The Leaky Semicolon

Compositional Semantic Dependencies for Relaxed-Memory Concurrency

ALAN JEFFREY, Roblox, USA
JAMES RIELY, DePaul University, USA
MARK BATTY, University of Kent, UK
SIMON COOKSEY, University of Kent, UK
ILYA KAYSIN, JetBrains Research, Russia and University of Cambridge, UK
ANTON PODKOPAEV, HSE University, Russia

Program logics and semantics tell a pleasant story about sequential composition: when executing $(S_1; S_2)$, we first execute S_1 then S_2. To improve performance, however, processors execute instructions out of order, and compilers reorder programs even more dramatically. By design, single-threaded systems cannot observe these reorderings; however, multiple-threaded systems can, making the story considerably less pleasant. A formal attempt to understand the resulting mess is known as a "relaxed memory model." Prior models either fail to address sequential composition directly, or overly restrict processors and compilers, or permit nonsense thin-air behaviors which are unobservable in practice.

To support sequential composition while targeting modern hardware, we enrich the standard event-based approach with *preconditions* and *families of predicate transformers*. When calculating the meaning of $(S_1; S_2)$, the predicate transformer applied to the precondition of an event e from S_2 is chosen based on the set of events in S_1 upon which e depends. We apply this approach to two existing memory models.

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

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

ACM Reference Format:
Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The Leaky Semicolon: Compositional Semantic Dependencies for Relaxed-Memory Concurrency. *Proc. ACM Program. Lang.* 6, POPL, Article 54 (January 2022), 65 pages. https://doi.org/10.1145/3498716

1 INTRODUCTION

Sequentiality is a *leaky abstraction* [Spolsky 2002]. For example, sequentiality tells us that when executing $(r_1 := x\,;\, y := r_2)$, the assignment $r_1 := x$ is executed before $y := r_2$. Thus, one might reasonably expect that the final value of r_1 is independent of the initial value of r_2. In most modern languages, however, this fails to hold when the program is run concurrently with $(s := y\,;\, x := s)$, which copies y to x.

In certain cases it is possible to ban concurrent access using separation [O'Hearn 2007], or to accept inefficient implementation in order to obtain sequential consistency (SC) [Marino et al. 2015].

Authors' addresses: Alan Jeffrey, Roblox, Chicago, USA, ajeffrey@roblox.com; James Riely, DePaul University, Chicago, USA, jriely@cs.depaul.edu; Mark Batty, University of Kent, Canterbury, UK, m.j.batty@kent.ac.uk; Simon Cooksey, University of Kent, Canterbury, UK, simon@graymalk.in; Ilya Kaysin, JetBrains Research, Russia and University of Cambridge, , UK, ik404@cam.ac.uk; Anton Podkopaev, HSE University, Saint Petersburg, Russia, apodkopaev@hse.ru.

2475-1421/2022/1-ART54
https://doi.org/10.1145/3498716

When these approaches are not available, however, the humble semicolon becomes shrouded in mystery, covered in the cloak of something known as a *memory model*. Every language has such a model: For each read operation, it determines the set of available values. Compilers and runtime systems are allowed to choose any value in the set. To allow efficient implementation, the set must not be too small. To allow invariant reasoning, the set must not be too large.

For optimized concurrent languages, it is surprising difficult to define a model that allows common compiler optimizations and hardware reorderings yet disallows nonsense behaviors that don't arise in practice. The latter are commonly known as "thin-air" behaviors [Batty et al. 2015]. There are only a handful of solutions, and all have deficiencies. These can be classified by their approach to dependency tracking (from strongest to weakest):

- Syntactic dependencies [Boehm and Demsky 2014; Kavanagh and Brookes 2018; Lahav et al. 2017; Vafeiadis and Narayan 2013]. These models require inefficient implementation of relaxed access. This is a non-starter for safe languages like Java and Javascript, and may be an unacceptable cost for low-level languages like C11.
- Semantic dependencies [Chakraborty and Vafeiadis 2019; Cho et al. 2021; Jagadeesan et al. 2010; Kang et al. 2017; Lee et al. 2020; Manson et al. 2005]. These models compute dependencies operationally using alternate worlds, making it impossible to understood a single execution in isolation; they also allow executions that violate temporal reasoning (see §9).
- No dependencies, as in C11 [Batty et al. 2015] and Javascript [Watt et al. 2019]. This allows thin-air executions.

These models are all non-compositional in the sense that in order to calculate the meaning of any thread, all threads must be known. Using the axiomatic approach of C11, for example, execution graphs are first constructed for each thread, using an operational semantics that allows a read to see any value. The combined graphs are then filtered using a set of acyclicity axioms that determine which reads are valid. These axioms use existentially defined global relations, such as memory order (mo), which must be a per-location total order on write actions.

Part of this non-compositionality is essential: In a concurrent system, the complete set of writes is known only at top-level. However, much of it is incidental. Two recent models have attempted to limit non-compositionality. Jagadeesan et al. [2020] defined Pomsets with Preconditions (PwP), which use preconditions and logic to calculate dependencies for a Java-like language. Paviotti et al. [2020] defined Modular Relaxed Dependencies (MRD), which use event structures to calculate a semantic dependency relation (sdep). PwP is defined using (acyclic) labelled partial orders, or *pomsets* [Gischer 1988]. MRD adds a causality axiom to C11, stating that $(\mathsf{sdep} \cup \mathsf{rf})$ must be acyclic. In both approaches, acyclicity enables inductive reasoning.

While PwP and MRD both treat *concurrency* compositionally, neither gives a compositional account of *sequentiality*. PwP uses prefixing, adding one event at a time on the left. MRD encodes sequential composition using continuation-passing. In both, adding an event requires perfect knowledge of the future. For example, suppose that you are writing system call code and you wish to know if you can reorder a couple of statements. Using PwP or MRD, you cannot tell whether this is possible without having the calling code! More formally, Jagadeesan et al. state the equivalence allowing reordering independent writes as follows:

$$[\![x := M;\ y := N;\ S]\!] = [\![y := N;\ x := M;\ S]\!] \ \text{ if } x \neq y$$

This requires a quantification over all continuations S. This is problematic, both from a theoretical point of view—the syntax of programs is now mentioned in the definition of the semantics—and in practice—tools cannot quantify over infinite sets. This problem is related to contextual equivalence, full abstraction [Milner 1977; Plotkin 1977] and the CIU theorem of Mason and Talcott [1992].

In this paper, we show that PwP can be extended with *families of predicate transformers* (PwT) to calculate sequential dependencies in a way that is *compositional* and *direct*: *compositional* in that the denotation of $(S_1\,;\,S_2)$ can be computed from the denotation of S_1 and the denotation of S_2, and *direct* in that these can be calculated independently. With this formulation, we can show:

$$[\![x := M\,;\,y := N]\!] = [\![y := N\,;\,x := M]\!] \text{ if } x \neq y$$

Then the equivalence holds in any context—this form of the equivalence enables reasoning about peephole optimizations. Said differently, unlike prior work, PwT allows the presence or absence of a dependency to be understood in isolation—this enables incremental and modular validation of assumptions about program dependencies in larger blocks of code.

Our main insight is that for language models, *sequentiality* is the hard part. *Concurrency* is easy! Or at least, it is no more difficult than it is for hardware. Compilers make the difference, since they typically do little optimization between threads. We motivate our approach to sequential dependencies in §2 and provide formal definitions in §3. In §8, we extend the model to include additional features, such as address calculation and RMWs. We discuss related and future work in §9–10.

We extend PwT to a full memory model in §4, based on PwP [Jagadeesan et al. 2020]. §5 summarizes the results for this model. In addition to powering such a bespoke model, the dependency relation calculated by PwT can also be used with off-the-shelf models. For example, in §6 we show that it can be used as an sdep relation for C11, adapting the approach of MRD [Paviotti et al. 2020]. §7 describes a tool for automatic evaluation of litmus tests in this model. C11 allows thin-air in order to avoid overhead in the implementation of relaxed reads. Safe languages like OCaml [Dolan et al. 2018] have typically made the opposite choice, accepting a performance penalty in order to avoid thin-air. Just as PwT can be used to strengthen C11, it could also be used to weaken these models, allowing optimal lowering for relaxed reads while banning thin-air.

PwT has been formalized in Coq. We have formally verified that the sequential composition satisfies the expected monoid laws (Lemma 3.5). In addition we have formally verified that $[\![\texttt{if}(\phi)\,\{S_1\,;\,S_3\}\,\texttt{else}\,\{S_2\,;\,S_3\}]\!] \supseteq [\![\texttt{if}(\phi)\{S_1\}\,\texttt{else}\,\{S_2\}\,;\,S_3]\!]$ (Lemma 3.6e).

Supplementary material for this paper is available at https://weakmemory.github.io/pwt.

2 OVERVIEW

This paper is about the interaction of two of the fundamental building blocks of computing: sequential composition and mutable state. One would like to think that these are well-worn topics, where every issue has been settled, but this is not the case.

