e \notin \text{codom}(\text{rfi})$. Let ob′ be defined as for ob, simply replacing dob with dob′.

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

We prove the following theorem in §B.

Theorem 5.1. *For any consistent* ARMv8 *execution graph, the constructed candidate is a top-level memory model pomset.*

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

6 LOCAL DATA RACE FREEDOM AND SEQUENTIAL CONSISTENCY

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

When constructing a pomset, define *program order* (po) in the obvious way. As usual[7], we say that d *synchronizes with* e (notation $d \xrightarrow{\text{sw}} e$) exactly when d fulfills e, d is a release, e is an acquire, and $\neg(d \xrightarrow{\text{po}} e)$. Let $\text{hb} = (\text{po} \cup \text{sw})^+$ be the *happens-before* relation. In Pub1, for example, $(\mathsf{W}x1)$ 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 $d \overset{L}{\rightsquigarrow} e$) when they conflict at some location in L, but are unordered by hb: neither $d \xrightarrow{\text{hb}} e$ nor $e \xrightarrow{\text{hb}} d$. The definition of L-race uses *program* order, not *pomset* order, and thus is stable with respect to augmentation. 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.

To state the theorem, we require several technical definitions. The reader unfamiliar with [Dolan et al. 2018] may prefer to skip to the examples that follow the theorem statement, coming back to the definitions as needed.

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

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

We say that P' C*-extends* P if $P \in \min[\![C]\!]$, $P' \in \min[\![C]\!]$, and $P \in \nabla P'$.

We say that P is L*-unstable in* C if $P \in \min[\![C]\!]$ and either (1) P is not po-convex (something missing in program order), or (2) there is a C-extension of P with a *crossing* L-race: that is, there is some $d \in E$, some P' C-extending P, and some $e \in E' \setminus E$ such that $d \overset{L}{\rightsquigarrow} e$.

We say that P is L*-stable in* C if $P \in \min[\![C]\!]$ and P is not L-unstable in C.

Note that the empty pomset is L-stable.

Let $\lessdot_L = < \cup \text{po}_L$, where po_L is the restriction of po to events that read or write locations in L.

We say that $P' \in \min[\![C]\!]$ is L*-sequential in* C after P if (1) P is L-stable in C, (2) P' C-extends P, (3) P' is po-convex, and (4) $\lessdot_L$ is acyclic in $P' \setminus P$.

Theorem 6.1. *Suppose P is L-stable in C, P' is an L-sequential C-extension of P, and P'' is a nonempty C-extension of P'. Then either*

(1) $\exists e \in (E'' \setminus E')$ *that* L-covers P': $(\nexists d \in (E'' \setminus E') : \; d \lessdot_L e)$ *or*

(2) $\exists P'''$*: an* L*-sequential* C*-extension of* P' *with an* L-race after P: $(\exists \{d, e\} \subseteq (E''' \setminus E) : \; d \overset{L}{\rightsquigarrow} e)$.

Proof Sketch. Suppose there is no e satisfying (1). Then we must find P''' satisfying (2). From (1), we deduce that there is a minimal $\lessdot_L$ cycle in P''. From the semantics, we deduce that there

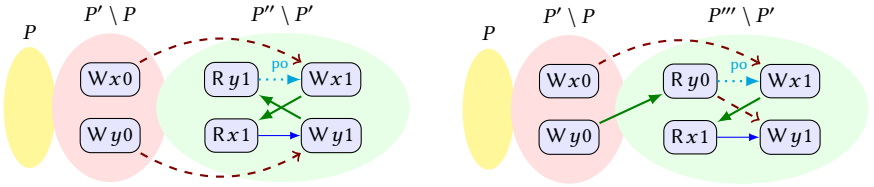
[7] To allow this simple definition of sw, we consider the base language of §2, which does not include fences or RMWs.

must be at least one read in this cycle that is minimal w.r.t. pomset order. From the semantics, we find the required P''' by choosing to read a different value. □

We discuss the required properties of the semantics by example, taking $L = \{x, y\}$ and $P = \emptyset$.

