Let Go of That Which Does Not Serve You

Semantic Independence and Multi-Copy Atomicity in a Model of Relaxed Memory

Anonymous Author(s)
Submission Id:

Abstract

The fundamental problem in the study of relaxed memory is to simultaneously achieve efficient implementability and thread-compositional reasoning. We define a model for C11-style concurrency that meets these goals, building on simple and traditional mathematical foundations that are well understood at LICS. We show that the 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 Intel and ARM microprocessors without requiring extra fences on relaxed operations.

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

Keywords Relaxed Memory Models, Temporal Safety Properties, Compiler Optimizations, Pomsets, Hoare Logic

ACM Reference Format:
Anonymous Author(s). 2020. Let Go of That Which Does Not Serve You: Semantic Independence and Multi-Copy Atomicity in a Model of Relaxed Memory. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 20 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." *None* of the extant memory models validate both "implementation flexibility" and "ease of use." This paper provides a solution.

There are two dimensions to "implementation flexibility." First, the model should be realizable on modern hardware with minimal synchronization. A canonical example is that the relaxed atomics of C11 (or the plain variables of Java) should not require any extra synchronization.

Second, the model should facilitate compiler transformations. Ideally, the model should support the known transformations used to optimize synchronization-free single-threaded code. A canonical example is that independent statements should commute.

There are also two dimensions 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—including those with data races—should support compositional reasoning on temporal safety properties [Abadi and Lamport 1993; Misra and Chandy 1981; Pnueli 1985; Stark 1985]. The following program "is type correct if it declares x, y and r of type $\mathtt{D}$. However, it has a legal execution [in Java] where they reference a $\mathtt{C}$ object" [Lochbihler 2013, Fig. 8]:

$$\begin{array}{ll} z{:=}1 \parallel y{:=}x & \\ \parallel r{:=}y;\ \mathtt{if}(z)\{s{:=}\mathtt{new\,C}\}\,\mathtt{else}\,\{r{:=}\mathtt{new\,D}\};\ x{:=}r & (*) \end{array}$$

Informally, all threads satisfy the invariant "allocation at type $\mathtt{C}$ is preceded by reading 1 for z" and "allocation at type $\mathtt{D}$ is preceded by reading 0 for z." If composability of safety were to hold, the full program would satisfy both invariants. Lochbihler argues that composability is necessary to prove type safety of racy Java programs without partitioning memory by type—an unrealistic assumption for any allocator.

Models that validate "ease-of-use" include 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]. However, all of these models invalidate reordering of independent statements. Several require extra fences after read actions in hardware implementations.

Java [Manson et al. 2005] and related models [Chakraborty and Vafeiadis 2019; Jagadeesan et al. 2010; Kang et al. 2017] are intended to validate more optimizations. However, all of these models invalidate compositional reasoning, as in (∗). Thus, they cannot support both type safety and realistic memory allocation.

Our approach has two key ingredients.

First, we weaken the program-order within a thread to capture only *essential dependencies.* Previous language models have used syntactic notions of dependency [Batty et al. 2011]. Instead, we embed logic formulae in events, using classical Hoare [1969] logic to compute dependencies between read

A note.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

and write events. Consider the following program fragments:

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

All these fragments satisfy $\{\mathsf{true}\}\ C_i\ \{y = 1\}$; 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.

Second, 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 partial order. We use *partially ordered multisets (pomsets)* [Gischer 1988; Plotkin and Pratt 1997], with the acyclicity of the pomset providing a global notion of time. Just as MCA dramatically simplifies the programmer model for hardware, this global notion of time dramatically simplifies our model for the language level.

We show that 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 DRF-SC criterion (§6), and
- validates single-threaded compiler optimizations (§7).

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.

In the main paper, we present the model, examples, the results concerning compositional reasoning and optimization, and a discussion of related work. The details of ARMV8/TSO-compilation and DRF-SC may be found in the appendix.

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. As is common for work on relaxed memory, we treat loops via unrolling: loops introduce complexities—such as liveness and continuity—that are orthogonal to the main topic of the paper.

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 Wr : $\mathcal{A} \rightarrow (\mathcal{X} \times \mathcal{V})$, and there are subsets of $\mathcal{A}$: Acq, Rel, SC, and Term.

We say that a is a *read* if $a \in \mathrm{dom}(\mathsf{Rd})$, a *write* if $a \in \mathrm{dom}(\mathsf{Wr})$, an *acquire* if $a \in \mathsf{Acq}$, a *release* if $a \in \mathsf{Rel}$, a *termination* if $a \in \mathsf{Term}$, and *SC* if $a \in \mathsf{SC}$. Note that these are *not* disjoint. When $\mathsf{Rd}(a) = (x, v)$, we say that *a reads v from x*, and similarly for writes.

We require that every SC read is an acquire, and every SC write is a release. We also require that termination events are releasing, but do not read, write, or acquire.

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

Example Language. Our example language includes actions of the form $(\checkmark)$, which is a *termination*, $(\mathsf{R}^{\mu} x v)$, which *reads* v from x and $(\mathsf{W}^{\mu} x v)$, which *writes* v to x. The *access mode* $(\mu ::= \mathsf{rlx} \mid \mathsf{ra} \mid \mathsf{sc})$ is either *relaxed*, *release-acquire*,

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

or *sequentially-consistent*. ra/sc reads are acquires, and ra/sc writes are releases.

We elide the rlx-mode annotation in examples.

We define the language by prefixing individual reads and writes. In §A we provide an equivalent semantics that supports full sequential composition of the form $(C; D)$.

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

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

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

Semantic domain. Our model is based on *partially ordered multisets* (*pomsets*) [Gischer 1988], ranged over by P and Q:

Definition 2.1. A *(memory model) pomset* is a tuple $(E, \le, \lambda)$:

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

We also refer pomsets as *executions*. We write pairs in $(\Phi \times \mathcal{A})$ as $(\phi \mid a)$, eliding ϕ when it is a tautology. We write $d < e$ when $d \le e$ and $d \ne e$. We identify pomsets up to isomorphism.

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 $\Phi(e)$."

We expect each pomset to have at most one termination event, which is ordered after all other events. A pomset is *completed* if it contains a termination.

The formula associated with an event is a *precondition*. The following commands gives rise to the pomset below them, capturing data and control dependencies.

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

r = 1 | Wy1
r < 0 | Wy1

(†)

In mapping the Hoare triple $\{\phi\}\ C\ \{\psi\}$ to our semantics, ψ represents a set of write actions, and ϕ represents their preconditions. For example, interpreting the pomset on the left above yields $\{r = 1\}\ y{:=}r\ \{y = 1\}$.

The ordering relation, $\le$, represents *causality*. The fourth requirement of the definition ensures *causal strengthening:* formulae do not weaken over time, as measured by $\le$.

The last requirement ensures *compatibility:* a pomset represents a single execution. For complete programs, we can restrict attention to events that are tautological. Richer formulae are needed for intermediate program fragments.

The semantics of programs is given as sets of pomsets. The sets of pomsets are closed with respect to *augmentation*, *prefixing*, *implication*, *disjunction* and *prefix weakening*:

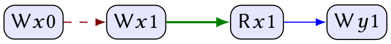
Definition 2.2. Let P' be an *augmentation* of P if $E' = E$, $\lambda' = \lambda$, and $\le' \supseteq \le$. Let P' be a *prefix* of P if $E \supseteq E' \supseteq \{d \in E \mid \exists e \in E'.\ d \le e\}$, $\le' = \le|_{E'}$, and $\lambda' = \lambda|_{E'}$.

Let $P' \in \text{pre}(\mathcal{P})$ if P' is a prefix of some $P \in \mathcal{P}$.

Let P' *imply* P if $E' = E$, $\le' = \le$, $\mathcal{A}' = \mathcal{A}$, and $\Phi'(e)$ implies $\Phi(e)$. For $i \in I$, let P' be a *disjunct* of P^i if $E' = E^i$, $\le' = \le^i$, $\mathcal{A}' = \mathcal{A}^i$, and $\Phi'(e)$ implies $\bigvee_i \Phi^i(e)$. Let P^2 be a *prefix weakening* of P^1 by $P^{2\prime} \in \text{pre}(P^2)$ if P^1 implies P^2 and there is some $P^{1\prime} \in \text{pre}(P^1)$ such that $P^{1\prime}$ implies $P^{2\prime}$.

Each pomset in the semantics of a program has at most one termination event, which is ordered after all other events. By causal strengthening, its precondition must therefore imply the precondition of all other events.

We visualize pomsets as directed graphs. In examples, we draw pomsets that are *completed*, *augmentation-minimal* and *implication-minimal*. We elide the uninteresting termination event. For example, the semantics of $\texttt{var}\, x; (x{:=}0; x{:=}1 \parallel y{:=}x)$ includes:



We visualize order using arrows that indicate the reason that the order arises. $(\mathsf{W}x0) \dashrightarrow (\mathsf{W}x1)$ is a *coherence* requirement: the write of 1 must follow the write of 0, since these are in *conflict* and in program order. $(\mathsf{W}x1) \rightarrow (\mathsf{R}x1)$ is a *reads-from* requirement: the read of x must be *fulfilled* by a matching write. $(\mathsf{R}x1) \rightarrow (\mathsf{W}y1)$ is a *dependency* requirement: the write to y *depends on* the read of x. Although we use multiple arrows, we emphasize that they are all part of the same $\le$ relation.