2.1 Sequential Composition

Novice programmers are taught *sequential abstraction*: that the program $S_1\,;\,S_2$ executes S_1 before S_2. Since the late 1960s, we've been able to explain this using logic [Hoare 1969]. In Dijkstra's [1975] formulation, we think of programs as *predicate transformers*, where predicates describe the state of memory in the system. In the calculus of weakest preconditions, programs map postconditions to preconditions. We recall the definition of $wp_S(\psi)$ for loop-free code below (where r–s range over thread-local *registers* and M–N range over side-effect-free *expressions*).

$$wp_{r:=M}(\psi) = \psi[M/r] \qquad wp_{S_1;S_2}(\psi) = wp_{S_1}(wp_{S_2}(\psi)) \qquad wp_{\texttt{skip}}(\psi) = \psi$$
$$wp_{\texttt{if}(M)\{S_1\}\,\texttt{else}\,\{S_2\}}(\psi) = ((M{\neq}0) \Rightarrow wp_{S_1}(\psi)) \wedge ((M{=}0) \Rightarrow wp_{S_2}(\psi))$$

Without loops, the Hoare triple $\{\phi\}\ S\ \{\psi\}$ holds exactly when $\phi \Rightarrow wp_S(\psi)$. This is an elegant explanation of sequential computation in a sequential context. Note that the assignment rule is sound because a read from a thread-local register must be fulfilled by a preceding write in the

same thread. In a concurrent context, with shared variables (x–z), the obvious generalization of the assignment rule for reads, $wp_{r:=x}(\psi) = \psi[x/r]$, is unsound! In particular, a read from a shared memory location may be fulfilled by a write in another thread.

In this paper we answer the following question: what does sequential composition mean in a concurrent context? An acceptable answer must satisfy several desiderata:

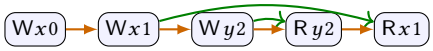
(1) it should not impose too much order, overconstraining the implementation,
(2) it should not impose too little order, allowing bogus executions, and
(3) it should be *compositional* and *direct*, as described in §1.

Memory models differ in how they navigate between desiderata 1 and 2. In one direction there are both more valid compiler optimizations and also more potentially dubious executions, in the other direction, less of both. To understand the tradeoffs, one must first understand the underlying hardware and compilers.

2.2 Memory Models

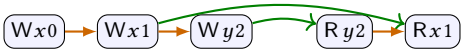
For single-threaded programs, memory can be thought of as you might expect: programs write to, and read from, memory references. This can be thought of as a total order over memory actions ($\rightarrow$), where each read has a matching *fulfilling* write ($\rightarrow$), for example:

$$x := 0\,;\; x := 1\,;\; y := 2\,;\; r := y\,;\; s := x$$



This model extends naturally to the case of shared-memory concurrency, leading to a *sequentially consistent* semantics [Lamport 1979], in which *program order* inside a thread implies a total *causal order* between read and write events, for example (where ; has higher precedence than ||):

$$x := 0\,;\; x := 1\,;\; y := 2 \;\|\; r := y\,;\; s := x$$

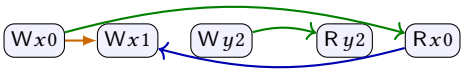


We can represent such an execution as a labelled partial order, or *pomset* [Gischer 1988; Pratt 1985]. A program may give rise to many executions, each reflecting a different interleaving of the threads.

Unfortunately, this model does not compile efficiently to commodity hardware, resulting in a 37–73% increase in CPU time on Arm8 [Liu et al. 2019] and, hence, in power consumption. Developers of software and compilers have therefore been faced with a difficult trade-off, between an elegant model of memory, and its impact on resource usage (such as size of data centers, electricity bills and carbon footprint). Unsurprisingly, many have chosen to prioritize efficiency over elegance.

This has led to *relaxed memory models*, in which the requirement of sequential consistency is weakened to only apply *per-location*. This allows executions that are inconsistent with program order, such as the following, which contains an *antidependency* ($\rightarrow$):

$$x := 0\,;\; x := 1\,;\; y := 2 \;\|\; r := y\,;\; s := x$$



In such models, the causal order between events is important, and includes control and data dependencies ($\rightarrow$) to avoid paradoxical "out of thin air" examples such as the following. (We routinely elide initializing writes when they are uninteresting.)

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

This candidate execution forms a cycle in causal order, so is disallowed, but this depends crucially on the control dependency from $(\mathsf{R}x1)$ to $(\mathsf{W}y1)$, and the data dependency from $(\mathsf{R}y1)$ to $(\mathsf{W}x1)$. If either is missing, then this execution is acyclic and hence allowed. For example dropping the control dependency results in the following execution, which should be allowed:

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

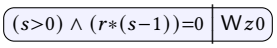
While syntactic dependency calculation suffices for hardware models, it is not preserved by common compiler optimizations. For example, consider the following program:

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

Because $y := 1$ occurs on both branches of the conditional, a compiler may lift it out. With the dependency removed, the compiler could reorder the read of x and write to y, allowing both reads to see 1. Attempting to generate this execution with syntactic dependencies, however, results in the following candidate execution, which has a cycle and therefore is disallowed:

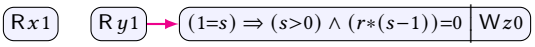
To address this, Jagadeesan et al. [2020] introduced *Pomsets with Preconditions* (PwP), where events are labeled with logical formulae. Nontrivial preconditions are introduced by store actions (modeling data dependencies) and conditionals (modeling control dependencies):

$$\texttt{if}(s{>}0)\{z := r{*}(s{-}1)\}$$



In this diagram, $(s{>}0)$ is a control dependency and $(r{*}(s{-}1)){=}0$ is a data dependency. Preconditions are updated as events are prepended (we assume the usual precedence for logical operators):

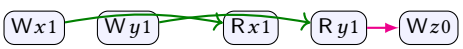
$$r := x\,;\; s := y\,;\; \texttt{if}(s{>}0)\{z := r{*}(s{-}1)\}$$



In this diagram there are two reads. As evidenced by the arrow, the read of y is ordered before the write, reflecting possible dependency; the read of x is not, reflecting independency. The dependent read of y allows the precondition of the write to weaken: now the old precondition need only be satisfied assuming the hypothesis $(1{=}s)$. The independent read of x allows no such weakening. Nonetheless, the precondition of the write is now a tautology, and so can be elided in the diagram.

We can complete the execution by adding the required writes:

$$x := 1\,;\; y := 1 \parallel r := x\,;\; s := y\,;\; \texttt{if}(s{>}0)\{z := r{*}(s{-}1)\}$$



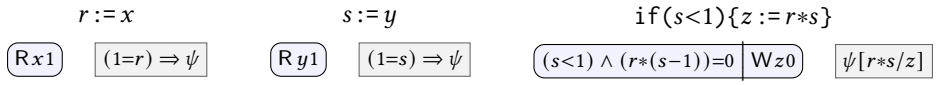
In order for a PwP to be *complete*, all preconditions must be tautologies and all reads must be fulfilled by matching writes. The first requirement captures the sequential semantics. The second requirement captures the concurrent semantics. These correspond to two views of memory for each thread: thread-local and global. In a *multicopy-atomic* (MCA) architecture, there is only one global view, shared by all processors, which is neatly captured by the order of the pomset (see §4).

An untaken conditional produces no events. PwP models this by including the empty pomset in the semantics of every program fragment. To then ensure that `skip` is not a refinement of $x := 1$, PwP include a *termination* action, ✓, which we have elided in the examples above.

2.3 Predicate Transformers For Relaxed Memory

PwP shows how the logical approach to sequential dependency calculation can be mixed into a relaxed memory model. Our contribution is to extend PwP with predicate transformers to arrive at a model of sequential composition. Predicate transformers are a good fit for logical models of dependency calculation, since both are concerned with preconditions.

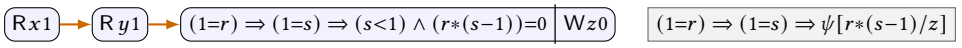
Our first attempt is to associate a predicate transformer with each pomset. We visualize this in diagrams by showing how ψ is transformed, for example:



The predicate transformer for a write $z := M$ matches Dijkstra: taking ψ to $\psi[M/z]$. For a read $r := x$, however, Dijkstra would transform ψ to $\psi[x/r]$, which is equivalent to $(x{=}r) \Rightarrow \psi$ under the assumption that registers are assigned at most once. Instead, we use $(1{=}r) \Rightarrow \psi$, reflecting the fact that 1 may come from a concurrent write. The obligation to find a matching write is moved from the sequential semantics of *substitution* and *implication* to the concurrent semantics of *fulfillment*.

For a sequentially consistent semantics, sequential composition is straightforward: we apply each predicate transformer to subsequent preconditions, composing the predicate transformers.

$$r := x\,;\, s := y\,;\, \texttt{if}(s{<}1)\{z := r{*}(s{-}1)\} \qquad (*)$$



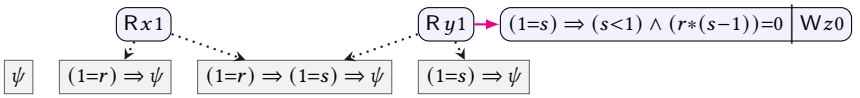
This works for the sequentially consistent case, but needs to be weakened for the relaxed case.

The key observation of this paper is that rather than working with one predicate transformer, we should work with a *family* of predicate transformers, indexed by sets of events. For example, for single-event pomsets, there are two predicate transformers, since there are two subsets of any one-element set. The *independent* transformer is indexed by the empty set, whereas the *dependent* transformer is indexed by the singleton. We visualize this by including more than one transformed predicate, with a dotted edge leading to the dependent one ($\cdots\!\!\rightarrow$). For example:

The model of sequential composition then picks which predicate transformer to apply to an event's precondition by picking the one indexed by all the events before it in causal order.

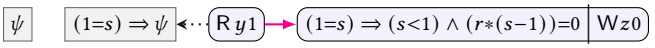
For example, we can recover the expected semantics for (*) by choosing the predicate transformer which is independent of $(\mathsf{R}x1)$ but dependent on $(\mathsf{R}y1)$, which is the transformer which maps ψ to $(1{=}s) \Rightarrow \psi$. (In subsequent diagrams, we only show predicate transformers for reads.)

$$r := x\,;\, s := y\,;\, \texttt{if}(s{<}1)\{z := r{*}(s{-}1)\}$$



In the diagram, the dotted lines indicate set inclusion into the index of the transformer-family. As a quick correctness check, we can see that sequential composition is associative in this case, since it does not matter whether we associate to the left—with the intermediate step as in the diagram above, eliding the write action—or to the right—with the intermediate step:

$$s := y\,;\, \texttt{if}(s{<}1)\{z := r{*}(s{-}1)\}$$



This is an instance of the general result that sequential composition forms a monoid (Lemma 3.5).