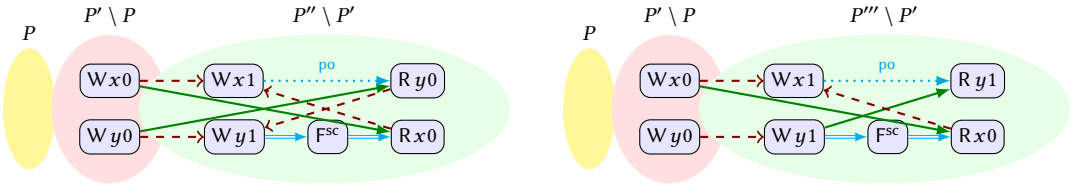
Suppose that the $\leq$-minimal read is fulfilled by a write in P''. In this case, we break the cycle by reading an "older" value from P'. For example, we switch from P'' on the left to below to P''' on the right. Program order goes from left to right, with the left thread above the right one; we only show program order explicitly when it is relevant to the example.

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



Suppose that the $\leq$-minimal read is fulfilled by a stale write in P'. In this case, we break the cycle by reading an "newer" value from P''. (We include a fence in the following example to break the symmetry.)

$$(y := 0;\ x := 1;\ r := y) \parallel (x := 0;\ y := 1;\ \mathsf{F}^{\mathsf{sc}};\ s := x)$$



Because we define L-stability using $\lessdot_L$ rather than po, this result does not suffer from the complications caused by "mixed races" in [Dongol et al. 2019]. In particular, the P' chosen on the left below is *not* L-sequential, because it is not $\lessdot_L$-convex.

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

To satisfy the premise of the theorem, we must choose P' as on the right. This simplification is enabled by denotational reasoning and a precise characterization of the dependency.

7 SINGLE-THREADED OPTIMIZATIONS

We discuss compiler optimizations both concretely and abstractly. Concretely, we show the validity of specific optimizations, such as the roach motel laws for synchronization. Abstractly, we establish sufficient conditions to replace any command C by an equivalent D: if C and D are synchronization-free and sequentially equivalent, and furthermore D is *linear*—performs at most one read and at most one write on each location in each execution—then C can be refined to D.

The linearity restriction ensures that the context cannot interfere with the atomic execution of the command, and, dually, that the atomic execution of the command cannot interfere with

the context. To see the need for this, consider that the introduction of redundant reads is valid sequentially, but not valid concurrently. For example, $r := x;\ \texttt{if}(r \neq r)\{y := 1\}$ cannot be refined to $r := x;\ s := x;\ \texttt{if}(r \neq s)\{y := 1\}$. In a concurrent context, the latter program may see different values for the two reads.

A program is *sequential* if it lacks $\|$, and *synchronization-free* if lacks fences and ra/sc access. We argue that our model is fully flexible with respect to optimization of such programs, as long as the optimizations do not introduce new writes or "relevant" reads. To do so, §7.1 formalizes the relation between Hoare triples and pomsets with preconditions. §7.2 then isolates a *linear* fragment of our language and shows the soundness of *all* transformations of synchronization-free sequential programs into this fragment. §7.3 discusses specific optimizations, some of which relax the linearity assumption and include synchronization. Finally, §7.4, discusses "optimizations" that fail, such as Boehm's [2018] RFUB example.

7.1 Pomsets for Hoare Logic

In §2, we used Hoare logic [Gordon 2012; Hoare 1969] to reason about *dependency analysis* for sequential code. Here, we use an alternative interpretation of Hoare logic to establish the soundness of *program transformations*. We develop a pomset semantics for pairs of formulae $[\![\phi \mid \psi]\!]$, and show a relation between $[\![C]\!]$ and $[\![\phi \mid \psi]\!]$ for valid Hoare triples $\{\phi\}\ C\ \{\psi\}$. In §2, preconditions were discharged by read events. Here, instead, preconditions are derived from the read events themselves.