Semantics of the Example Language. In the remainder of this section, 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 discuss the other operators first.

Definition 2.3. Let $(\mathcal{P}\sigma)$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ when there is $P \in \mathcal{P}$ such that: $E' = E$, $\le' = \le$, $\mathcal{A}'(e) = \mathcal{A}(e)$, and $\Phi'(e) = \Phi'(e)\sigma$.

Let $P \in \text{STOP}$ when $E = \{e\}$ and $\mathcal{A}(e) = \checkmark$.

$$[\![r{:=}M; C]\!] = [\![C]\!][M/r] \qquad [\![\texttt{skip}]\!] = \text{STOP}$$

Definition 2.4. We say d *fulfills* e *on* x if

- d writes v to x,
- e reads v from x,
- $d < e$, and
- if c writes to x then either $c \le d$ or $e \le c$.

A pomset is x-*closed* if every read on x is fulfilled, and every formula is independent of x ($\forall v.\ \phi \vDash \phi[v/x] \vDash \phi$).

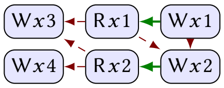
A pomset is *top-level* if it is x-closed for every location x.

Let $(\nu x.\mathcal{P})$ be $\mathcal{P}'\subseteq\mathcal{P}$ such that $P'\in\mathcal{P}'$ when P' is x-closed.

$$[\![\mathtt{var}\, x\,;\, C]\!] = \nu x\,.\,[\![C]\!]$$

When a location is bound, every read of that location must be *fulfilled* by a matching write. At top-level, fulfillment imposes a total order on conflicting writes that are read; it does not impose order between unread writes:

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



The restriction of pomset order to conflicting events is called the *extended coherence order* (eco). This relation can always be extended to totally order all conflicting events, as is common in hardware memory models.

The definition of location binding validates *scope extrusion* [Milner 1999]: if C does not mention x then $[\![C \parallel \mathtt{var}\, x\,;\, D]\!] = [\![\mathtt{var}\, x\,;\, (C \parallel D)]\!]$. However, the definition does not validate renaming of locations: if $x \neq y$ then $[\![\mathtt{var}\, y\,;\, C]\!] \neq [\![\mathtt{var}\, x\,;\, C[x/y]]\!]$, even if C does not mention x. This is consistent with support for address calculation, which is required by realistic memory allocators.

Definition 2.5. 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:

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

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

Concurrent composition is roughly union. Because of the compatibility requirement in Definition 2.1, we do not include events with contradictory preconditions. Consider:

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

$$\boxed{r < 0 \mid \mathsf{W}\,y\,1} \qquad\qquad\qquad \boxed{r \geq 0 \mid \mathsf{W}\,y\,1}$$

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:

$$\boxed{r < 0 \vee r \geq 0 \mid \mathsf{W}\,y\,1}$$

Coalesced events inherit order from both sides.

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

Definition 2.6. Let $(\phi \rhd \mathcal{P})$ be the set $\mathcal{P}' \subseteq \mathcal{P}$ such that $P' \in \mathcal{P}'$ when ϕ implies $\Phi(e')$, for every $e' \in E'$.

$$[\![\mathtt{if}(M)\{C\}\,\mathtt{else}\,\{D\}]\!] = \big(M \rhd [\![C]\!]\big) \parallel \big(\neg M \rhd [\![D]\!]\big)$$

Conditional execution is defined using *filtering*, which selects the subset of pomsets that imply a formula, and composition, which allows coalescing using disjunction, as discussed above. This reflects the Hoare rule for conditionals:

$$\frac{\{M\}\ C\ \{\psi\} \qquad \{\neg M\}\ D\ \{\psi\}}{\{M \vee \neg M\}\ \mathtt{if}(M)\{C\}\,\mathtt{else}\,\{D\}\ \{\psi\}}$$

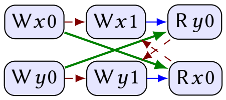
Program Order Prefixing. We present several candidate definitions for prefixing before giving the final definition. The candidates are progressively more general and less ordered. We begin by considering programs with trivial expressions, including all of program order in pomset order. To simplify the definition, we construct the set of pomsets with the new action, then prefix close.

Candidate 2.7. Let $(\phi \mid a) \overset{\mathsf{triv}}{\Longrightarrow} \mathcal{P}$ be the set $\mathrm{pre}(\mathcal{P}')$ 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.

$$\begin{aligned}
[\![r := x^{\mu}\,;\, C]\!] &= \textstyle\bigcup_v\ (\mathsf{R}^{\mu} x v) \overset{\mathsf{triv}}{\Longrightarrow} [\![C]\!]\\
[\![x^{\mu} := v\,;\, C]\!] &= \mathsf{W}^{\mu} x v \overset{\mathsf{triv}}{\Longrightarrow} [\![C]\!]
\end{aligned}$$

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:

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



(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 2.7.

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 2.8. Let $(\phi \mid a) \overset{\mathsf{sc}}{\Rightarrow} \mathcal{P}$ be the set $\mathrm{pre}(\mathcal{P}')$ 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 2.9.

$$\llbracket r := x^{\mu}\,;\, C\rrbracket = \textstyle\bigcup_{v}\, (\mathsf{R}^{\mu}xv) \overset{\mathsf{sc}}{\Rightarrow} \llbracket C\rrbracket\ [x/r]$$
$$\llbracket x^{\mu} := M\,;\, C\rrbracket = \ \|_{v}\ (M = v \mid \mathsf{W}^{\mu}xv) \overset{\mathsf{sc}}{\Rightarrow}\ \llbracket C\rrbracket$$

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 $\llbracket x := 1{+}r{*}r{-}r\rrbracket$ includes both of the pomsets:

$$\boxed{r{=}0 \mid \mathsf{W}x1} \qquad \boxed{r{=}1 \mid \mathsf{W}x1}$$

By using $\|_{v}$ in the definition of write, it also includes:

$$\boxed{r{=}0 \vee r{=}1 \mid \mathsf{W}x1}$$

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*: $\llbracket C\rrbracket = \llbracket \mathtt{if}(M)\{C\}\,\mathtt{else}\,\{C\}\rrbracket$.

For reads, item 4b allows some preconditions to weaken and requires others to strengthen. Recall (†):

$$y := r \qquad\qquad \mathtt{if}(r < 0)\{y := 1\}$$
$$\boxed{r = 1 \mid \mathsf{W}y1} \qquad\qquad \boxed{r < 0 \mid \mathsf{W}y1}$$

Prepending $r := x$ first causes the substitution $[x/r]$:

$$r := x;\ y := r \qquad\qquad r := x;\ \mathtt{if}(r < 0)\{y := 1\}$$
$$(\mathsf{R}x1) \overset{\mathsf{sc}}{\Rightarrow} \boxed{x = 1 \mid \mathsf{W}y1} \qquad\qquad (\mathsf{R}x1) \overset{\mathsf{sc}}{\Rightarrow} \boxed{x < 0 \mid \mathsf{W}y1}$$

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

$$\boxed{\mathsf{R}x1} \rightarrow \boxed{1 = 1 \mid \mathsf{W}y1} \qquad\qquad \boxed{\mathsf{R}x1} \quad \cancel{\boxed{1 < 0 \mid \mathsf{W}y1}} \qquad (\ddagger)$$

On the right, $(\mathsf{W}y1)$ has become impossible. 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 $\|_{v}$.

General Prefixing. We now relax item 5 of Definition 2.8 so that only some program order is preserved in the pomset, arriving at the final definition of the prefix operator:

Definition 2.10. Two actions *conflict* if one writes a location and the other either reads or writes the same location.

Let $(\phi \mid a) \Rightarrow \mathcal{P}$ be the set $\mathrm{pre}(\mathcal{P}')$ where $P' \in \mathcal{P}'$ when there is some $P \in \mathcal{P}$ that satisfies items 1-4 of Definition 2.8 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 $d <' e$,
5d. if d is an SC write and e is an SC read, then $d <' e$, and

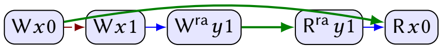
Candidate 2.11.

$$\llbracket r := x^{\mu}\,;\, C\rrbracket = \textstyle\bigcup_{v}\, (\mathsf{R}^{\mu}xv \Rightarrow \llbracket C\rrbracket[x/r]$$
$$\llbracket x^{\mu} := M\,;\, C\rrbracket = \ \|_{v}\ (M = v \mid \mathsf{W}^{\mu}xv \Rightarrow \llbracket C\rrbracket\ [M/x]$$

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 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:

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

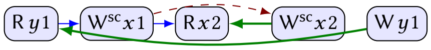


By Definition 2.4, $(\mathsf{R}x0)$ is *unfulfilled* in this pomset. It fails the last requirement of the definition, since $(\mathsf{W}x0) \le (\mathsf{W}x1) < (\mathsf{R}x0)$. In order to satisfy this requirement, $(\mathsf{R}x0)$ must be ordered before $(\mathsf{W}x1)$, but this creates a cycle.