3 SEQUENTIAL SEMANTICS

After some preliminaries (§3.1–3.2), we define the model and establish some basic properties (§3.3 and Fig. 1). We then explain the model using examples (§3.4–3.9). We encourage readers to skim the definitions and then skip to §3.4, coming back as needed.

In this section, we concentrate on the sequential semantics, ignoring the requirement that concurrent reads be *fulfilled* by matching writes. We extend the model to a full concurrent semantics in §4 and §6 by defining a *reads-from* relation (rf) subject to various constraints.

3.1 Preliminaries

The syntax is built from

- a set of *values* $\mathcal{V}$, ranged over by v, w, ℓ, k,
- a set of *registers* $\mathcal{R}$, ranged over by r, s,
- a set of *expressions* $\mathcal{M}$, ranged over by M, N, L.

Memory references, aka *locations*, are tagged values, written $[\ell]$. Let $\mathcal{X}$ be the set of memory references, ranged over by x, y, z. We require that

- values and registers are disjoint,
- values are finite[1] and include at least the constants 0 and 1,
- expressions include at least registers and values,
- expressions do *not* include memory references: $M[N/x] = M$ (for all x).

We model the following language.

$$\mu, \nu \ ::= \ \mathsf{rlx} \ \mid \ \mathsf{rel} \ \mid \ \mathsf{acq} \ \mid \ \mathsf{sc}$$

$$S \ ::= \ r := M \ \mid \ r := [L]^{\mu} \ \mid \ [L]^{\mu} := M \ \mid \ \mathsf{F}^{\mu} \ \mid \ \mathtt{skip} \ \mid \ S_1 \,;\, S_2 \ \mid \ \mathtt{if}(M)\{S_1\}\,\mathtt{else}\,\{S_2\} \ \mid \ S_1 \Vdash S_2$$

Access modes, μ, are relaxed (rlx), release (rel), acquire (acq), and sequentially consistent (sc). Reads ($r := [L]^{\mu}$) support rlx, acq, sc. Writes ($[L]^{\mu} := r$) support rlx, rel, sc. Fences (F^{μ}) support rel, acq, sc. Register assignments ($r := M$) only affect thread-local state and therefore have no mode. In examples, the default mode for reads and writes is rlx—we systematically drop the annotation.

Commands, aka *statements*, S, include fences and memory accesses at a given mode, as well as the usual structural constructs. Following Ferreira et al. [1996], $\Vdash$ denotes parallel composition, preserving thread state on the right after a join. In examples without join, we use the symmetric $\|$ operator.

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

Throughout §1–7 we require that each register is assigned at most once in a program. In §8, we drop this restriction, requiring instead that there are registers that do not appear in programs.

The semantics is built from the following.

- a set of *events* $\mathcal{E}$, ranged over by e, d, c, and subsets ranged over by E, D, C,
- a set of *logical formulae* Φ, ranged over by ϕ, ψ, θ,
- a set of *actions* $\mathcal{A}$, ranged over by a, b,
- a family of *quiescence symbols* Q_x, indexed by location.

We require that

- formulae include tt, ff, Q_x, and the equalities $(M{=}N)$ and $(x{=}M)$,

[1] We require finiteness for the semantics of address calculation (§8.4), which quantifies over all values. Using types, one could limit the finiteness assumption to the subset of values used for address calculation.

- formulae are closed under $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and substitutions $[M/r]$, $[M/x]$, $[\phi/\mathsf{Q}_x]$,
- there is a relation $\vDash$ between formulae, capturing entailment,
- $\vDash$ has the expected semantics for $=$, $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and substitutions $[M/r]$, $[M/x]$, $[\phi/\mathsf{Q}_x]$,
- there is a subset of $\mathcal{A}$, distinguishing *read* actions,
- there are four binary relations over $\mathcal{A} \times \mathcal{A}$: *delays* and *matches* $\subseteq$ *blocks* $\subseteq$ *overlaps*.

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

We write $\phi \equiv \psi$ when $\phi \vDash \psi$ and $\psi \vDash \phi$. We say ϕ is a *tautology* if $\mathsf{tt} \vDash \phi$. We say ϕ is *unsatisfiable* if $\phi \vDash \mathsf{ff}$, and *satisfiable* otherwise.

3.2 Actions in This Paper

In this paper, each action is either a read, a write, or a fence:

$$a, b \ ::= \ \mathsf{R}^{\mu} x v \ \mid \ \mathsf{W}^{\mu} x v \ \mid \ \mathsf{F}^{\mu}$$

We use shorthand when referring to actions. In definitions, we drop elements of actions that are existentially quantified. In examples, we drop elements of actions, using defaults. Let $\sqsubseteq$ be the smallest order over access and fence modes such that $\mathsf{rlx} \sqsubseteq \mathsf{rel} \sqsubseteq \mathsf{sc}$ and $\mathsf{rlx} \sqsubseteq \mathsf{acq} \sqsubseteq \mathsf{sc}$. We write $(\mathsf{W}^{\sqsupseteq \mathsf{rel}})$ to stand for either $(\mathsf{W}^{\mathsf{rel}})$ or $(\mathsf{W}^{\mathsf{sc}})$, and similarly for the other actions and modes.

Definition 3.1. Actions (R) are *read* actions.
We say a *matches* b if $a = (\mathsf{W} x v)$ and $b = (\mathsf{R} x v)$.
We say a *blocks* b if $a = (\mathsf{W} x)$ and $b = (\mathsf{R} x)$, regardless of value.
We say a *overlaps* b if they access the same location, regardless of whether they read or write.
Let $\bowtie_{\mathsf{co}}$ capture write-write, read-write coherence: $\bowtie_{\mathsf{co}} = \{(\mathsf{W} x, \mathsf{W} x),\ (\mathsf{R} x, \mathsf{W} x),\ (\mathsf{W} x, \mathsf{R} x)\}$.
Let $\ltimes_{\mathsf{sync}}$ capture conflict due to synchronization:[2] $\ltimes_{\mathsf{sync}} = \{(a, \mathsf{W}^{\sqsupseteq \mathsf{rel}}),\ (a, \mathsf{F}^{\sqsupseteq \mathsf{rel}}),\ (\mathsf{R}, \mathsf{F}^{\sqsupseteq \mathsf{acq}}),\ (\mathsf{R}^{\sqsupseteq \mathsf{acq}}, b),\ (\mathsf{F}^{\sqsupseteq \mathsf{acq}}, b),\ (\mathsf{F}^{\sqsupseteq \mathsf{rel}}, \mathsf{W}),\ (\mathsf{W}^{\sqsupseteq \mathsf{rel}} x, \mathsf{W} x)\}$.
Let $\bowtie_{\mathsf{sc}}$ capture conflict due to sc access: $\bowtie_{\mathsf{sc}} = \{(\mathsf{W}^{\mathsf{sc}}, \mathsf{W}^{\mathsf{sc}}),\ (\mathsf{R}^{\mathsf{sc}}, \mathsf{W}^{\mathsf{sc}}),\ (\mathsf{W}^{\mathsf{sc}}, \mathsf{R}^{\mathsf{sc}}),\ (\mathsf{R}^{\mathsf{sc}}, \mathsf{R}^{\mathsf{sc}})\}$.
We say a *delays* b if $a \bowtie_{\mathsf{co}} b$ or $a \ltimes_{\mathsf{sync}} b$ or $a \bowtie_{\mathsf{sc}} b$.

3.3 PwT: Pomsets with Predicate Transformers

Predicate transformers are functions on formulae that preserve logical structure, providing a natural model of sequential composition. The definition follows Dijkstra [1975].[3]

Definition 3.2. A *predicate transformer* is a function $\tau : \Phi \longrightarrow \Phi$ such that
(x1) $\tau(\psi_1 \wedge \psi_2) \equiv \tau(\psi_1) \wedge \tau(\psi_2)$,
(x2) $\tau(\psi_1 \vee \psi_2) \equiv \tau(\psi_1) \vee \tau(\psi_2)$,
(x3) if $\phi \vDash \psi$, then $\tau(\phi) \vDash \tau(\psi)$.

We consistently use ψ as the parameter of predicate transformers. Note that substitutions ($\psi[M/r]$ and $\psi[M/x]$) and implications on the right ($\phi \Rightarrow \psi$) are predicate transformers.

As discussed in §1, predicate transformers suffice for sequentially consistent models, but not relaxed models, where dependency calculation is crucial. For dependency calculation, we use a *family* of predicate transformers, indexed by sets of events. When computing $[\![S_1 \,;\, S_2]\!]$, we will use τ^C as the predicate transformer for event $e \in [\![S_2]\!]$, where C includes all of the events in $[\![S_1]\!]$ that

[2] This formalization includes *release sequences* $(\mathsf{W}^{\sqsupseteq \mathsf{rel}} x, \mathsf{W} x)$. Symmetry would suggest that we include $(\mathsf{R} x, \mathsf{R}^{\sqsupseteq \mathsf{acq}} x)$, but this is not sound for Arm8.