Definition 7.1. Let $P \in [\![\phi \wedge \chi \mid \psi]\!]$ when it possible to satisfy the following:

Let X be the set of locations such that $x \in X$ exactly when ϕ depends on x. For each $x \in X$, choose $d_x \in E$ that reads x. Let $D_X = \{d_x \mid x \in X\}$. Let σ be the substitution generated as follows: $x\sigma = v$ exactly when d_x reads v from x.

Let Y be the set of locations such that $y \in Y$ exactly when ψ depends on y. For each $y \in Y$, choose $e_y \in E$ that writes y. Let $E_Y = \{e_y \mid y \in Y\}$. Let π be the substitution generated as follows: $y\pi = v$ exactly when e_y writes v to y.

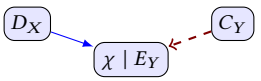
Require that $\phi\sigma$ and $\psi\pi$ are satisfiable.

Require that $\Phi(e_y)$ implies χ (for each e_y).

Require that if $c \leq e_y$ and c is a read, then $c \in D_X$.

Require that if $e_y \leq c$ and c is a write to Y, then $c \in E_Y$.

Pictorially, we have:



Here, E_Y are the final writes to Y, with precondition χ. C_Y are other writes to Y, which must be ordered before E_Y. D_X are the reads that the writes depend upon.

Under this interpretation, precondition strengthening in Hoare logic validates read introduction. In our semantics, reads have no side effects. Thus, it should be sound to introduce irrelevant reads. Yet, $[\![x := M;\ r := x;\ C]\!] \neq [\![x := M;\ C]\!]$, even when r does not appear in C. To make such equations hold, we define $\mathsf{read}(\mathcal{P})$ to saturate $\mathcal{P}$ with reads.

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

Theorem 7.2. *If C is synchronization-free and sequential, $\{\phi\}\ C\ \{\psi\} \iff [\![\phi \mid \psi]\!] \cap \mathsf{read}[\![C]\!] \neq \emptyset$.*

Proof. The proof proceeds by induction on the derivation of the Hoare triple.

We first consider the structural rules. Precondition strengthening follows from augmentation closure. The structural rule for disjunction $\frac{\{\phi_1\}\ C\ \{\psi_1\} \quad \{\phi_2\}\ C\ \{\psi_2\}}{\{\phi_1 \vee \phi_2\}\ C\ \{\psi_1 \vee \psi_2\}}$ follows from disjunction closure

(Definition 2.4). The structural rule for conjunction $\frac{\{\phi_1\}\ C\ \{\psi_1\} \quad \{\phi_2\}\ C\ \{\psi_2\}}{\{\phi_1 \wedge \phi_2\}\ C\ \{\psi_1 \wedge \psi_2\}}$ follows from the fact that pomsets have only concurrency and no conflict.

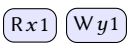
The remaining cases follow directly from the semantics. The only subtlety is the write rule, which uses $\|$ to ensure disjunction closure. □

Preconditions can be placed in ϕ or χ in Definition 7.1, resulting in different pomsets:

$$\{x = 1\}\ y := x\ \{y = 1\}$$

Control dependencies are calculated correctly:

$$\{\mathsf{true}\}\ \mathtt{if}(x)\{y := 1\}\,\mathtt{else}\,\{y := 1\}\ \{y = 1\}$$



For any consistent set of preconditions, we can always find a single pomset:

$$\{x_1 = 1\}\ y_1 := x_1\ \{y_1 = 1\} \qquad \{x_2 = 1\}\ y_2 := x_2\ \{y_2 = 1\}$$

Corollary 7.3. *If $\bigwedge_{i\in I} \phi_i$ is satisfiable, $\bigwedge_{i\in I}\{\phi_i\}\ C\ \{\psi_i\} \iff \bigcap_{i\in I}[\![\phi_i \mid \psi_i]\!] \cap \mathsf{read}[\![C]\!] \neq \emptyset$.*

7.2 Linearity

A command C is *linear* if for every $P \in [\![C]\!]$, there is at most one read and at most one write on any location. Intuitively, this means that the context around C is unable to interfere with the atomic execution of C; dually, neither can the atomic execution of C interfere with the context. From an arbitrary command, it is a straightforward exercise to construct a linear command that is sequentially equivalent.