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:

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



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

We relax program order on non-SC accesses in order to allow outcomes like that of execution candidate (SB). Order is

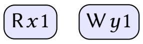
[2] When d is not a read, 4c trivially implies 5a.

[3] Recall that termination actions are releases.

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

relaxed between reads, between writes to different locations, and from a read to an independent write:

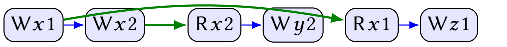
$$C = r := x;\ \mathtt{if}(r)\{y := 1\}\,\mathtt{else}\,\{y := 1\}$$



Let C be the program above. The existence of this pomset is justified by the triple $\{\text{true}\}\ C\ \{y = 1\}$.

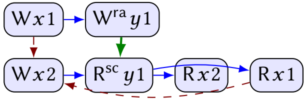
Unordered actions can be scheduled freely. As a result, our model of coherence is similar to that of Dolan et al. [2018]. Since reads are not ordered by 5b, we allow the following, which C11 forbids:

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



Conversely, we forbid the following, which Java allows:

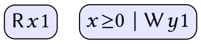
$$x := 1;\ y^{\mathsf{ra}} := 1 \parallel x := 2;\ r := y^{\mathsf{sc}};\ r := x;\ r := x$$



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

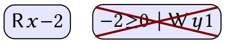
Item 5a imposes order from read to write when weakening the precondition of the write via 4b, as on the left hand side of (‡). Item 4b *allows* a precondition to weaken, but does not *require* it. Item 5a only requires order when the precondition weakens. Thus, no order is required in:

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

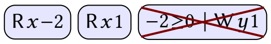


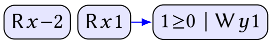
Let C be the program above. The existence of this pomset is justified by the triple $\{x \geq 0\}\ C\ \{y = 1\}$. It is not justified by the value of the read action.

Nonetheless, item 4b requires that the value of the read action must be *compatible* with subsequent formulae. In this example, the write precondition must become unsatisfiable when -2 is read from x:



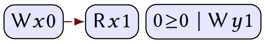
This holds for all preceding reads, unless the precondition is discharged by introducing order. Thus, $[\![s := x;\ C]\!]$ contains both of the following pomsets:



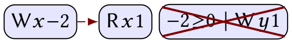


Like item 4b, which substitutes $[v/x]$ during a read, Candidate 2.11) substitutes $[M/x]$ during a write. Like 4b, this affects subsequent preconditions, either allowing them to weaken, or requiring them to strengthen. For write prefixing, however, there is no rule corresponding to item 5a. Unlike a read event, order is *not* imposed from a write event to the subsequent events whose precondition it weakens:

$x := 0;\ C$

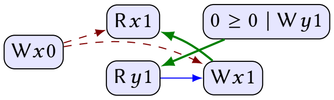


$x := -2;\ C$



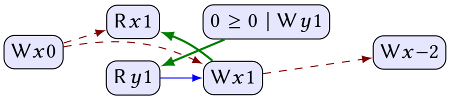
In the JMM causality test cases [Pugh 2004], such executions are justified via compiler analysis, possibly in collusion with the scheduler: If every observed value can be shown to satisfy a precondition, then the precondition can be dropped. For example, TC1 determines that the following top-level execution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

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



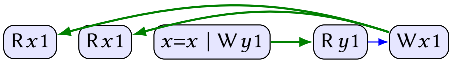
Unlike [Jeffrey and Riely 2016], our semantics is robust with respect to the introduction of concurrent writes, as in TC9:

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



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

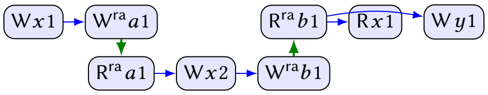
$$r := x;\ s := x;\ \mathtt{if}(r{=}s)\{y := 1\} \parallel x := y$$



Note that in the prefix $[\![s := x;\ \mathtt{if}(r{=}s)\{y := 1\}]\!]$, the precondition on $(\mathsf{W}y1)$ 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 create an event that must be fulfilled. In combination, these ensure that a write substitution cannot be used to ignore a blocking write. In the following execution candidate, there is no order from $(\mathsf{R}x1)$ to $(\mathsf{R}y1)$, potentially allowing the program to write a stale value. However, $(\mathsf{R}x1)$ cannot be fulfilled, causing the execution candidate to be disallowed:

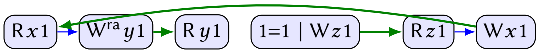
$$x := 1;\ a^{\mathsf{ra}} := 1;\ \mathtt{if}(b^{\mathsf{ra}})\{y := x\} \parallel \mathtt{if}(a^{\mathsf{ra}})\{x := 2;\ b^{\mathsf{ra}} := 1\}$$



Note that we change $(\mathsf{R}x1)$ to $(\mathsf{R}x2)$ then the precondition of $(\mathsf{W}y1)$ must imply 2=1.

As a final example in this vein, consider Example 3.6 of Podkopaev et al. [2019]:

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



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

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

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 2.12. Let $(\mathrm{cover}^{\mathsf{rlx}}\, x \rhd \mathcal{P})$ 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 $(\mathrm{cover}^{\mathsf{ra}}\, x \rhd \mathcal{P}) = (\mathrm{cover}^{\mathsf{sc}}\, x \rhd \mathcal{P}) = \emptyset$.

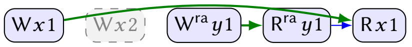
$$[\![r := x^\mu;\ C]\!] = \textstyle\bigcup_v\ (\mathsf{R}^\mu x v) \Rightarrow [\![C]\!][x/r]$$

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

$$\cup\ (\mathrm{cover}^\mu\, x \rhd [\![C]\!][M/x])$$

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:

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

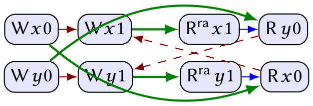


In this drawing, we have included a "non-event"—dashed border—to mark the eliminated write.

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.

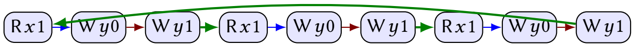
$$x := 0;\ x := 1 \ \parallel r := x^{\mathsf{ra}};\ s := y$$
$$\parallel\ y := 0;\ y := 1 \ \parallel r := y^{\mathsf{ra}};\ s := x$$



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*.

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



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

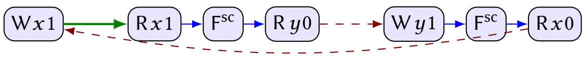
$$[\![\mathsf{F}^\nu;\ C]\!] = (\mathsf{F}^\nu) \Rightarrow [\![C]\!]$$

The semantics of syntactic fences is subtly different from the fencing behavior of ra/sc-accesses in C11. The additional order of syntactic fences is captured by adding the following to Definition 2.10 of *prefixing*:

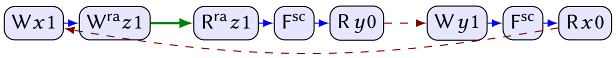
5e. if d reads and $\mathcal{A}(e) = (\mathsf{F}^{\mathsf{acq}})$, then $e <' d$, and
5f. if $\mathcal{A}(d) = (\mathsf{F}^{\mathsf{rel}})$ and e writes, then $d <' e$.

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:

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



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



The executions are disallowed, due to cycles.

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.

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

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

Let us revisit the discussion of the use of ∥ in Candidate 2.9. Note that $[\![a[r] := 0;\ a[0] := !r]\!]$ includes both of the following pomsets ("$!M$" evaluates to 1 if M is 0, and 0 otherwise):

$r{=}0 \mid \mathsf{W}a[0]0$ $r{=}0 \mid \mathsf{W}a[0]1$ $r{=}1 \mid \mathsf{W}a[1]0$ $r{=}1 \mid \mathsf{W}a[0]0$

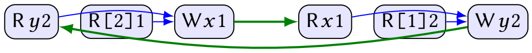
By using ∥, it also includes:

$r{=}0 \vee r{=}1 \mid \mathsf{W}a[0]0$ $r{=}0 \mid \mathsf{W}a[0]1$ (∗∗)

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

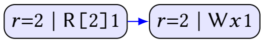
Because we do not enforce order between reads, there is some danger that address calculations could introduce anomalous behaviors that arise *out of thin air* (OOTA) [Batty et al. 2015]. Consider the following attempted execution, where all memory addresses are initialized to 0, except that `[2]` is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

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



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 $d \leq e$ 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 $(E, \leq, \lambda, \xrightarrow{\text{rmw}})$, where $\xrightarrow{\text{rmw}} \subseteq \leq$ relates the two events of a successful RMW. Additionally, require that:

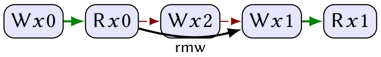
- If c, e write the same x, $c \leq e$ and $d \xrightarrow{\text{rmw}} e$ then $c \leq d$.
- If c, e write the same x, $d \leq c$ and $d \xrightarrow{\text{rmw}} e$ then $e \leq c$.