[3] In addition to the three criteria of Def. 3.2, Dijkstra [1975] requires (x4′) $\tau(\mathsf{ff}) \equiv \mathsf{ff}$. The dependent transformer for read actions (R4a) fails x4′, since ff is not equivalent to $v{=}r \Rightarrow \mathsf{ff}$. We can define an analog of x4′ for our model using the register naming conventions of §8. Define θ_λ to capture the *register state* of a pomset: $\theta_\lambda = \bigwedge_{\{(e,v) \in (E \times \mathcal{V}) \mid \lambda(e) = (\mathsf{R}\, v)\}} (s_e{=}v)$ where $E = \mathsf{dom}(\lambda)$. We say that ϕ is λ-*inconsistent* if $\phi \wedge \theta_\lambda$ is unsatisfiable. We can then require (x4) if ψ is λ-inconsistent then $\tau(\psi)$ is λ-inconsistent. x4 is not needed for the results of this paper, therefore we have elided it from the main development.

precede e in causal order ($d <_1 e$ implies $d \in C$). Under the following definition, the larger C is, the better, at least in terms of satisfying preconditions. Adding more order can only increase the size of C. Thus more order means weaker preconditions.

Definition 3.3. A *family of predicate transformers* over E consists of a predicate transformer τ^D for each $D \subseteq \mathcal{E}$, such that if $C \cap E \subseteq D$ then $\tau^C(\psi) \vDash \tau^D(\psi)$.

In a family of predicate transformers, the transformer of a smaller set must entail the transformer of a larger set. Thus bigger sets are *better* and $\tau^E(\psi)$—the transformer of the biggest set—is the *best*. (The definition is insensitive to events outside E—it is for this reason that we have taken $D \subseteq \mathcal{E}$ rather than $D \subseteq E$.)

Definition 3.4. A *pomset with predicate transformers* (PwT) is a tuple $(E, \lambda, \kappa, \tau, \checkmark, <)$ where

(M1) $E \subseteq \mathcal{E}$ is a set of *events*,
(M2) $\lambda : E \to \mathcal{A}$ defines an *action* for each event,
(M3) $\kappa : \mathcal{E} \to \Phi$ defines a *precondition* for each event, such that
 (M3a) $e \notin E$ implies $\kappa(e) = \mathsf{ff}$,
(M4) $\tau : 2^{\mathcal{E}} \to \Phi \to \Phi$ is a *family of predicate transformers* over E,
(M5) $\checkmark : \Phi$ is a *termination condition*, such that
 (M5a) $\checkmark \vDash \tau^E(\mathsf{tt})$,
(M6) $< \subseteq E \times E$, is a strict partial order capturing *causality*.

A PwT is *complete* if

(C3) $\kappa(e)$ is a tautology (for every $e \in E$), (C5) $\checkmark$ is a tautology.

We refer to PwTs simply as pomsets. Let P range over pomsets, and $\mathcal{P}$ over sets of pomsets.

Throughout the rest of this section, we endeavor to explain Fig. 1, which gives the semantics of programs $[\![\cdot]\!]$. We use consistent sub- and super-scripts to refer to the components of a pomset. For example $<_1$ is the order of P_1, $<'$ is the order of P', and $<$ is the order of P. We also use consistent numbering. For example, item 3 always refers to κ and item 5 always refers to $\checkmark$. As usual, we write $d \leq e$ to mean $d < e$ or $d = e$.

The core of the model is a labeled partial order, including a set of events (M1), a labeling (M2), and an order (M6). On top of this basic structure, M3–M5 add a layer of logic. For each pomset, M5 provides a termination condition. For each event in a pomset, M3 provides a precondition. For each set of events in a pomset, M4 provides a predicate transformer. The partial order and the logic are tied together formally in the definition of κ'_2 in *SEQ* in Fig. 1, which calculates dependencies.

Before discussing the details, we note that the semantics satisfies the expected monoid laws, as well as some laws concerning the conditional. We have verified Lemma 3.5 and Lemma 3.6e in Coq[4]. Similar laws apply to parallel composition—for example $[\![S]\!] = [\![\mathtt{skip} \Vdash S]\!]$. Note, however, that $[\![S]\!] \neq [\![S \Vdash \mathtt{skip}]\!]$—this asymmetric operator throws away thread state from the left.

LEMMA 3.5. *(a)* $[\![S]\!] = [\![(S\,;\,\mathtt{skip})]\!] = [\![(\mathtt{skip}\,;\,S)]\!]$. *(b)* $[\![(S_1\,;\,S_2)\,;\,S_3]\!] = [\![S_1\,;\,(S_2\,;\,S_3)]\!]$.

The proof of (a) requires M5a for the termination condition in $(S\,;\,\mathtt{skip})$. The proof of (b) requires both conjunction closure (X1, for the termination condition) and disjunction closure (X2, for the predicate transformers themselves). The proof of (b) also requires that S6 enforce projection as well as inclusion (see the definition of *respects* in Fig. 1).

LEMMA 3.6. *(c)* $[\![\mathtt{if}(\phi)\{S_1\}\mathtt{else}\{S_2\}]\!] \supseteq [\![S_1]\!]$ *if* ϕ *is a tautology.*
(d) $[\![\mathtt{if}(\phi)\{S\}\mathtt{else}\{S\}]\!] \supseteq [\![S]\!]$.
(e) $[\![\mathtt{if}(\phi)\{S_1\,;\,S_3\}\mathtt{else}\{S_2\,;\,S_3\}]\!] \supseteq [\![\mathtt{if}(\phi)\{S_1\}\mathtt{else}\{S_2\}\,;\,S_3]\!]$.

[4]Specifically, we have proven these results for the semantics of Fig. 1 with the refinements of §3.7, §8.1, and §8.3

(f) $[\![\texttt{if}(\phi)\{S_1\,;\,S_2\}\,\texttt{else}\,\{S_1\,;\,S_3\}]\!] \supseteq [\![S_1\,;\,\texttt{if}(\phi)\{S_2\}\,\texttt{else}\,\{S_3\}]\!]$.
(g) $[\![\texttt{if}(\neg\phi)\{S_2\};\,\texttt{if}(\phi)\{S_1\}]\!] \subseteq [\![\texttt{if}(\phi)\{S_1\}\,\texttt{else}\,\{S_2\}]\!] \supseteq [\![\texttt{if}(\phi)\{S_1\};\,\texttt{if}(\neg\phi)\{S_2\}]\!]$.

In §8.3, we refine the semantics to validate the reverse inclusions for (d–f) using if-introduction. Although the semantics of Fig. 1 validates the reverse inclusions for (g), these do not hold for PwT-MCA (see §10).

The semantics is closed with respect to augmentation: P_2 is an *augment* of P_1 if all fields are equal except, perhaps, the order, where we require $<_2 \supseteq <_1$.

Lemma 3.7. *If $P_1 \in [\![S]\!]$ and P_2 augments P_1 then $P_2 \in [\![S]\!]$.*

Augment closure captures the intuition that it is always sound for a compiler to make more conservative assumptions about dependencies than the semantics.

Unless otherwise noted, all pomsets in examples are *complete* and *augment-minimal*.

3.4 Pomsets and Complete Pomsets: Termination

Ignoring the logic, the definitions of Fig. 1 are straightforward. Reads, writes and fences map to pomsets with at most one event—we allow the empty pomset so that these may appear in the untaken branch of a conditional. `skip` and register assignment map to the empty pomset. The structural rules combine pomsets: *PAR* performs disjoint union, inheriting labeling and order from the two sides. *SEQ* and *IF* both perform a union.

We say that $d \in E_1$ and $e \in E_2$ *coalesce* if $d = e$. As a trivial consequence of using union rather than disjoint union, s1 validates *mumbling* [Brookes 1996] by coalescing events. For example $[\![x := 1\,;\,x := 1]\!]$ includes the singleton pomset $\boxed{\mathsf{W}x1}$. From this it is easy to see that $[\![x := 1\,;\,x := 1]\!] \supseteq [\![x := 1]\!]$ is a valid refinement. It is equally obvious that $[\![x := 1]\!] \not\supseteq [\![x := 1\,;\,x := 1]\!]$ is not a valid refinement, since the latter includes a two-element pomset, but the former does not. (These are observationally distinguished by the context: $[-] \parallel r := x\,;\,x := 2\,;\,s := x\,;\,\texttt{if}(r{=}s)\{z := 1\}$.)

In complete pomsets, c3 requires that all preconditions must be tautologies. In order to allow complete pomsets with untaken conditionals, such as $\texttt{if}(\mathsf{ff})\{x := 1\}$, we allow the empty pomset in the semantics of all statements. Termination conditions ensure that the empty pomset is not used inappropriately. At top level, c5 requires that $\checkmark$ is a tautology. w5 and f5 ensure that writes and fences are included in complete pomsets, unless they are inside an untaken conditional. For example, termination conditions ensure that $[\![x := 1]\!] \not\supseteq [\![\texttt{skip}]\!]$, since $[\![\texttt{skip}]\!]$ includes the empty pomset with $\checkmark \equiv \mathsf{tt}$, but $[\![x := 1]\!]$ can only include the empty pomset with $\checkmark \equiv \mathrm{K}(\emptyset) = \mathsf{ff}$.

For reads, the definition of $\checkmark$ depends on the mode: relaxed reads may be elided in complete pomsets (r5a), but acquiring reads must be included (r5b). From this, it is easy to see that $[\![r := x]\!] \supseteq [\![\texttt{skip}]\!]$ is a valid refinement (where the default mode is rlx).

Note that $[\![x := 2]\!]$ can write any value v; the fact that v must be 2 is captured in the logic. In particular, w5 requires that $\checkmark \equiv 2{=}v$ for this program and c5 requires that $\checkmark$ be a tautology at top-level. In combination, these ensure that complete pomsets do not include bogus writes. Consider the following incomplete pomsets:

$x := 1$	$x := 2$	$\texttt{if}(M)\{x := 3\}$
$\boxed{\mathsf{W}x1}$	$\boxed{2{=}3 \mid \mathsf{W}x3}$	$\boxed{M{\neq}0 \mid \mathsf{W}x3}$

By merging, the semantics allows the following:

$$x := 1\,;\,x := 2\,;\,\texttt{if}(M)\{x := 3\}$$

$$\boxed{\mathsf{W}x1} \quad \boxed{M{\neq}0 \mid \mathsf{W}x3}$$

However, this pomset is incomplete—regardless of M—since $\checkmark \equiv 2{=}3 \equiv \mathsf{ff}$.