We say that C and C' *satisfy the same Hoare triples* when $\{\phi\}\ C\ \{\psi\}$ if and only if $\{\phi\}\ C'\ \{\psi\}$, for every ϕ and ψ.

Corollary 7.4. *Let C and C' be synchronization-free and sequential. Further, let C' be linear. Then C and C' satisfy the same Hoare triples if and only if* $\mathsf{read}[\![C]\!] \supseteq \mathsf{read}[\![C']\!]$.

7.3 Valid Rewrites

When $\mathsf{read}[\![C]\!] \supseteq \mathsf{read}[\![C']\!]$, we say that C' is a *valid transformation* of C.

To enable reasoning about program fragments, transformation validity must be preserved by *contexts*. In §2, we defined the semantics by prefixing one action at a time. This helps to make the semantics understandable, but it also creates impoverished contexts.

To discuss valid transformations without getting lost in notation, we present them using simple locations, rather than calculated addresses. The extension is simple: For address expressions $[M]$ and $[N]$, replace $x = y$ by provable equality of M and N, and $x \neq y$ by provable inequality. Operations on sets can be defined similarly. Let $\mathsf{id}(C)$ be the set of locations and registers that occur in C.

Theorem 7.2 immediately validates peephole optimizations, such as redundant load elimination (RL), store forwarding (SF), dead store elimination (DS), and independent reorderings (WW, RW and RR). Using the semantics directly, we can prove some properties without using read. Note that

if $\mathcal{P}' \supseteq \mathcal{P}$, then $\mathsf{read}(\mathcal{P}') \supseteq \mathsf{read}(\mathcal{P})$.

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

$$\mathsf{read}[\![x:=M;\ s:=x;\ C]\!] \supseteq \mathsf{read}[\![x:=M;\ s:=M;\ C]\!] \tag{SF}$$

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

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

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

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

RL and SF follow from read closure. DS follows from *write elimination* (Definition 2.14).

Since reads are unordered in our model, read optimizations are not limited by the power of aliasing analysis. When $r_2 \neq s$, we validate CSE, from §??.

$$\mathsf{read}[\![r_1:=x;\ s:=y;\ r_2:=x;\ C]\!] \supseteq \mathsf{read}[\![r_1:=x;\ r_2:=r_1;\ s:=y;\ C]\!] \tag{CSE}$$

This holds by composing RR and RL, regardless of whether $x = y$. (To see this, recall Co2.)

The semantics also validates roach-motel reorderings. The rules for relaxed writes are as follows:

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

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

Many laws hold for the conditional. We show dead code elimination (DC) and case analysis (CA), which follows from disjunction closure (Definition 2.4).

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

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

As expected, parallel composition commutes with conditionals and declarations, and conditionals and declarations commute with each other. We discussed scope extrusion on page 6.

7.4 Invalid Rewrites

We discussed the invalidity of variable renaming on page 6 and the invalidity of thread inlining on page 11 . Rewrites that introduce accesses are generally invalid — irrelevant read introduction is the exception. For example, *relevant* read introduction is invalid:

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

These are distinguished by the context $[-] \parallel x:=1 \parallel x:=2$. Write introduction is always invalid:

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

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

Boehm [2018] considers the following programs:

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

The left command is half of OOTA3, from §4. The right command is dubbed RFUB, for *Register assignment From an Unexecuted Branch*. Boehm observes that in the context $x:=y \parallel [-]$, these programs have different behaviors. Yet the OOTA example on the left never writes 1. Why should the unexecuted branch change that? As it turns out, both branches of the conditional in RFUB can execute, since the write to x is independent of the read from y. Considering just the two threads above, we have $\{\mathsf{true}\}$ RFUB $\{x = 1\}$, but not $\{\mathsf{true}\}$ OOTA $\{x = 1\}$. As a result, it is expected that RFUB may have additional behaviors. The change in the thread from OOTA3 to RFUB is not a valid refinement under Hoare logic and thus it is not valid in our semantics.