When selecting $E' \subseteq E$ in the definition of prefix (Definition 2.2), we must ensure both downclosure—including $\{d \in E \mid \exists e \in E'.\ d \leq e\}$—and RMW closure—including $\{d \in E \mid \exists e \in E'.\ e \xrightarrow{\text{rmw}} d\}$.

We elide the obvious and tedious semantic rules.

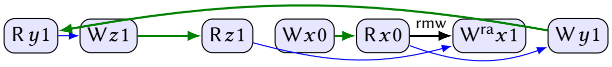
This definition ensures atomicity, disallowing executions such as [Podkopaev et al. 2019, Ex. 3.2]:

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



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]:

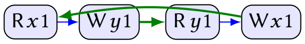
$$r := y;\ z := r \parallel r := z;\ x := 0;\ s := \mathsf{FADD}^{\mathsf{rlx,ra}}(x);\ y := s{+}1$$



4 Invariant Reasoning in 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:

$$(y := x \parallel r := y;\ x := r)$$



(††)

Although Java does not allow OOTA behaviors of (††), Lochbihler [2013] showed that it does allow OOTA behaviors of (∗), from §1. Jeffrey and Riely [2016] described a logic that rules out (††) but not (∗). In this section, we provide a more accurate test of OOTA behaviors by enhancing their logic with temporal features. The logic is not meant to be definitive; in §9, we discuss OOTA examples that require non-trivial extensions.

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]:

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

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 *prefix closed* 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:

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

We present two proof rules for programs. The first provides a logical view of x-*closure* (Definition 2.4):

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

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 prefix-closed. 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 prefixes in the prefix closures of $\mathcal{P}_1 \parallel \mathcal{P}_2$ satisfy the required property. Proof proceeds by induction on prefixes of $P \in \mathcal{P}_1 \parallel \mathcal{P}_2$. The case for empty prefix follows from assumption that ϱ is prefix closed. For the inductive case, consider $P \in P_1 \parallel P_2$ where $P_i \in \mathcal{P}_i$.

[5] Let false, ∨, ⇒ and $\diamondminus$ as usual; for example, $\diamondminus\varrho = \neg(\boxminus\neg\varrho)$.

[6] The order-minimal elements always validate $\boxminus\varrho$ and invalidate $\diamondminus\varrho$.

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

When all variables are initially bound to 0, we show that (††) satisfies $\neg \mathsf{W}x1$. We start with the invariant:

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

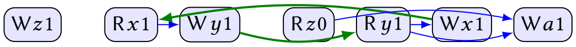
This invariant holds for each thread; thus, it holds for the aggregate program by composition. Closing y yields $\mathsf{R}y1 \Rightarrow \diamondminus \mathsf{W}y1$. Weakening the right conjunct: $\diamondminus \mathsf{W}y1 \Rightarrow \diamondminus \mathsf{R}x1$. Chaining these together: $\mathsf{R}y1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

We consider a variant of Lochbihler's example (∗):

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

This variant retains the essential temporal aspects of (∗). In this case, all threads satisfy the invariant "a write to a is preceded by a read of z as 1."

Attempting to write 1 to a results in a cycle:



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

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

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

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

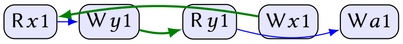
Closing x: $\mathsf{W}a1 \Rightarrow (\diamondminus \mathsf{W}x1 \wedge \boxminus(\mathsf{W}x1 \Rightarrow \diamondminus \mathsf{W}x1))$. Applying coinduction to the right conjunct:

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

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

Many examples are superficially similar, but in fact have fewer dependencies. In the following, $\mathsf{W}x1$ is independent of $\mathsf{R}y1$. We discuss another such example at the end of §7.

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



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 2.4. But this does not quite work.

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

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

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 Data Race Free Executions are Sequentially Consistent

When constructing a pomset, define *program order* ($\xrightarrow{\mathsf{po}}$) in the obvious way. Define $d \xrightarrow{\mathsf{sw}} e$ exactly when d fulfills e, d is a release, e is an acquire, and $\neg(d \xrightarrow{\mathsf{po}} e)$. Define $\xrightarrow{\mathsf{hb}} = (\xrightarrow{\mathsf{po}} \cup \xrightarrow{\mathsf{sw}})^*$. A pomset has a *data race* if there are conflicting events that are unordered by hb.

Note that the definition of a data-race does not mention *pomset* order. Instead, it relies on the auxiliary definition of *program* order. Thus it is stable with respect to augmentation.

Let a *generator* for C be a top-level pomset that is minimal with respect to augmentation and implication. Let $[\![C]\!]_{\mathsf{SC}}$ be the executions as defined in Candidate 2.11.

We prove the following theorem in §D.

Theorem 6.1. *Let P be a generator for C. (a) If P does not have a data race, then $P \in [\![C]\!]_{\mathsf{SC}}$. (b) If P has a data race, then there is some $P' \in [\![C]\!]_{\mathsf{SC}}$ that also has a data race.*

7 Single-Threaded Optimizations

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, we isolate a *linear* fragment of our language that ensures these restrictions. We then show the soundness of *all* transformations of synchronization-free sequential programs into this fragment. We end this section with a discussion of specific optimizations, some of which relax the linearity assumption and include synchronization.

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 $P \in [\![\phi \land \chi \mid \psi]\!]$ 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.

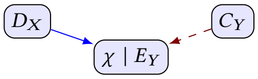
Require that $\phi\sigma$ and $\psi\pi$ are satisfiable.

Require that $\Phi(e_y)$ implies χ.

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.

In addition, Hoare postconditions are properties of *completed* executions. For example, in $\{\mathsf{true}\}\ x := 1;\ x := 2\ \{x{=}2\}$, the postcondition does not hold for the prefix $x := 1$. As a result, we also define $\mathsf{read}(\mathcal{P})$ to restrict attention to completed executions.

Let $\mathsf{read}(\mathcal{P})$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ when there is a completed pomset $P \in \mathcal{P}$ and some set 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. *Let C be synchronization-free and sequential. Then $\{\phi\}\ C\ \{\psi\}$ if and only if $[\![\phi \mid \psi]\!] \cap \mathsf{read}[\![C]\!] \neq \emptyset$.*

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

We first consider the structural rules. Precondition strengthening follows from augmentation closure. The proof that the structural rule of disjunction:

$$\frac{\{\phi_1\}\ C\ \{\psi_1\}, \{\phi_2\}\ C\ \{\psi_2\}}{\{\phi_1 \lor \phi_2\}\ C\ \{\psi_1 \lor \psi_2\}}$$

holds follows from closure of the semantics under disjuncts. The proof for the structural rule of conjunction:

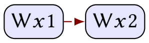
$$\frac{\{\phi_1\}\ C\ \{\psi_1\}, \{\phi_2\}\ C\ \{\psi_2\}}{\{\phi_1 \land \phi_2\}\ C\ \{\psi_1 \land \psi_2\}}$$

follows from the fact that pomsets have only concurrency and no conflict.

The remaining cases that use the rules for deducing Hoare triples by structural induction on the command follow directly from the semantics. The only subtleties are in the write rule, which uses $\parallel$ to ensure disjunction closure. □

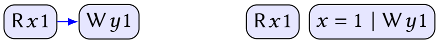
To illustrate multiple writes consider:

$$\{\mathsf{true}\}\ x := 1;\ x := 2\ \{x = 2\}$$



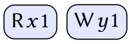
Preconditions can be placed in ϕ or χ in Definition 7.1, resulting in different pomsets:

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



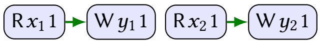
Control dependencies are calculated correctly:

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



For any compatible set of preconditions, we can always find a single pomset that includes all of the required writes.

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



Corollary 7.3. *If $\bigwedge_{i \in I} \phi_i$ is satisfiable:*

$$\bigwedge\nolimits_{i \in I} \{\phi_i\}\ C\ \{\psi_i\} \iff \bigcap\nolimits_{i \in I} [\![\phi_i \mid \psi_i]\!] \cap \mathsf{read}[\![C]\!] \neq \emptyset.$$

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* $\mathsf{read}[\![C]\!] \supseteq \mathsf{read}[\![C']\!]$.