If $P \in \mathit{SKIP}$ then $E = \emptyset$ and $\tau^D(\psi) \equiv \psi$ and $\checkmark \equiv \mathsf{tt}$.

If $P \in \mathit{ASSIGN}(r, M)$ then $E = \emptyset$ and $\tau^D(\psi) \equiv \psi[M/r]$ and $\checkmark \equiv \mathsf{tt}$.

Suppose R_i is a relation in $E_i \times E_i$. We say R *respects* R_i if $R \supseteq R_i$ and $R \cap (E_i \times E_i) = R_i$.

If $P \in \mathit{PAR}(\mathcal{P}_1, \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

(P1) $E = (E_1 \uplus E_2)$,
(P2) $\lambda = (\lambda_1 \cup \lambda_2)$,
(P3) $\kappa(e) \equiv \kappa_1(e) \vee \kappa_2(e)$,
(P4) $\tau^D(\psi) \equiv \tau_2^D(\psi)$,
(P5) $\checkmark \equiv \checkmark_1 \wedge \checkmark_2$,
(P6) $<$ respects $<_1$ and $<_2$.

If $P \in \mathit{SEQ}(\mathcal{P}_1, \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$
let $\kappa_2'(e) = \tau_1^C(\kappa_2(e))$ where $C = \{c \mid c < e\}$

(S1) $E = (E_1 \cup E_2)$,
(S2) $\lambda = (\lambda_1 \cup \lambda_2)$,
(S3) $\kappa(e) \equiv \kappa_1(e) \vee \kappa_2'(e)$,
(S4) $\tau^D(\psi) \equiv \tau_1^D(\tau_2^D(\psi))$,
(S5) $\checkmark \equiv \checkmark_1 \wedge \tau_1^{E_1}(\checkmark_2)$,
(S6) $<$ respects $<_1$ and $<_2$.

If $P \in \mathit{IF}(\phi, \mathcal{P}_1, \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

(I1) $E = (E_1 \cup E_2)$,
(I2) $\lambda = (\lambda_1 \cup \lambda_2)$,
(I3) $\kappa(e) \equiv (\phi \wedge \kappa_1(e)) \vee (\neg\phi \wedge \kappa_2(e))$,
(I4) $\tau^D(\psi) \equiv (\phi \wedge \tau_1^D(\psi)) \vee (\neg\phi \wedge \tau_2^D(\psi))$,
(I5) $\checkmark \equiv (\phi \wedge \checkmark_1) \vee (\neg\phi \wedge \checkmark_2)$,
(I6) $<$ respects $<_1$ and $<_2$.

Let $\mathrm{K}(D) = \bigvee_{d \in D} \kappa(d)$. Note that $\mathrm{K}(\emptyset) = \mathsf{ff}$.

If $P \in \mathit{FENCE}(\mu)$ then

(F1) $|E| \leqslant 1$,
(F2) $\lambda(e) = \mathsf{F}^\mu$,
(F3) $\kappa(e) \equiv \mathsf{tt}$,
(F4) $\tau^D(\psi) \equiv \psi$,
(F5) $\checkmark \equiv \mathrm{K}(E)$.

If $P \in \mathit{WRITE}(x, M, \mu)$ then $(\exists v \in \mathcal{V})$

(W1) $|E| \leqslant 1$,
(W2) $\lambda(e) = \mathsf{W}^\mu x v$,
(W3) $\kappa(e) \equiv M{=}v$,
(W4) $\tau^D(\psi) \equiv \psi[M/x][\mathrm{K}(E)/\mathsf{Q}_x]$,
(W5) $\checkmark \equiv \mathrm{K}(E)$,

If $P \in \mathit{READ}(r, x, \mu)$ then $(\exists v \in \mathcal{V})$

(R1) $|E| \leqslant 1$,
(R2) $\lambda(e) = \mathsf{R}^\mu x v$,
(R3) $\kappa(e) \equiv \mathsf{Q}_x$,
(R4a) if $e \in E \cap D$ then $\tau^D(\psi) \equiv (\kappa(e) \Rightarrow v{=}r) \Rightarrow \psi$,
(R4b) if $e \in E \setminus D$ then $\tau^D(\psi) \equiv (\kappa(e) \Rightarrow (v{=}r \vee x{=}r)) \Rightarrow \psi$,
(R4c) if $E = \emptyset$ then $\tau^D(\psi) \equiv \psi$,
(R5a) if $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{tt}$,
(R5b) if $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \mathrm{K}(E)$.

$$[\![r := M]\!] = \mathit{ASSIGN}(r, M) \qquad [\![\mathsf{F}^\mu]\!] = \mathit{FENCE}(\mu) \qquad [\![S_1 \between S_2]\!] = \mathit{PAR}([\![S_1]\!], [\![S_2]\!])$$
$$[\![x^\mu := M]\!] = \mathit{WRITE}(x, M, \mu) \qquad [\![\mathtt{skip}]\!] = \mathit{SKIP} \qquad [\![S_1\,;\, S_2]\!] = \mathit{SEQ}([\![S_1]\!], [\![S_2]\!])$$
$$[\![r := x^\mu]\!] = \mathit{READ}(r, x, \mu) \qquad [\![\mathtt{if}(M)\{S_1\}\mathtt{else}\{S_2\}]\!] = \mathit{IF}(M{\neq}0, [\![S_1]\!], [\![S_2]\!])$$

Fig. 1. PwT Semantics

Ignoring predicate transformers, P5 and S5 both take $\checkmark \equiv \checkmark_1 \wedge \checkmark_2$. This is as expected: the program terminates if both subprograms terminate. In I5, $\checkmark \equiv (\phi \wedge \checkmark_1) \vee (\neg\phi \wedge \checkmark_2)$: the program terminates as long as the taken branch terminates. Thus $[\![\mathtt{if(tt)\{}x := 1\mathtt{\}\,else\,\{}y := 1\mathtt{\}}]\!]$ contains a complete pomset with exactly one event: $(\mathsf{W}x1)$. To construct this pomset, we take the singleton from the left and the empty set from the right. This is a general principle: for code that contributes no events at top-level, use the empty set.

3.5 Preconditions, Predicate Transformers, and Data Dependencies

In this section, we ignore the Q_x symbols that appear in the semantics of read and write, taking $\mathsf{Q}_x = \mathsf{tt}$, for all x. We also introduce the independent transformer for reads (R4b) without explaining why it is defined as it is. We take up both subjects in §3.8.

Preconditions are discharged during sequential composition by applying predicate transformers $-\tau_1-$from the left to preconditions$-\kappa_2(e)-$on the right. The specific rule is S3, which uses the transformed predicate $\kappa_2'(e) = \tau_1^C(\kappa_2(e))$, where $C = \{c \mid c < e\}$ is the set of events that precede e in causal order. We call C the *dependent set* for e. Then $E \setminus C$ is the *independent set*.

Before looking at the details, it is useful to have a high-level view of how nontrivial preconditions and predicate transformers are introduced.

Preconditions are introduced in:

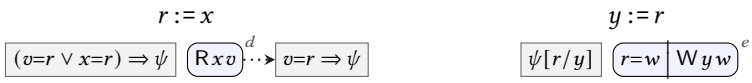
(W3) for data dependencies,
(I3) for control dependencies.

Predicate transformers are introduced in:

(R4a) for reads in the dependent set,
(R4b) for reads in the independent set,
(W4) for writes.

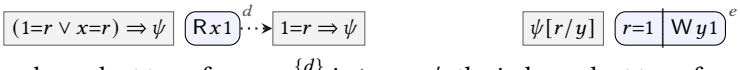
The rules track dependencies. We discuss data dependencies (W3) here and control dependencies (I3) in §3.6. We enrich the semantics to handle address dependencies in §8.4.

A simple example of a data dependency is a pomset $P \in [\![r := x\,;\, y := r]\!]$. If P is complete, it must have two events. Then *SEQ* (Fig. 1) requires $P_1 \in [\![r := x]\!]$ and $P_2 \in [\![y := r]\!]$ of the following form. (We only show the independent transformer for writes—ignoring Q_x, the dependent and independent transformers for writes are the same.)

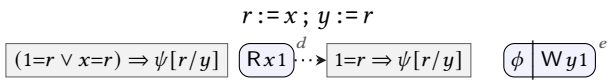


(†)

First we consider the case that $v = w$. For example, if $v = w = 1$, we have:



For the read, the dependent transformer $\tau_1^{\{d\}}$ is $1{=}r \Rightarrow \psi$; the independent transformer $\tau_1^{\emptyset}$ is $(1{=}r \vee x{=}r) \Rightarrow \psi$. These are determined by R4a and R4b, respectively. For the write, both $\tau_2^{\{e\}}$ and $\tau_2^{\emptyset}$ are $\psi[r/y]$, as are determined by W4. Combining these into a single pomset, we have:

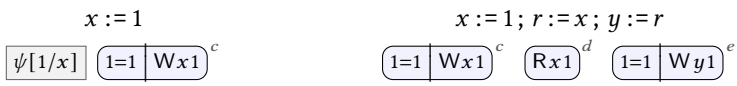


Looking at the precondition ϕ of the write, recall that in order for e to participate in a top-level pomset, the precondition ϕ must be a tautology at top-level. There are two possibilities.