8 OTHER RELATED WORK

We survey related work not discussed in §1-4 and §10.

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].

Disselkoen et al. [2019] introduced the notion of pomsets with preconditions. They studied *micro-architecture*—specifically, speculative execution. We have presented an *architectural* model, leading to many formal differences between our model and theirs. Chief among these: Their pomsets do not satisfy our *consistency* requirement, and thus contradictory preconditions are allowed in the same pomset, modeling aborted speculative executions. Similarly, they take reads to be side-effecting, modeling cache effects. There are many less fundamental differences. For example, their use of *three-valued pomsets* means that they allow MCA2, but disallow its two-location variant. This odd behavior stems from *semi-transitivity*, inherited from [Lamport 1986].

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

There is also a rich literature on the use of transformations over SC executions to model relaxed memory: Saraswat et al. [2007] aimed to describe a JMM-like model this way. DRF-SC holds, but Sevcik [2011] discovered that it permits OOTA behavior. Demange et al. [2013] developed BMM, which permits reordering of a relaxed write with a following relaxed read. BMM is designed as a restriction of the JMM that compiles efficiently to TSO. It requires fencing on other architectures.

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

9 LIMITATIONS

Our work has several limitations, each of which provides an opportunity for future research. In several cases, these limitations have allowed us to *surpass prior work in significant ways.*

The model does not provide read-read coherence (Co2), even for address dependencies (OOTA2). This model of coherence does not correspond to either Java or C++, and therefore is unfamiliar. *We consider this all upside. C++ invalidates* CSE *on relaxed access. Java invalidates the local* DRF-SC *theorem [Dolan et al. 2018].*

We treat loops via unrolling: loops introduce complexities—such as liveness and continuity—that are orthogonal to the main topic of the paper. We also do not include types, allocation, garbage collection, etc. This is the norm for work on relaxed memory: The problem is already hard enough. *On the upside, we do allow thread creation via* $\|$*, which is unusual.*

Except for roach motel (RelW, AcqW), the model does not validate transformations that involve synchronizations and fences [Vafeiadis et al. 2015]. For example, the model does not support elimination of redundant synchronization accesses. Accommodating such optimizations will likely require changes to the definition of cover (page 12) and read (page 19). *On the upside, Corollary 7.4 is the first abstract characterization of valid transformations for relaxed access.*

Our model realizes *multi-copy atomicity* (MCA). Thus it will not compile efficiently to non-MCA architectures, such as POWER and ARMv7. To do so, one cannot include the order required by the last item of Definition 2.5 in pomset order. It may be sufficient to use a weaker second order. Disselkoen et al. [2019] attempted to define such a weaker order, but their model is anomalous on MCA2—as we

discussed in §8. *On the upside, ours is the first language-level model of* MCA. *The use of preconditions does not depend on* MCA, *so there is hope to extend our approach to dependency calculations to non-*MCA *architectures.*

Our DRF-SC theorem does not address fences or RMWs. *On the upside, we have proven a* local DRF-SC *theorem, which is stronger than the standard* DRF-SC *result.*

The logic we presented in §4 is only strong enough to prove a few examples. Svendsen et al. [2018] presented a different a logic, capable of showing that the following program cannot write 2:

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

The attempted execution is *not* allowed by our semantics since there is no write to fulfill $(\mathsf{R}y1)$. Proving this requires the ability to reason about values. As another example, consider [Chakraborty and Vafeiadis 2019, Fig. 3]:

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

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

The attempted execution is *not* allowed by our semantics, due to the evident cycle. Surprisingly, this outcome is allowed by the promising semantics [Kang et al. 2017]. Chakraborty and Vafeiadis developed WEAKESTMO to address this example. Intuitively, it is not possible for the left thread to read 3 for x when the right thread reads 2. Proving this may require a logic with modalities to deal with intervening writes and coherence. *On the upside, our logic is powerful enough to show that our semantics disallows* OOTA *executions of OOTA4 — WEAKESTMO and the promising semantics both permit* OOTA *executions of OOTA4 (see §A).*

Our results have not been formally verified, but only because we do not have the resources to do it ourselves. It is worth noting that the current emphasis on formal verification has the unintended effect of making research more conservative: Large libraries are built, and large institutions with teams of graduate students build them.