Valid Rewrites. When $\mathsf{read}\llbracket C\rrbracket \supseteq \mathsf{read}\llbracket C'\rrbracket$, 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 allow for richer contexts, we appeal to the alternate presentation of the language given in §A. We refactor the syntax of commands and define contexts:

$$\begin{aligned} C, D &::= \texttt{skip} \mid \mathsf{F}^{\nu} \mid r\texttt{:=}M \mid r\texttt{:=}\texttt{[}L\texttt{]}^{\mu} \mid \texttt{[}L\texttt{]}^{\mu}\texttt{:=}M \\ &\quad\mid C \parallel D \mid C\texttt{;}\,D \mid \texttt{var}\,x\texttt{;}\,C \mid \texttt{if}(M)\{C\}\,\texttt{else}\,\{D\} \\ \mathbb{C}, \mathbb{D} &::= [-] \mid \mathbb{C} \parallel D \mid C \parallel \mathbb{D} \mid \mathbb{C}\texttt{;}\,D \mid C\texttt{;}\,\mathbb{D} \mid \texttt{var}\,x\texttt{;}\,\mathbb{C} \\ &\quad\mid \texttt{if}(M)\{\mathbb{C}\}\,\texttt{else}\,\{D\} \mid \texttt{if}(M)\{C\}\,\texttt{else}\,\{\mathbb{D}\} \end{aligned}$$

Lemma 7.5. *Let $\mathbb{D}$ be a context and $\mathsf{read}\llbracket C\rrbracket \supseteq \mathsf{read}\llbracket C'\rrbracket$:*

$$\mathsf{read}\llbracket \mathbb{D}[C]\rrbracket \supseteq \mathsf{read}\llbracket \mathbb{D}[C']\rrbracket$$

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 $\texttt{[}M\texttt{]}$ and $\texttt{[}N\texttt{]}$, 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 (RL), store forwarding (SF), dead store (DS), and independent reorderings. Using the semantics directly, we can prove some properties without using read. Note that if $\mathcal{P}' \supseteq \mathcal{P}$, then $\mathsf{read}(\mathcal{P}') \supseteq \mathsf{read}(\mathcal{P})$.

$$\mathsf{read}\llbracket r\texttt{:=}x\texttt{;}\,s\texttt{:=}x\rrbracket \supseteq \mathsf{read}\llbracket r\texttt{:=}x\texttt{;}\,s\texttt{:=}r\rrbracket \tag{RL}$$
$$\mathsf{read}\llbracket x\texttt{:=}M\texttt{;}\,s\texttt{:=}x\rrbracket \supseteq \mathsf{read}\llbracket x\texttt{:=}M\texttt{;}\,s\texttt{:=}M\rrbracket \tag{SF}$$
$$\llbracket x\texttt{:=}M\texttt{;}\,x\texttt{:=}N\rrbracket \supseteq \llbracket x\texttt{:=}N\rrbracket \tag{DS}$$
$$\llbracket x\texttt{:=}M\texttt{;}\,y\texttt{:=}N\rrbracket = \llbracket y\texttt{:=}N\texttt{;}\,x\texttt{:=}M\rrbracket \tag{WW}$$
$$\llbracket r\texttt{:=}x\texttt{;}\,y\texttt{:=}N\rrbracket = \llbracket y\texttt{:=}N\texttt{;}\,r\texttt{:=}x\rrbracket \tag{RW}$$
$$\llbracket r\texttt{:=}x\texttt{;}\,s\texttt{:=}y\rrbracket = \llbracket s\texttt{:=}y\texttt{;}\,r\texttt{:=}x\rrbracket \tag{RR}$$

(RR) requires either $r \neq s$ or $x = y$. (WW) and (RW) require that two sides of the semicolon have disjoint ids; for example, (RW) requires $\mathsf{id}(r\texttt{:=}x) \cap \mathsf{id}(y\texttt{:=}N) = \emptyset$. (RL) and (SF) follow from read closure. (DS) follows from the write elimination allowed in Definition 2.12.

Since reads are unordered in our model, read optimizations are not limited by the power of aliasing analysis, as they are with stronger models of coherence [Pugh 1999, §2.3]. Composing (RR) and (RL), for $r_2 \neq s$ we have:

$$\mathsf{read}\llbracket r_1\texttt{:=}x\texttt{;}\,s\texttt{:=}y\texttt{;}\,r_2\texttt{:=}x\rrbracket \supseteq \mathsf{read}\llbracket r_1\texttt{:=}x\texttt{;}\,s\texttt{:=}y\texttt{;}\,r_2\texttt{:=}r_1\rrbracket$$

This holds regardless of whether $x = y$.

By induction on the length of the pomsets in C, we can use the reorderings to establish, more generally, that when C and D are assignment sequences and $\mathsf{id}(C) \cap \mathsf{id}(D) = \emptyset$:

$$\llbracket C\texttt{;}\,D\rrbracket = \llbracket D\texttt{;}\,C\rrbracket$$

The semantics also validates roach-motel reorderings. Let C be synchronization-free, with disjoint ids as before:

$$\llbracket C\texttt{;}\,s\texttt{:=}x^{\mathsf{ra}}\rrbracket \supseteq \llbracket s\texttt{:=}x^{\mathsf{ra}}\texttt{;}\,C\rrbracket \tag{A}$$
$$\llbracket x^{\mathsf{ra}}\texttt{:=}M\texttt{;}\,C\rrbracket \supseteq \llbracket C\texttt{;}\,x^{\mathsf{ra}}\texttt{:=}M\rrbracket \tag{R}$$

As expected, sequential and parallel composition commute with conditionals and location binding, and conditionals and location binding commute with each other. We show sequential scope extrusion (SSE), which concerns sequential composition and location binding:

$$\llbracket C\texttt{;}\,\texttt{var}\,x\texttt{;}\,D\rrbracket = \llbracket \texttt{var}\,x\texttt{;}\,(C\texttt{;}\,D)\rrbracket \tag{SSE}$$

(SSE) requires that x does not appear in C.

Many laws hold for the conditional. We show case analysis (CA) and dead code elimination (DC).

$$\llbracket \texttt{if}(M)\{C\}\,\texttt{else}\,\{C\}\rrbracket = \llbracket C\rrbracket \tag{CA}$$
$$\llbracket \texttt{if}(M)\{C\}\,\texttt{else}\,\{D\}\rrbracket = \llbracket C\rrbracket \tag{DC}$$

(DC) requires that M be a tautology. The correctness of (CA) follows from disjunction closure (Definition 2.2).

Invalid Rewrites. Relevant read introduction is invalid:

$$\llbracket r\texttt{:=}x\texttt{;}\,\texttt{if}(r{\neq}r)\{z\texttt{:=}1\}\rrbracket \not\supseteq \llbracket r\texttt{:=}x\texttt{;}\,s\texttt{:=}x\texttt{;}\,\texttt{if}(r{\neq}s)\{z\texttt{:=}1\}\rrbracket$$

These are distinguished by the context $[-] \parallel x\texttt{:=}1 \parallel x\texttt{:=}2$.

Write introduction is invalid, even for equal values:

$$\llbracket x\texttt{:=}1\texttt{;}\,C\rrbracket \not\supseteq \llbracket x\texttt{:=}1\texttt{;}\,x\texttt{:=}1\texttt{;}\,C\rrbracket$$

These are distinguished by the context:

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

With weaker notions of coherence [Manson et al. 2005], these commands are indistinguishable.

Thread inlining is invalid (see §2): $\llbracket C \parallel D\rrbracket \not\supseteq \llbracket C\texttt{;}\,D\rrbracket$.

Boehm [2018] considers the following programs:

$$\llbracket r\texttt{:=}y\texttt{;}\,x\texttt{:=}r\rrbracket \not\supseteq \llbracket r\texttt{:=}y\texttt{;}\,\texttt{if}(r{\neq}1)\{z\texttt{:=}1\texttt{;}\,r\texttt{:=}1\}\texttt{;}\,x\texttt{:=}r\rrbracket$$

The left command is half of the OOTA example from §4 (††). The right command is dubbed RFUB, for *Register assignment From an Unexecuted Branch*. Boehm observes that in the context $x\texttt{:=}y \parallel [-]$, 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 OOTA 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 §9.

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 the three-location execution at the end of §2, but disallow all two-location variants. 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]. See Jeffrey and Riely [2019] for a discussion.

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 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.

First, 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 Sevcík and Aspinall [2008] discovered that several sequential compiler optimizations were invalid in the Java Memory Model [Manson et al. 2005].

Second, 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].

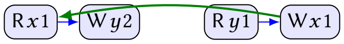
These contributions rest on a semantics that supports a simple, executable prescription of *essential dependencies*. This provides a clear interface between the language user and the compiler writer, and a clear bridge from the compiler writer to computer architecture.

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.

Ours is the first language-level model to incorporate *multi-copy atomicity* (MCA). However, our main contributions do not depend on MCA. It is possible to apply our ideas to stronger models, and to weaker ones. For stronger models, such as Boehm and Demsky [2014] and Dolan et al. [2018], it is likely sufficient to require more order in the pomset. To compile efficiently to weaker architectures, such as POWER and ARMV7, it appears necessary to use a model with more than one ordering relation.

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:

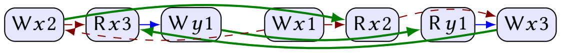
$(y := x + 1 \parallel x := y)$



The attempted execution is disallowed since there is no write to fulfill ($\mathsf{R}\,y1$). This example requires the ability to reason about values.