- If $d < e$ then we apply the dependent transformer and $\phi \equiv (1{=}r \Rightarrow r{=}1)$, a tautology.
- If $d \not< e$ then we apply the independent transformer and $\phi \equiv ((1{=}r \vee x{=}r) \Rightarrow r{=}1)$. Under the assumption that r is bound (see footnote 3), this is logically equivalent to $(x{=}1)$.

Eliding transformers and tautological preconditions, the two outcomes are:

The independent case on the right can only participate in a top-level pomset if the precondition $(x{=}1)$ is discharged. To do so, we can prepend a program that writes 1 to x:

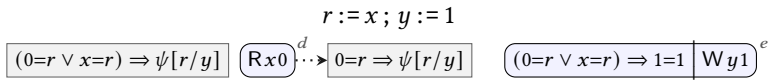


Here we apply the transformer from the left $(\psi[1/x])$ to $(x{=}1)$, resulting in the tautology $(1{=}1)$.

Now suppose that $v \neq w$ in (†). Again there are two possibilities. Taking v=0 and w=1:

Assuming that r is bound, both preconditions on e are unsatisfiable.

If a write is independent of a read, then clearly no order is imposed between them. For example, the precondition of e is a tautology in:



Note that both R4a and R4b degenerate to the identity transformer when $\kappa(e) = \mathtt{ff}$. This is the same as the transformer for the empty pomset (R4c).

Also note that $[\![S_1 \Vdash S_2]\!]$ is asymmetric, taking the predicate transformer for S_2 in P4.

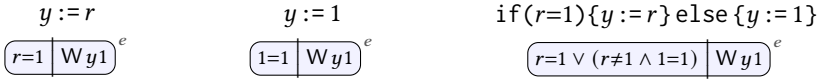
3.6 Control Dependencies

In $IF(\phi, \mathcal{P}_1, \mathcal{P}_2)$, the predicate transformer (I4) is $(\phi \wedge \tau_1^D(\psi)) \vee (\neg\phi \wedge \tau_2^D(\psi))$, which is the disjunctive equivalent of Dijkstra's conjunctive formulation: $(\phi \Rightarrow \tau_1^D(\psi)) \wedge (\neg\phi \Rightarrow \tau_2^D(\psi))$.

Control dependencies are introduced by the conditional. For coalescing events in $E_1 \cap E_2$, I3 requires $(\phi \wedge \kappa_1(e)) \vee (\neg\phi \wedge \kappa_2(e))$. For other events from E_i, it requires $\phi \wedge \kappa_i(e)$, using M3a. Control dependencies are eliminated in the same way as data dependencies. Consider:

As for (†), there are two possibilities:

When events coalesce, I3 ensures that control dependencies are calculated semantically, rather than syntactically. For example, consider $P \in [\![\mathtt{if}(r{=}1)\{y := r\}\,\mathtt{else}\,\{y := 1\}]\!]$, which is built from $P_1 \in [\![y := r]\!]$ and $P_2 \in [\![y := 1]\!]$. For example, consider:



Here, the precondition in the combined pomset (on the right) is a tautology, independent of r.

The semantics allows common code to be lifted out of a conditional, validating the transformation $[\![\mathtt{if}(M)\{S\}\,\mathtt{else}\,\{S\}]\!] \supseteq [\![S]\!]$. The semantics also validates dead code elimination: if $M{\neq}0$ is a tautology then $[\![\mathtt{if}(M)\{S_1\}\,\mathtt{else}\,\{S_2\}]\!] \supseteq [\![S_1]\!]$. Here, we take the empty pomset as the denotation of S_2. Since $M{=}0$ is unsatisfiable, I5 ignores the termination condition of S_2. It is worth noting that the reverse inclusion, dead-code-introduction, holds for *complete* pomsets, but not in general.

3.7 A Refinement: No Dependencies into Reads

To avoid stalling the CPU pipeline unnecessarily, hardware does not enforce control dependencies between reads. To support if-introduction (§8.3), software models must not distinguish control dependencies from other dependencies. Thus, we are forced to drop all dependencies into reads. To achieve this, we modify the definition of κ_2' in Fig. 1.

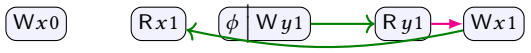
$$\kappa_2'(e) = \begin{cases} \tau_1^{E_1}(\kappa_2(e)) & \text{if } \lambda(e) \text{ is a read} \\ \tau_1^{C}(\kappa_2(e)) & \text{otherwise, where } C = \{c \mid c < e\} \end{cases}$$

Thus reads always use the "best" transformer, $\tau_1^{E_1}$. In order for non-reads to get a good transformer, they need to add order. Throughout the remainder of the paper, we use this definition.

3.8 Local State

Several of the JMM Causality Test Cases [Pugh 2004] center on compiler optimizations that result from limiting the range of variables. Because the compiler is allowed to collude with the scheduler when estimating the range, we refer to this as *local invariant reasoning*. The basic idea is that a write to y is independent of a read of x that precedes it, as long as the local state of x prior to the read justifies the write. For example, consider TC1:[5]

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



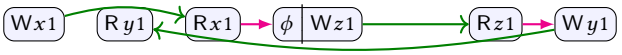
Using local invariant reasoning, a compiler could determine that x is always either 0 or 1, and therefore that the write to y does not depend on the read of x, allowing these to be reordered, resulting in the execution shown above. This is captured by our semantics as follows. Using R4b and W4, the precondition ϕ is $((1{=}r \vee x{=}r) \Rightarrow r \geqslant 0)[0/x]$ which is $((1{=}r \vee 0{=}r) \Rightarrow r \geqslant 0)$ which is indeed a tautology, justifying the independency. When used to form complete pomsets, R4b requires that subsequent preconditions be tautological under the assumption that the value of the read is used $(1{=}r)$ and under the assumption that the local value of x is used instead $(x{=}r)$.

This requires that we put locations into logical formulae, in addition to registers. While logical formulae involving registers are discharged by predicate transformers from *ASSIGN* or *READ* (Fig. 1), logical formulae involving locations are discharged by predicate transformers from *WRITE*. In other words, registers track the value of reads, whereas locations track the value of the most recent local write. This provides a local view of memory, distinct from the global view manifest in the labels on events. See [Jagadeesan et al. 2020] for further discussion.

A related concern arises when eliding changes to local state from the untaken branch of a conditional, creating *indirect dependencies*. Consider the following example [Paviotti et al. 2020, §6.3]:

$$\begin{array}{r} x := 1;\ r := y;\ \texttt{if}(r{=}0)\{x := 0;\ s := x;\ \texttt{if}(s)\{z := 1\}\} \\ \texttt{else}\,\{s := x;\ \texttt{if}(s)\{z := 1\}\} \end{array} \quad \Big\| \quad \texttt{if}(z)\{y := 1\}$$

In SC executions, the left thread always takes the then-branch of the conditional, reading 0 for x and therefore not writing z. As a result the second thread does not write y, and the program is data-race-free under SC. To satisfy the DRF-SC theorem, no other executions should be possible. Complete executions of the left thread that take the then-branch must include $(\mathsf{W}x0)$, whereas those that take the else-branch must *not* include $(\mathsf{W}x0)$. A problem arises if events from the subsequent code of the left thread—common to the two branches—coalesce, thus removing an essential control dependency. Consider the following candidate execution:



(††)

Note that the write to z depends on the read of x, but not the read of y. Ignoring Q_x, as we have done up to now, the precondition ϕ is:

$$\begin{array}{rl} \phi \equiv (1{=}r \vee y{=}r) \Rightarrow & (r{=}0 \wedge (1{=}s \Rightarrow s{\neq}0)) \\ & \wedge\ (r{\neq}0 \wedge (1{=}s \Rightarrow s{\neq}0)) \end{array}$$

Since $(1{=}s)$ implies $(s{\neq}0)$, the precondition is a tautology and (††) is allowed, violating DRF-SC.

[5] TC6 and TC8–9 are similar. TC2 and TC17–18 require both local invariant reasoning and resolving the nondeterminism of reads using redundant read elimination—see §8.1.

Without Q_x, the semantics enforces $(\mathsf{W}z1)$'s direct dependency on $(\mathsf{R}x1)$, but not its *indirect* dependency on $(\mathsf{R}y1)$. By eliding $(\mathsf{W}x0)$, we have forgotten the local state of x in the untaken branch of the execution. Nonetheless, we are using the subsequent—*stale*—read of x, by merging it with the read from the taken branch. This *half-stale* merged read is then used to justify $(\mathsf{W}z1)$.

In Fig. 1, R4 corrects this by introducing quiescence symbols into predicate transformers. Quiescence symbols capture the intuition that—in the untaken branch of a conditional—the value of a read from x can only be used if the most recent local write to x is included in the execution. Quiescence symbols are eliminated from formulae by the closest preceding write (w4). With quiescence, the precondition of (††) becomes the following:

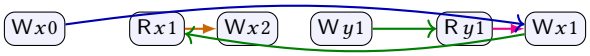
$$\begin{aligned}\phi' \equiv (\mathsf{Q}_y \Rightarrow 1{=}r \vee y{=}r) \Rightarrow\ & (r{=}0 \wedge ((\mathsf{Q}_x[\mathsf{ff}/\mathsf{Q}_x] \Rightarrow 1{=}s) \Rightarrow s{\neq}0))\\ & \wedge (r{\neq}0 \wedge ((\mathsf{Q}_x[1{=}1/\mathsf{Q}_x] \Rightarrow 1{=}s) \Rightarrow s{\neq}0))\end{aligned}$$

Adding initializing writes, Q_y becomes tt at top-level. Regardless, ϕ' is non-tautological: in the top conjunct, we have lost the ability to use $1{=}s$ to prove $s{\neq}0$. Intuitively, Q_x is true when the local state of x is up to date, and false when it is stale. In order to read x, Q_x requires that the most recent prior write to x must be in the pomset.