10 CONCLUSIONS

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

In §4, we proposed a cyclical proof rule for parallel composition of temporal properties in the style of Abadi and Lamport [1993]. This provides an objectively falsifiable alternative to the notion of *out of thin air* (OOTA) execution, which has been famously difficult to prove or disprove [Batty et al. 2015; Boehm 2018].

In §7, we formalized a connection between Hoare logic and the sequential fragment of our language, allowing us to make a general statement about transformations. Thus, we present a solution to a problem that has been open since Cenciarelli et al. [2007] discovered reordering of independent statements is invalid in the JMM [Manson et al. 2005].

These contributions rest on a semantics that supports a simple, executable prescription of *essential dependencies*, calculated using logic and manifest in pomset order. This provides a clear interface between the language user and the compiler writer, and a clear bridge from the compiler writer to computer architecture.

We presented a denotational semantics using pomsets with preconditions. The model may also be expressible in the operational framework of Ferreira et al. [2010], which parameterizes a standard SC operational semantics with respect to a set of program transformations.

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Mike Gordon. 2012. Background reading on Hoare Logic. https://www.cl.cam.ac.uk/archive/mjcg/HL/Notes/Notes.pdf.

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

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

A. Jeffrey and J. Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, M. Grohe, E. Koskinen, and N. Shankar (Eds.). ACM, 759–767. https://doi.org/10.1145/2933575.2934536

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. 2007. A Theory of Memory Models. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '07)*. ACM,

New York, NY, USA, 161–172. https://doi.org/10.1145/1229428.1229469

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

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

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

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

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

A THE PROMISING SEMANTICS ALLOWS OOTA1/OOTA4

We have confirmed that Lochbihler's [2013], OOTA1, is allowed by the promising semantics. Because the promising semantics does not have object creation, we used the variant OOTA4.

On Aril 3, 2018, we sent OOTA4 to Svendsen et al. [2018]. We present the example here exactly as it was in our correspondence:

```
Thread s:
  a:=x          // a==1
  y:=a
Thread t:
  b:=z          // b==1
  if (b==1)
    c:=y        // c==1
    x:=c
  else
    x:=1
Thread u:
  z:=1
```

The transitions of the threads are as follows:

```
s0 --R(x,v)--> s1 --W(y,v)--> s2
t0 --R(z,0)--> t1 --W(x,1)--> t2
t0 --R(z,1)--> t3 --R(y,v)--> t4 --W(x,v)--> t5
u0 --W(z,1)--> u1
```

To get the result, first execute `u` to get message `<z:1@1>`. Then `t` promises `<x:1@1>`, which it can fulfill by reading `b/z==0`. Then execute `s` to get message `<y:1@1>`. Then execute `t`, reading `b/z==1` and `c/y==1` and fulfill the promise by writing `<x:1@1>`.

On Aril 3, 2018, Svendsen et al. replied:

> *Yes, it is allowed in the way you described.*

On March 17, 2020, the authors confirmed that this did not change in Promising 2.0.

We sent the same example to Chakraborty and Vafeiadis [2019]. On December 10, 2018, they replied:

> *Yes the following outcome is allowed in our semantics. The event structure and the extracted execution are in pic-2 (attached).*

B PROOF OF COMPILATION FOR ARMV8

In this section, we develop the proof of correctness of compilation to ARMv8.

Our language can be translated to ARM following Podkopaev et al. [2019], thus using `ldr` for relaxed read, `ldar` for ra/sc reads are acquires, `str` for relaxed write, and `stlr` for ra/sc writes. `F` instruction can be translated to `dmb.sy`, since it has release-acquire semantics. Acquire fences map to `dmb.ld`, and release fences to `dmb.sy` — `dmb.st` does not provide order to prior reads.