As another example, the following outcome is allowed by the promising semantics [Kang et al. 2017], but not by WEAKESTMO [Chakraborty and Vafeiadis 2019, Fig. 3] nor in our semantics, due to the cycle:

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



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.

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

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

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

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

Eike Ritter and Valeria de Paiva. 1997. On Explicit Substitution and Names (Extended Abstract). In *Automata, Languages and Programming, 24th International Colloquium, ICALP'97, Bologna, Italy, 7-11 July 1997, Proceedings (Lecture Notes in Computer Science)*, Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela (Eds.), Vol. 1256. Springer, 248–258. https://doi.org/10.1007/3-540-63165-8_182

B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

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.

Jaroslav Sevcík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 5142. Springer, 27–51. https://doi.org/10.1007/978-3-540-70592-5_3

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

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

A Sequential Composition

We provide an alternative semantics that supports full sequential composition, building $[\![C;D]\!]$ from $[\![C]\!]$ and $[\![D]\!]$. To simplify the definitions[7], we assume, without loss of generality [Rosen et al. 1988], that each register is assigned at most once syntactically. Since we exclude loops and functions, this trivially ensures that each register is assigned at most once per pomset.

We refactor the syntax:

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

Explicit Substitutions. Let $\mathbb{M}$ range over *extended expressions*, which may include memory locations. We introduce explicit substitutions over extended expressions, following the conventions of Ritter and de Paiva [1997]:

$$\varkappa \;::=\; x \mid r \qquad\qquad \sigma,\,\pi \;::=\; \langle\rangle \mid \langle\sigma,\,\mathbb{M}/\varkappa\rangle \mid \sigma\,;\sigma'$$

$\langle\rangle$ is the identity substitution. We write $\langle\langle\rangle,\,\mathbb{M}/\varkappa\rangle$ as $\langle\mathbb{M}/\varkappa\rangle$.

Application is written $\sigma * \phi$. We only apply substitutions to formulae—which do not bind locations or registers. The definition is homomorphic over the syntax of formulae. For the basis, $\langle\sigma,\,\mathbb{M}/\varkappa'\rangle * \varkappa$ is $\mathbb{M}$ if $\varkappa' = \varkappa$ and is $\sigma * \varkappa$ otherwise.

Sequencing is defined so that $\sigma\,;\langle\mathbb{M}/\varkappa\rangle = \langle\sigma,\,\sigma * \mathbb{M}/\varkappa\rangle$ and $(\sigma_1\,;\sigma_2) * \phi = \sigma_1 * (\sigma_2 * \phi)$.

We say that σ *subsumes* π if for every $\varkappa$, either $\pi * \varkappa = \sigma * \varkappa$ or $\pi * \varkappa = \varkappa$. For example, every substitution subsumes $\langle\rangle$.

We say that σ is *independent of* x if $\sigma * x = x$.

Let $\mathcal{S}$ be the set of all (explicit) substitutions.

Substitutions in the Model.

Change Definition 2.1, of *memory model pomset*, so that $\lambda : E \rightarrow (\Phi \times \mathcal{A} \times \mathcal{S})$, from which we derive the additional function $\mathcal{S} : E \rightarrow \mathcal{S}$.

We write triples in $(\Phi \times \mathcal{A} \times \mathcal{S})$ as $(\phi \mid a \mid \sigma)$, eliding ϕ when it is a tautology, eliding a when it is the termination action, and eliding σ when it is the identity substitution.

As a notational convenience, let $\mathcal{S}(P)$ return the substitution of the termination event in P, if one exists.

We ignore substitutions, except on read events and termination events. In the definition of sequential composition, read substitutions are used to calculate dependencies, as in item 5a of Definition 2.10. The terminal substitutions in $\mathcal{P}^1$ are composed with the substitutions in $\mathcal{P}^2$ when calculating $(\mathcal{P}^1\,;\mathcal{P}^2)$, and symmetrically.

Extend Definition 2.2, of relations between pomsets, to include subsumption: P' *subsumes* P if $E' = E$, $\leq' = \leq$, $\Phi' = \Phi$, and $\mathcal{A}' = \mathcal{A}$ and $\mathcal{S}'(e)$ subsumes $\mathcal{S}(e)$.

The semantics of programs is closed w.r.t. *reverse subsumption*: if $P \in [\![C]\!]$ and P is subsumed by P', then $P' \in [\![C]\!]$.

Subsumption is dual to implication: Stronger preconditions impose a greater burden on the preceding code; stronger substitutions can better mitigate this burden in following code. In examples, we only show executions that are implication and augmentation minimal; similarly, we only show executions that are subsumption-maximal.

Change the partial function Rd, from the data model, so that $\mathsf{Rd} : \mathcal{A} \rightarrow (\mathcal{R} \times \mathcal{X} \times \mathcal{V})$. When $\mathsf{Rd}(a) = (r, x, v)$, we say that *$a$ reads v from x into r*.

The semantics satisfies the following invariant: If P is subsumption-maximal and $\mathcal{S}(P)$ is defined, then for every e that reads x into r, there are σ_1 and σ_2 such that $\mathcal{S}(P) = (\sigma_1\,;\sigma_2)$ and $\mathcal{S}(e) = (\sigma_1\,;\langle x/r\rangle\,;\sigma_2)$.

Let $\mathcal{R}(P)$ be the substitution of values for registers that is derived from the reads of P as follows: $\mathcal{R}(P) * r = v$ if some event in P reads v into r, and $\mathcal{R}(P) * r = r$ otherwise.

Semantics of the Example Language.

As concrete syntax for read actions, we write $(\mathsf{R}^{\mu} r x v)$.

Change Definition 2.4, of *x-closed*, to require that every substitution is independent of x.

Change Definition 2.5, of $P' \in (\mathcal{P}^1 \parallel \mathcal{P}^2)$, to additionally require that for all $e \in E'$, either:

$$\begin{aligned} &\mathcal{S}'(e) \text{ is subsumed by both } \mathcal{S}^1(e) \text{ and } \mathcal{S}^2(e), \\ &\mathcal{S}'(e) \text{ is subsumed by } \mathcal{S}^1(e) \text{ and } e \notin E^2, \quad \text{or} \\ &\mathcal{S}'(e) \text{ is subsumed by } \mathcal{S}^2(e) \text{ and } e \notin E^1. \end{aligned}$$

Parallel composition, conditional and location binding are otherwise unchanged from §2.

Note that only shared register state is preserved by parallel composition. For example, $(r\,\mathtt{:=}\,1 \parallel r\,\mathtt{:=}\,1)\,\mathtt{;}\,x\,\mathtt{:=}\,r$ can write 1 to x, but $(r\,\mathtt{:=}\,1 \parallel \mathtt{skip})\,\mathtt{;}\,x\,\mathtt{:=}\,r$ cannot.

To simplify the base cases, we use a literal notation for pomsets and define $\mathsf{close}(P)$ to be the smallest set that includes P and is closed w.r.t. prefixing, implication, augmentation, and reverse subsumption.

$$\begin{aligned} [\![\mathtt{skip}]\!] &= \mathsf{close}\big(\{\boxed{\boxed{}}\}\big) \\ [\![r\,\mathtt{:=}\,M]\!] &= \mathsf{close}\big(\{\boxed{\boxed{\langle M/r\rangle}}\}\big) \\ [\![\mathsf{F}^{\nu}]\!] &= \mathsf{close}\big(\{\boxed{\mathsf{F}^{\nu}} \rightarrow \boxed{\boxed{}}\}\big) \\ [\![r\,\mathtt{:=}\,\mathtt{[}L\mathtt{]}^{\mu}]\!] &= \textstyle\bigcup_{\ell,v}\ \mathsf{close}\big(\{\boxed{L = \ell \mid \mathsf{R}^{\mu} r\,\mathtt{[}\ell\mathtt{]}\,v \mid \langle\mathtt{[}\ell\mathtt{]}/r\rangle} \rightarrow \boxed{\boxed{}}\}\big) \\ [\![\mathtt{[}L\mathtt{]}^{\mu}\,\mathtt{:=}\,M]\!] &= \ \|_{\ell,v}\ \mathsf{close}\big(\{\boxed{L = \ell \wedge M = v \mid \mathsf{W}^{\mu}\mathtt{[}\ell\mathtt{]}\,v} \rightarrow \boxed{\boxed{\langle M/\mathtt{[}\ell\mathtt{]}\rangle}}\}\big) \\ [\![C\mathtt{;}\,D]\!] &= [\![C]\!]\,;[\![D]\!] \end{aligned}$$

Whereas a write introduces the substitution $\langle M/\mathtt{[}\ell\mathtt{]}\rangle$ on the terminal event, a read introduces the substitution $\langle\mathtt{[}\ell\mathtt{]}/r\rangle$ on the read event itself.

The most significant challenge is defining sequential composition. Unfortunately, *disjunction* and *prefix weakening* (Definition 2.2) do not come easily.