We also include quiescence symbols directly in preconditions of reads (R3). This guarantees initialization in complete pomsets: every $(\mathsf{R}x)$ must have a sequentially preceding $(\mathsf{W}x)$ in order to eliminate the precondition Q_x.

We end this subsection by noting that value range analysis of MRD [Paviotti et al. 2020] is overly conservative. Consider the following execution:

$$x := 0;\ (r := x;\ \mathsf{if}(r \leqslant 1)\{x := 2;\ y := 1\} \parallel x := y)$$



PwT correctly allows this execution; MRD forbids it by requiring $(\mathsf{R}x1) \rightarrow (\mathsf{W}y1)$. The co-product mechanism in MRD seeks an isomorphic justification under the $(\mathsf{R}x2)$ branch of the read in the event structure, and—failing to find such a justification—leaves the dependency in place.

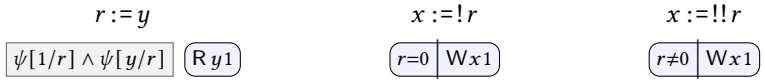
3.9 The Burdens of Associativity

Many of the design choices in PwT are motivated by Lemma 3.5—in particular, the need for sequential composition to be associative. In this subsection, we give three examples.

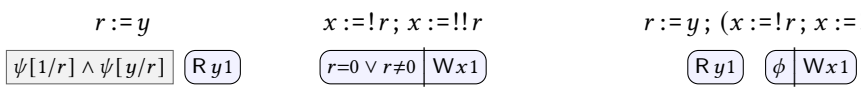
First, the predicate transformers we have chosen for R4a and R4b are different from the ones used traditionally, which are written using substitution. Attempting to write R4a and R4b in this style we would have (as in [Jagadeesan et al. 2020]):

(R4a′) if $e \in E \cap D$ then $\tau^D(\psi) \equiv \psi[v/r]$,
(R4b′) if $e \in E \setminus D$ then $\tau^D(\psi) \equiv \psi[v/r] \wedge \psi[x/r]$.

R4b′ does not distribute through disjunction (x2), and therefore is not a predicate transformer. This is not merely a theoretical inconvenience: adopting R4b′ would also break associativity. Consider the following example, where "!" represents logical negation:

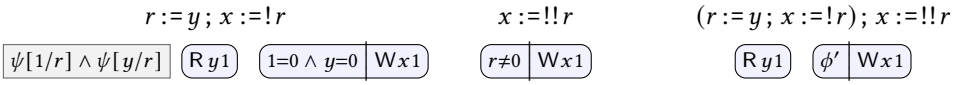


Associating to the right, we coalesce the writes then prepend the read:



The precondition ϕ is $(1{=}0 \vee y{=}0) \wedge (1{\neq}0 \vee y{\neq}0)$, which is a tautology.

Associating to the left, instead, we prepend the read then coalesce the writes:



The precondition ϕ' is $(1{=}0 \wedge y{=}0) \vee (1{\neq}0 \wedge y{\neq}0)$, which is not a tautology.

Our solution is to Skolemize, replacing substitution by implication, with uniquely chosen registers. Using Skolemization, Fig. 1 computes $\phi' \equiv ((1{=}r \vee y{=}r) \Rightarrow r{=}0) \vee ((1{=}r \vee y{=}r) \Rightarrow r{\neq}0)$, which is equivalent to $\phi \equiv (1{=}r \vee y{=}r) \Rightarrow (r{=}0 \vee r{\neq}0)$. Both are tautologies.

Second, Jagadeesan et al. impose *consistency*, which requires that for every pomset P, $\bigwedge_e \kappa(e)$ is satisfiable. Associativity requires that we allow inconsistent preconditions. To see this, note that

$$\big(\texttt{if}(M)\{x:=1\};\ \texttt{if}(!M)\{x:=1\}\big)\ ;\ \big(\texttt{if}(M)\{y:=1\};\ \texttt{if}(!M)\{y:=1\}\big)$$

has a complete pomset that writes x and y, regardless of M. In order to match this in

$$\texttt{if}(M)\{x:=1\}\ ;\ \big(\texttt{if}(!M)\{x:=1\};\ \texttt{if}(M)\{y:=1\}\big)\ ;\ \texttt{if}(!M)\{y:=1\},$$

the middle pomset must include the inconsistent actions $(M{=}0 \mid \mathsf{W}x1)$ and $(M{\neq}0 \mid \mathsf{W}y1)$.

Finally, we drop Jagadeesan et al.'s *causal strengthening* for the same reason. Consider:

$$\texttt{if}(M)\{r:=x\};\ y:=r;\ \texttt{if}(!M)\{s:=x\}$$

Associating to the right, this program has a complete pomset containing $(\mathsf{W}y1)$. Associating to the left, with causal strengthening, it does not.

4 PwT-MCA: POMSETS WITH PREDICATE TRANSFORMERS FOR MCA

In this section, we develop a model of concurrent computation by adding *reads-from* to Fig. 1. To model coherence and synchronization, we add *delay* to the rule for sequential composition. For MCA architectures, it is sufficient to encode delay in the pomset order. The resulting model, PwT-MCA$_1$, supports optimal lowering for relaxed access on Arm8, but requires extra synchronization for acquiring reads. (*Lowering* is the translation of language-level operators to machine instructions. A lowering is *optimal* if it provides the most efficient execution possible.)

A variant, PwT-MCA$_2$, supports optimal lowering for all access modes on Arm8. To achieve this, PwT-MCA$_2$ drops the global requirement that *reads-from* implies pomset order (M7c). The models are the same, except for *internal reads*, where a thread reads its own write. We show an example at the beginning of §4.2. The lowering proofs can be found in the supplementary material. The proofs use recent alternative characterizations of Arm8 [Alglave et al. 2021].

4.1 PwT-MCA1

We define PwT-MCA$_1$ by extending Def. 3.4 and Fig. 1. The definition uses several relations over actions—matches, blocks and delays—as well a distinguished set of read actions; see §3.2.

Definition 4.1. The definition of PwT-MCA$_1$ extends that of PwT with a relation rf such that

(M7) $\mathsf{rf} \subseteq E \times E$ is an injective relation capturing *reads-from*, such that
(M7a) if $d \xrightarrow{\mathsf{rf}} e$ then $\lambda(d)$ matches $\lambda(e)$,
(M7b) if $d \xrightarrow{\mathsf{rf}} e$ and $\lambda(c)$ blocks $\lambda(e)$ then either $c \leq d$ or $e \leq c$,
(M7c) if $d \xrightarrow{\mathsf{rf}} e$ then $d < e$.

The definition of completeness extends Def. 3.4 as follows:

(C7) if $\lambda(e)$ is a read then there is some $d \xrightarrow{\mathsf{rf}} e$.

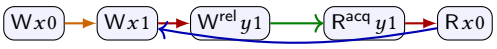
The semantic function extends Fig. 1 as follows:

(S6a) if $\lambda_1(d)$ delays $\lambda_2(e)$ then $d \leq e$, (P7) (S7) (I7) rf respects rf_1 and rf_2.

In complete pomsets, reads-from (rf) must pair every read with a matching write (C7). The requirements M7a, M7b, and M7c guarantee that reads are *fulfilled*, as in [Jagadeesan et al. 2020, §2.7]. Parallel composition, sequential composition, and the conditional respect reads-from (P7, S7, I7).

From Def. 3.1, recall that a *delays* b if $a \bowtie_{\mathsf{co}} b$ or $a \ltimes_{\mathsf{sync}} b$ or $a \bowtie_{\mathsf{sc}} b$. S6a guarantees that sequential order is enforced between conflicting accesses of the same location ($\bowtie_{\mathsf{co}}$), into a release and out of an acquire ($\ltimes_{\mathsf{sync}}$), and between SC accesses ($\bowtie_{\mathsf{sc}}$). Combined with the fulfillment requirements (M7a, M7b, M7c), these ensure coherence, publication, subscription and other idioms. For example, consider the following:[6]

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



The execution is disallowed due to the cycle. All of the order shown is required at top-level: The intra-thread order comes from S6a: $(\mathsf{W}x0) \to (\mathsf{W}x1)$ is required by $\bowtie_{\mathsf{co}}$. $(\mathsf{W}x1) \to (\mathsf{W}^{\mathsf{rel}}y1)$ and $(\mathsf{R}^{\mathsf{acq}}y1) \to (\mathsf{R}x0)$ are required by $\ltimes_{\mathsf{sync}}$. The cross-thread order is required by fulfillment: C7 requires that all top-level reads are in the image of $\xrightarrow{\mathsf{rf}}$. M7a ensures that $(\mathsf{W}^{\mathsf{rel}}y1) \xrightarrow{\mathsf{rf}} (\mathsf{R}^{\mathsf{acq}}y1)$, and M7c subsequently ensures that $(\mathsf{W}^{\mathsf{rel}}y1) < (\mathsf{R}^{\mathsf{acq}}y1)$. The *antidependency* $(\mathsf{R}x0) \to (\mathsf{W}x1)$ is required by M7b. (Alternatively, we could have $(\mathsf{W}x1) \to (\mathsf{W}x0)$, again resulting in a cycle.)

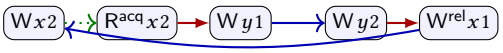
The semantics gives the expected results for store buffering and load buffering, as well as litmus tests involving fences and SC access. The model of coherence is weaker than C11, in order to support common subexpression elimination, and stronger than Java, in order to support local reasoning about data races. For further examples, see [Jagadeesan et al. 2020, §3.1].

Lemmas 3.5 and 3.6 hold for PwT-MCA$_1$. We discuss 3.6g further in §10.