Relative to the ARM specification, we have removed loops, and read-modify-write (RMW) operations. The implementation of RMW operations follows Podkopaev et al. [2019]. We only omit it because a systematic treatment includes a loop to account for the possible failure of the RMW operation.

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

The ARMv8 model is described using the following relations.

- [R], [W], [Acq], [Rel]: identity on reads, writes, acquires and releases.
- $[\mathcal{X}]$: relates any two events that touch the same location.
- po: program order.
- data, ctrl, addr: data, control and address dependencies.
- rf: reads-from. rf^{-1} relates each read to a matching write on the same location.
- co: coherence, which is a total order on the writes to a single location.
- $\mathsf{fr} \triangleq \mathsf{co}; \mathsf{rf}^{-1}$: from-read, which relates reads to subsequent writes.

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

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

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

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

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

$$\mathsf{bob} \triangleq [\mathsf{Acq}]; \mathsf{po} \cup \mathsf{po}; [\mathsf{Rel}]; \mathsf{coi}^? \qquad \text{(Barrier order)}$$

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

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

$\mathsf{codom}(\mathsf{rf}) = \mathsf{dom}(\mathsf{Rd})$	(rf-completeness)
For every location x, co totally orders the writes of x	(co-totality)
$\mathsf{poloc} \cup \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{co}$ is acyclic	(sc-per-loc)
ob is acyclic	(external)

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

From G we construct a candidate pomset P as follows:

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

The relation ob′ is defined from ob by restricting the order into and out of an 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′.

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

- $\Phi(e)$ implies $\Phi(d)$ whenever $d \leq e$. Trivial, since every formula is true.
- If e reads v from x, then there is some d such that
  - $d < e$,
  - d writes v to x, and
  - if c writes to x then either $c \leq d$ or $e \leq c$.

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

Lemma B.2. *Let e, d be distinct events and d' $(\xrightarrow{\mathsf{ob}} \cap \overset{\mathsf{po}}{\cdots\!\rightarrow})$ d $((\overset{\mathsf{eco}}{\cdots\!\rightarrow} \cap \overset{\mathsf{po}}{\cdots\!\rightarrow}) \setminus \xrightarrow{\mathsf{rfi}})$ e $(\xrightarrow{\mathsf{ob}} \cap \overset{\mathsf{po}}{\cdots\!\rightarrow})$ e'. Then $d' \xrightarrow{\mathsf{ob}} e'$.*

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

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

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

- If $d'\ (\xrightarrow{ob} \cap \overset{po}{\cdots\!\!\rightarrow})\ d (\xrightarrow{eco} \cap \overset{po}{\cdots\!\!\rightarrow}) e$ then $d' \xrightarrow{ob} e$.
- If $d\ (\xrightarrow{eco} \cap \overset{po}{\cdots\!\!\rightarrow})\ e (\xrightarrow{ob} \cap \overset{po}{\cdots\!\!\rightarrow}) e'$ then $d \xrightarrow{ob} e'$.

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

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

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

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

- If $e \xrightarrow{bob} e'$, then:
$$e\ ([\mathsf{Acq}]; \mathsf{po} \cup \mathsf{po}; [\mathsf{Rel}]; \mathsf{coi}^?)\ e'$$
Since e is not a read, $e(\mathsf{po}; [\mathsf{Rel}]; \mathsf{coi}^?)e'$. Result follows since $d \overset{po}{\cdots\!\!\rightarrow} e$.
- If $e \xrightarrow{dob} e'$, then e is a read. □

We now turn to proving that $(\mathsf{ob}' \cup \mathsf{eco})^*$ is acyclic.

It suffices to consider possible cycles in $(\mathsf{ob}' \cup \mathsf{eco})^*$ that do not involve the read events fulfilled by rfi. This is because the read events that are fulfilled by rfi have the following properties:

- Any in-edge into the event factors through an in-edge from an acquire or the fulfilling write to the read
- Any out-edge from the event factors through an out-edge to a release

Thus, if there is a cycle involving a read event fulfilled by rfi, there is also a cycle without such a read event.