[7] This restriction can be removed as follows. Separate registers into two categories: those used for direct assignment ($r\,\mathtt{:=}\,M$) and those used for reads ($r\,\mathtt{:=}\,\mathtt{[}L\mathtt{]}^{\mu}$). When constructing a pomset, define *program order* ($\xrightarrow{\text{po}}$) in the obvious way. Let d be *visible* if it reads into some r and there is no e that reads into r such that $d \xrightarrow{\text{po}} e$. Let $\mathcal{R}(P)$ be derived from visible reads (rather than all reads). In item 5a of Definition A.1, only consider events, d, that are visible reads.

Recall (∗∗) from §3: $\llbracket a[r] := 0;\ a[0] := !r \rrbracket$. In the semantics, an event from the first statement can coalesce with an event from the second. Thus when computing $\llbracket a[r] := 0 \rrbracket\ ;\ \llbracket a[0] := !r \rrbracket$, we must coalesce events with incompatible preconditions (r=0, r=1) that occur on different sides of the sequencing operator. This makes a direct definition difficult.

Instead of a direct definition, we first construct the sequential composition *without* coalescing events, then close the resulting set of pomsets to ensure the required properties.

There is also a challenge dealing with redundant write elimination: $\llbracket x := 1;\ x := 2 \rrbracket$ should contain a pomset that includes only ($\mathsf{W}x2$). We achieve this using the same strategy: closing after the construction.

Let c be an *unused write* in P when it is a relaxed write to some x such that (1) c fulfills no reads, (2) there is some $d > c$ that writes x, and (3) for every release $e > c$ there is some $e > d \geq c$ that writes x.

Finally, there is a challenge in calculating the preconditions for the events of $\mathcal{P}^2$ in $(\mathcal{P}^1\ ;\mathcal{P}^2)$ when terminal event of $\mathcal{P}^1$ is missing, and symmetrically. Again, we use the same strategy: We compute sequential composition using completed executions, then prefix close.

Definition A.1. Let $\mathrm{dpw}(\mathcal{P})$ be the least set that includes $\mathcal{P}$ and that is closed w.r.t. disjunction, prefixing, prefix weakening, and unused write removal.

Let $(\mathcal{P}^1\ ;\mathcal{P}^2)$ be the set $\mathrm{dpw}(\mathcal{P}')$ where $P' \in \mathcal{P}'$ when there are $P^1 \in \mathcal{P}^1$, and $P^2 \in \mathcal{P}^2$ such that the following hold.

Let d range over E^1. Let e range over E^2.

1. $E' = E^2 \uplus \{d \in E^1 \mid d$ is not a termination$\}$,
2. $\leq' \supseteq \leq^1 \cup \leq^2$,

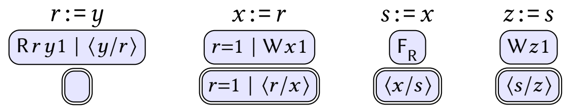
3a. $\mathcal{A}'(d) = \mathcal{A}^1(d)$,
3b. $\Phi'(d)$ implies $\Phi^1(d)$,
3c. $\mathcal{S}'(d)$ is subsumed by $\mathcal{S}^1(d)\ ;\mathcal{S}(P^2)$,
4a1. $\mathcal{A}'(e) = \mathcal{A}^2(e)$,
4a2. $\mathcal{S}'(e)$ is subsumed by $\mathcal{S}(P^1)\ ;\mathcal{S}^2(e)$,
4bc. $\Phi'(e)$ implies $\mathcal{R}(P^1) * \big(\mathcal{S}(P^1) * \Phi^2(e)\big)$,
5a. if d is a read and e is a write, then either $d \leq' e$ or $\Phi'(e)$ implies $\mathcal{R}(P^1) * \big(\mathcal{S}^1(d) * \Phi^2(e)\big)$,
5b-f. as before (see §2-3).

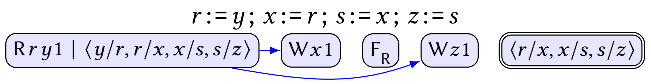
The item numbers are chosen to match those of the corresponding clauses in §2.

Items 4b and 4c collapse into a single item here. In 4bc, note that the domain of $\mathcal{S}(P^1)$ is disjoint from the domain of $\mathcal{R}(P^1)$, although registers in the domain of $\mathcal{S}(P^1)$ may appear in the expressions in the codomain of $\mathcal{R}(P^1)$.

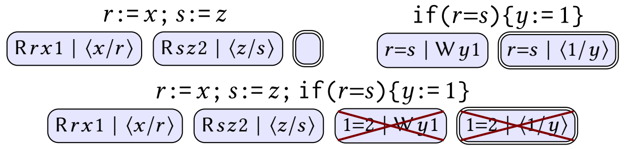
Item 5 is morally unchanged. In 5a, recall that $\mathcal{S}(P) = (\sigma_1\ ;\sigma_2)$ and $\mathcal{S}(e) = (\sigma_1\ ;\langle x/r\rangle\ ;\sigma_2)$. Note that $\mathcal{R}(P^1)\ ;\mathcal{S}^1(d)$ is insensitive to the value assigned to r by $\mathcal{R}(P^1)$.

As a simple example, consider the following:

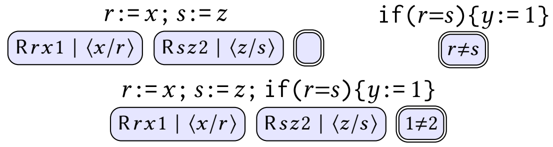




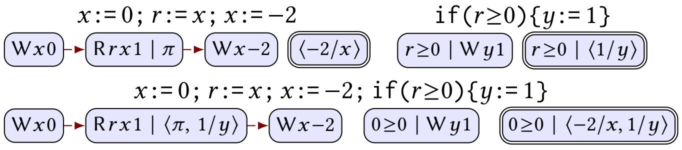
To see the need for parallel substitution of all register values via $\mathcal{R}(P^1)$ in 4bc, consider that the precondition of ($\mathsf{W}y1$) must be false after composing the following:



This pomset candidate does not satisfy the *compatibility* requirement of Definition 2.1. However, note that ($\mathtt{if}(r{=}s)\{y := 1\}$) is shorthand for ($\mathtt{if}(r{=}s)\{y := 1\}\,\mathtt{else}\,\{\mathtt{skip}\}$), and thus we do have a valid pomset for this composition, even with this choice of read values:



The see the need for substitutions on read actions, used in 5a, consider the following, where $\pi = \langle 0/x, x/r, -2/x\rangle$:



The semantics validates expected equations, such as $\llbracket C_1;\ C_2\rrbracket;\llbracket C_3\rrbracket = \llbracket C_1\rrbracket;\llbracket C_2;\ C_3\rrbracket$, $\llbracket \mathtt{if}(M)\{C;\ D_1\}\,\mathtt{else}\,\{C;\ D_2\}\rrbracket = \llbracket C\rrbracket\ ;\llbracket \mathtt{if}(M)\{D_1\}\,\mathtt{else}\,\{D_2\}\rrbracket$, and $\llbracket \mathtt{if}(M)\{C_1;\ D\}\,\mathtt{else}\,\{C_2;\ D\}\rrbracket = \llbracket \mathtt{if}(M)\{C_1\}\,\mathtt{else}\,\{C_2\}\rrbracket\ ;\llbracket D\rrbracket$.

The semantics is equivalent to that of the main text.

Let $\mathsf{kill} \rhd \mathcal{P}$ be the set $\mathcal{P}' \subseteq \mathcal{P}$ where $P' \in \mathcal{P}'$ if $\mathcal{S}(e') = \langle\rangle$, for every $e' \in E'$.

Proposition A.2. *Let* $\mathsf{old}\llbracket C\rrbracket$ *be the semantics of §2, adopting the read actions of this section—the read substitution being* $\langle\rangle$.

$$\mathsf{old}\llbracket C\rrbracket = \mathsf{kill} \rhd \llbracket C\rrbracket$$

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.

- $[\mathsf{R}]$, $[\mathsf{W}]$, $[\mathsf{Acq}]$, $[\mathsf{Rel}]$: 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.
- $\mathsf{fr} = \mathsf{co}; \mathsf{rf}^{-1}$: from-read, which relates reads to subsequent writes.

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

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

$$\begin{aligned}
\mathsf{eco} &= \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{co} && \text{(Extended coherence)}\\
\mathsf{obs} &= \mathsf{rfe} \cup \mathsf{fre} \cup \mathsf{coe} && \text{(Observed externally)}\\
\mathsf{dob} &= (\mathsf{addr} \cup \mathsf{data}); \mathsf{rfi}^? && \text{(Dependency order)}\\
&\quad \cup (\mathsf{ctrl} \cup \mathsf{data}); [\mathsf{W}]; \mathsf{coi}^? \\
&\quad \cup \mathsf{addr}; \mathsf{po}; [\mathsf{W}] \\
\mathsf{bob} &= [\mathsf{Acq}]; \mathsf{po} \cup \mathsf{po}; [\mathsf{Rel}]; \mathsf{coi}^? && \text{(Barrier order)}\\
\mathsf{ob} &= (\mathsf{obs} \cup \mathsf{dob} \cup \mathsf{bob})^+ && \text{(Acyclic order)}
\end{aligned}$$

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

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

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

From G we construct a candidate pomset P as follows:

- $E = \mathsf{E}$,
- $\mathcal{A}(e) = \mathrm{lab}(e)$, 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 $d \notin \mathrm{codom}(\mathsf{rfi}), e \notin \mathrm{codom}(\mathsf{rfi})$.

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' $(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})$ d $((\xrightarrow{\mathsf{eco}} \cap \xrightarrow{\mathsf{po}}) \setminus \xrightarrow{\mathsf{rfi}})$ e $(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})$ e'. Then $d' \xrightarrow{\mathsf{ob}} e'$.*

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

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

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

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

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

- If $d' \xrightarrow{\mathsf{bob}} d$, then:

  $$d' \ ([\mathsf{Acq}]; \mathsf{po} \cup \mathsf{po}; [\mathsf{Rel}]; \mathsf{coi}^?) \ d$$

  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, $d \xrightarrow{\mathsf{coi}} e$. Thus, result follows.
- If $d' \xrightarrow{\mathsf{dob}} d$, 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 $(\xrightarrow{\mathsf{eco}} \cap \xrightarrow{\mathsf{po}})$ e $(\xrightarrow{\mathsf{ob}} \cap \xrightarrow{\mathsf{po}})$ e' then $d \xrightarrow{\mathsf{ob}} e'$, under the assumptions that e' is not a release and e is not a release or a read and e, d are distinct.

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

- If $e \xrightarrow{\mathsf{bob}} e'$, then:

  $$e \ ([\mathsf{Acq}]; \mathsf{po} \cup \mathsf{po}; [\mathsf{Rel}]; \mathsf{coi}^?) \ e'$$

  Since e is not a read, $e(\mathsf{po}; [\mathsf{Rel}]; \mathsf{coi}^?)e'$. Result follows since $d \xrightarrow{\mathsf{po}} e$.
- If $e \xrightarrow{\mathsf{dob}} e'$, 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* $d \xrightarrow{\mathsf{ob}'} e$ *then* $\neg(e \xrightarrow{\mathsf{eco}} d)$.

Proof. Proof by contradiction. Let

$$e \xrightarrow{\mathsf{eco}} d \xrightarrow{\mathsf{ob}'} e$$

At least one of e, d is a write. So, if $e \not\xrightarrow{\mathsf{po}} d$, then $e \xrightarrow{\mathsf{ob}} d$, and we have a cycle in $\xrightarrow{\mathsf{ob}}$.

So, we conclude that $e \xrightarrow{\mathsf{po}} d$

Since $d \xrightarrow{\mathsf{ob}'} e$, there exists $d \not\xrightarrow{\mathsf{po}} c$, $d(\xrightarrow{\mathsf{eco}} \cap \xrightarrow{\mathsf{ob}'})c$, $c \xrightarrow{\mathsf{ob}'} e$.

We reason by cases.

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

□

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

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

Let $n \geq 1$ be such that:

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

where for all i, we have:

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

and

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

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 $d_0^i \xrightarrow{\mathsf{po}} d_1^i$, via lemma B.2, we deduce that $e_0^i \xrightarrow{\mathsf{ob}} d_1^i$, contradicting minimality of number of events in cycle.

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

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

which is a contradiction since it is a cycle in $\xrightarrow{\mathsf{eco}}$. □

C Modal pomsets

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

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

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

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

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

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

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

- We say that d *fulfills* e *on* x if d writes v to x, e reads v from x,
  - $d \triangleleft e$, and
  - if an event c writes to x then either $c \leq d$ or $e \leq c$.
- Augmentation has to include $\triangleleft$. i.e P' is an *augmentation* of P if $E' = E$, $\lambda' = \lambda$, $\trianglelefteq' \supseteq \trianglelefteq$, and $\leq' \supseteq \leq$.
- The definitions of substitution, restriction and the filtering operations stay the same, with $\trianglelefteq$ carried over unchanged. For example, substitution is defined as follows:
  Let $\mathcal{P}\sigma$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that: $E' = E$, $\trianglelefteq' = \trianglelefteq$, $\leq' = \leq$, and $\lambda'(e) = (\psi\sigma \mid a)$ when $\lambda(e) = (\psi \mid a)$.
- In composition,we require $\trianglelefteq' \supseteq \trianglelefteq^1 \cup \trianglelefteq^2$
- The changes to the definition 2.10 of prefixing are as follows. The key changes are that synchronization and dependency enforce $\triangleleft$ whereas coherence only enforces $\leq$.
  - $\trianglelefteq' \supseteq \trianglelefteq$.
  - 5b changes to: if d and e are actions in conflict, then $d \leq' e$,

- Item 5a, 5c, 5d, 5e 5c change to impose $\lhd$ order: eg. if a is an acquire or $\mathcal{A}(e)$ is a release then $c \lhd' e$.

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

C.1 Generators.

Modal pomsets provide a characterization of generators from section 6.

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

$(E, \leq, \lambda)$ is a generator for $[\![C]\!]$ if there exists $(E, \lhd, \leq, \lambda) \in [\![C]\!]_{\mathrm{M}}$ minimal w.r.t. augmentation and implication, and $\leq = (\unlhd \cup \mathsf{eco})^*$.

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

C.2 Closure properties

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

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

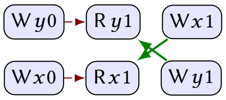
We say that $P' = P|_{E'}$ when $E' \subseteq E$, $\lambda' = \lambda|_{E'}$, and $\leq' = \leq|_{E'}$.

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

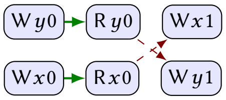
The semantics of read is "input"-enabled, since it permits the read of any visible value. Thus, any racy read in a program can be replaced by a read of a earlier value (w.r.t. eco), even while the races with existing independent writes are maintained. A canonical example to keep in mind for this lemma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

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

with both registers getting value 1 via the execution:



The lemma constructs the execution:



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

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

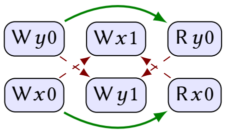
Proof. The form of C ensures that there is always a write to x that is related by $\xrightarrow{\mathsf{hb}}$ to any read. Thus, there is at least one other write than can satisfy the read recorded as e.

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

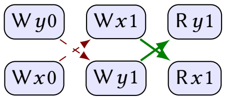
In the following lemma, invert the eco relationship between a read and a write. A canonical example to keep in mind for this lemma is the program:

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

with both registers getting value 0 via the execution:



The lemma constructs the execution:



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

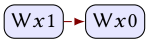
- *$e' \in Q \setminus P$ reads from x, with matching write d.*
- *The restriction of $\unlhd$ in P to $E_P \setminus (P$ after $e)$ agrees with the restriction of $\unlhd$ in Q to $E_Q \setminus (P$ after $e)$.*

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

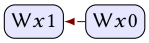
Any new event d' in Q after e' reading from x cannot have a matching write event $d'' \xrightarrow{\mathsf{eco}} d$ since that implies $d' \xrightarrow{\mathsf{eco}} d$ and a eco cycle $d \lhd e \lhd e' \xrightarrow{\mathsf{eco}} d$. Thus, the above lemma can be iterated if the new pomset is has any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

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

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



The lemma constructs the execution:



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

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

Then, there exists $Q \in [\![C]\!]_{\mathrm{M}}$ such that $E_P = E_Q$, $\trianglelefteq_P = \trianglelefteq_Q$, and $e \leq d$ in Q.

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

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

- $e \leq d$ in Q
- For all reads c matched to e, change from $d \leq c$ in P to $c \leq d$ in Q
- For all reads c matched to d, change from $c \leq e$ in P to $e \leq c$ in Q □

D Proof of DRF

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

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

We prove:

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

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

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

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

We prove by induction on $\mathrm{size}(C) - \mathrm{size}(Q)$ that given (P, Q) such that:

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

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

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

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

Otherwise, consider a maximal sequential prefix, extending Q, w.r.t. all of $\mathsf{po}, \mathsf{eco}, \lhd$. If it strictly contains Q, result follows from induction hypothesis.

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

If one of these events, say e is a write, we proceed as follows. Using hb-minimality of e, we deduce po minimality of e. Using the generator properties, we deduce that e is $\lhd$-minimal . Using lemma C.4, we build P_1 from P without changing Q to ensure that there are is no read $d \in P_1 \setminus Q$ such that $d \xrightarrow{\mathsf{eco}} e$. Using lemma C.5, we build P_2 from P_1 without changing Q to ensure that there are is no write $d \in P_2 \setminus Q$ such that $d \xrightarrow{\mathsf{eco}} e$. Thus, e is eco-minimal in $P_2 \setminus Q$. Result follows from induction hypothesis by considering (P_2, Q_1) where Q_1 is got from Q by adding e.

So, we can assume that all events in $P \setminus Q$, say $e_0, \ldots, e_n$ that are minimal w.r.t. hb are reads, and we have events $e'_0, e'_1, \ldots, e'_n, e_0$ such that:

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

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

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