In the rest of this section, we only consider events that are not read events fulfilled by rfi.

Lemma B.3. *If* $d \xrightarrow{ob'} e$ *then* $\neg(e \xrightarrow{eco} d)$.

Proof. Proof by contradiction. Let
$$e \xrightarrow{eco} d \xrightarrow{ob'} e$$
At least one of e, d is a write. So, if $e \not\overset{po}{\cdots\!\!\rightarrow} d$, then $e \xrightarrow{ob} d$, and we have a cycle in $\xrightarrow{ob}$.
So, we conclude that $e \overset{po}{\cdots\!\!\rightarrow} d$
Since $d \xrightarrow{ob'} e$, there exists $d \not\overset{po}{\cdots\!\!\rightarrow} c$, $d(\xrightarrow{eco} \cap \xrightarrow{ob'})c$, $c \xrightarrow{ob'} e$.
We reason by cases.

- If c is a write or e is a write, $c \xrightarrow{eco} e$ and we have an $\xrightarrow{eco}$ cycle.

- Otherwise, e is a read, d is a write.
  Let $d \xrightarrow{ob'} c_0 \xrightarrow{ob'} c_1 \ldots c_n \xrightarrow{ob'} e$ be the witness. If $c_n \overset{po}{\cdots\!\rightarrow} e$, then $c_n \xrightarrow{ob'} d$ and we have an $\xrightarrow{ob}$ cycle.
  So, $c_n \not\overset{po}{\cdots\!\rightarrow} e$. Thus, c_n is the write fulfilling e. So, we deduce: $c_n \overset{eco}{\dashrightarrow} e$ and $d \overset{eco}{\dashrightarrow} c_n$ yielding an $\overset{eco}{\dashrightarrow}$ cycle. □

Lemma B.4. $(ob' \cup eco)^*$ *is irreflexive.*

Proof. The simple case that $ob';eco$ is irreflexive is proved above. The full proof is by contradiction.

Let $n \geq 1$ be such that:

$$\begin{aligned} & e_0^0 \xrightarrow{ob'} e_1^0 \overset{eco}{\dashrightarrow} d_0^0 \xrightarrow{ob'} d_1^0 \\ (\overset{eco}{\dashrightarrow} \cap \xrightarrow{ob'})\ & e_0^1 \xrightarrow{ob'} e_1^1 \overset{eco}{\dashrightarrow} d_0^1 \xrightarrow{ob'} d_1^1 \\ (\overset{eco}{\dashrightarrow} \cap \xrightarrow{ob'})\ & \ldots \\ & \ldots d_1^n \\ (\overset{eco}{\dashrightarrow} \cap \xrightarrow{ob'})\ & e_0^0 \end{aligned}$$

where for all i, we have:

$$e_0^i \overset{po}{\cdots\!\rightarrow} e_1^i (\overset{eco}{\dashrightarrow} \cap \overset{po}{\cdots\!\rightarrow}) d_0^i \overset{po}{\cdots\!\rightarrow} d_1^i$$

and

$$\neg(d_1^i \overset{po}{\cdots\!\rightarrow} e_0^{(i+1) \mod n})$$

with the proviso that we have chosen the cycle with the minimum number of events.

For any i, if $e_0^i \neq e_1^i$ or $d_0^i \overset{po}{\cdots\!\rightarrow} d_1^i$, via lemma B.2, we deduce that $e_0^i \xrightarrow{ob} d_1^i$, contradicting minimality of number of events in cycle.

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

$$\begin{aligned} & e^0 \overset{eco}{\dashrightarrow} d^0 \\ (\overset{eco}{\dashrightarrow} \cap \xrightarrow{ob})\ & e^1 \overset{eco}{\dashrightarrow} d^1 \\ (\overset{eco}{\dashrightarrow} \cap \xrightarrow{ob})\ & \ldots \\ & \ldots d^n \\ (\overset{eco}{\dashrightarrow} \cap \xrightarrow{ob})\ & e^0 \end{aligned}$$

which is a contradiction since it is a cycle in $\overset{eco}{\dashrightarrow}$. □