4.2 PwT-MCA2

Lowering PwT-MCA$_1$ to Arm8 requires a full fence before every acquiring read.[7] To see why, consider the following attempted execution, where the final values of both x and y are 2.

$$x := 2;\ r := x^{\mathsf{acq}};\ y := r-1 \parallel y := 2;\ x^{\mathsf{rel}} := 1 \qquad \text{(INTERNAL-ACQ)}$$



The execution is allowed by Arm8, but disallowed by PwT-MCA$_1$, due to the cycle.

Arm8 allows the execution because the read of x is internal to the thread. This aspect of Arm8 semantics is difficult to model locally. To capture this, we found it necessary to drop M7c and relax S6a, adding local constraints on rf to *PAR*, *SEQ* and *IF*. (For parallelism, we explicitly specify the domain of d and e in S6a′.)

Definition 4.2. The definition of PwT-MCA$_2$ is derived from that of PwT-MCA$_1$ by removing M7c and S6a and adding the following:

(P6a) if $d \in E_1$, $e \in E_2$ and $d \xrightarrow{\mathsf{rf}} e$ then $d < e$,
(P6b) if $d \in E_1$, $e \in E_2$ and $e \xrightarrow{\mathsf{rf}} d$ then $e < d$,
(S6a′) if $d \in E_1$, $e \in E_2$ and $\lambda_1(d)$ delays $\lambda_2(e)$ then either $d \xrightarrow{\mathsf{rf}} e$ or $d \le e$,

[6] We use different colors for arrows representing order:
- $d \to e$ arises from $\bowtie_{\mathsf{co}}$ (S6a),
- $d \to e$ arises from $\ltimes_{\mathsf{sync}}$ or $\bowtie_{\mathsf{sc}}$ (S6a),
- $d \to e$ arises from control/data/address *dependency* (S3, definition of $\kappa'_2(d)$),
- $d \to e$ arises from *reads-from* (M7a),
- $d \to e$ arises from *blocking* (M7b).

In PwT-MCA$_2$, it is possible for rf to contradict $<$. In this case, we use a dotted arrow for rf: $d \dashrightarrow e$ indicates that $e < d$.

[7] Jagadeesan et al. [2020] erroneously elide the required synchronization on acquiring reads.

P6a and P6b ensure that $d \xrightarrow{\mathsf{rf}} e$ implies $d < e$ when the actions come from different threads. However, we may have $d \xrightarrow{\mathsf{rf}} e$ and $e < d$ within a thread, as between $(\mathsf{W}x2)$ to $(\mathsf{R}^{\mathsf{acq}}x2)$ in INTERNAL-ACQ, thus allowing this execution. M7b and S6a′ are sufficient to stop stale reads within a thread. For example, it prevents a read of 1 in $x := 1\,;\, x := 2\,;\, r := x$.

With the weakening of S6a, we must be careful not to allow spurious pairs to be added to the rf relation. For example, $[\![\mathtt{if}(b)\{r := x \parallel x := 1\}\,\mathtt{else}\,\{r := x\,;\, x := 1\}]\!]$ should not include $(\mathsf{R}x1) \to (\mathsf{W}x1)$, taking rf from the left and $<$ from the right. The use of "respects" in I6 and I7 ensures this.

As a consequence of dropping M7c, sequential rf must be validated during pomset construction, rather than post-hoc. In §6, we show how to construct program order (po) for complete pomsets using phantom events (π). Using this construction, the following lemma gives a post-hoc verification technique for rf. Let π^{-1} be the inverse of π.

LEMMA 4.3. *If $P \in [\![S]\!]_{\mathsf{mca2}}$ is complete, then for every $d \xrightarrow{\mathsf{rf}} e$ either*

- *external fulfillment:* $d < e$ *and if* $\lambda(c)$ blocks $\lambda(e)$ *then either* $c \le d$ *or* $e \le c$, *or*
- *internal fulfillment:* $(\exists d' \in \pi^{-1}(d))(\exists e' \in \pi^{-1}(e))$
  $d' \xrightarrow{\mathsf{po}} e'$ *and* $(\nexists c)\ \lambda(c)$ blocks $\lambda(e)$ *and* $d' \xrightarrow{\mathsf{po}} c \xrightarrow{\mathsf{po}} e'$.

These mimic the *external consistency* requirements of Arm8 [Alglave et al. 2021].

5 PwT-MCA RESULTS

Prop. 6.1 of Jagadeesan et al. [2020] establishes a compositional principle for proving that programs validate formula in past-time temporal logic. The principal is based entirely on the pomset order relation. Its proof, and all of the no-thin-air examples in [Jagadeesan et al. 2020, §6] hold equally for the models described here.

In the supplementary material, we show that PwT-MCA_1 supports the optimal lowering of relaxed accesses to Arm8 and that PwT-MCA_2 supports the optimal lowering of *all* accesses to Arm8. The proofs are based on two recent characterizations of Arm8 [Alglave et al. 2021]. For PwT-MCA_1, we use *External Global Consistency*. For PwT-MCA_2, we use *External Consistency*.

In the supplementary material, we also sketch a proof of sequential consistency for local-data-race-free programs. The proof uses *program order*, which we construct for C11 in §6. The same construction works for PwT-MCA. (This proof sketch assumes there are no RMW operations.)

The semantics validates many peephole optimizations, such as reorderings on relaxed access:

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

Here $\mathsf{id}(M)$ is the set of locations and registers that occur in M. Using augmentation closure, the semantics also validates roach-motel reorderings [Sevčík 2008]. For example, on read/write pairs:

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

Notably, the semantics does *not* validate read-introduction. When combined with if-introduction (§8.3), read-introduction can break temporal reasoning. This combination is allowed by speculative operational models. See §9 for a discussion.

6 PwT-C11: POMSETS WITH PREDICATE TRANSFORMERS FOR C11

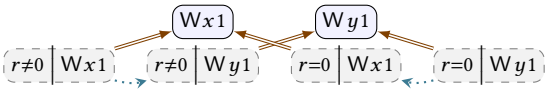
PwT can be used to generate semantic dependencies to prohibit thin-air executions of C11, while preserving optimal lowering for relaxed access. We follow the approach of Paviotti et al. [2020],

using our semantics to generate C11 candidate executions with a dependency relation, then applying the axioms of RC11 [Lahav et al. 2017]. The No-Thin-Air axiom of RC11 is overly restrictive, requiring that $(\mathsf{rf} \cup \mathsf{po})$ be acyclic. Instead, we require that $(\mathsf{rf} \cup <)$ is acyclic. This is a more precise categorization of thin-air behavior, and it allows aggressive compiler optimizations that would be erroneously forbidden by RC11's original No-Thin-Air axiom.

The chief difficulty is instrumenting our semantics to generate program order, for use in the various axioms of C11. Using the obvious construction (described in the proof of Lemma 6.2), program order (po) is a pre-order, which may include cycles due to coalescing. For example:

$$\texttt{if}(r)\{x := 1; y := 1\}\,\texttt{else}\,\{y := 1; x := 1\} \qquad \boxed{\mathrm{W}x1} \cdots \boxed{\mathrm{W}y1}$$

We solve this by adding *phantom* events. The function π maps phantom events to *real* events. For this program, we have the following PwT-po. (We visualize po using a dotted arrow $\cdots\!\!\rightarrow$, and π using a double arrow $\Longrightarrow$.)



Once the pomset is completed, r will be known, causing all the preconditions to be either tautological or unsatisfiable. We can then extract program order by restricting phantom events to have tautological preconditions (Def. 6.3). Thus, our strategy for C11 is to first construct a complete PwT-po, then extract top-level program order, then apply the axioms of RC11. We refer to a PwT-po that survives this filtering as a PwT-C11.

Definition 6.1. A PwT-po is a PwT (Def. 3.4) equipped with relations π and po such that

(M8) $\pi : (E \to E)$ is an idempotent function capturing *merging*, such that
let $R = \{e \mid \pi(e){=}e\}$ be *real* events, let $\overline{R} = (E \setminus R)$ be *phantom* events,
let $S = \{e \mid \forall d.\ \pi(d){=}e \Rightarrow d{=}e\}$ be *simple* events, let $\overline{S} = (E \setminus S)$ be *compound* events,

(M8a) $\lambda(e) = \lambda(\pi(e))$, (M8b) if $e \in \overline{S}$ then $\kappa(e) \vdash \bigvee_{\{c \in \overline{R} \mid \pi(c)=e\}} \kappa(c)$.

(M9) po $\subseteq (S \times S)$ is a partial order capturing *program order*.

A PwT-po is *complete* if

(C3) if $e \in R$ then $\kappa(e)$ is a tautology, (C5) $\checkmark$ is a tautology.

A complete PwT-po is a PwT-C11 if it additionally satisfies the axioms of RC11.

Since π is idempotent, we have $\pi(\pi(e)) = \pi(e)$. Equivalently, we could require $\pi(e) \in R$.

We use π to partition events E in two ways: we distinguish *real* events R from *phantom* events $\overline{R}$; we distinguish *simple* events S from *compound* events $\overline{S}$. From idempotency, it follows that all phantom events are simple ($\overline{R} \subseteq S$) and all compound events are real ($\overline{S} \subseteq R$). In addition, all phantom events map to compound events (if $e \in \overline{R}$ then $\pi(e) \in \overline{S}$).

Lemma 6.2. *If P is a* PwT *then there is a* PwT-po *P'' that conservatively extends it.*

Proof. The proof strategy is as follows: We extend the semantics of Fig. 1 with po. The obvious definition gives us a preorder rather than a partial order. To get a partial order, we replay the semantics without merging to get an *unmerged* pomset P'; the construction also produces the map π. We then construct P'' as the union of P and P', using the dependency relation from P.

We extend the semantics with po as follows. For pomsets with at most one event, po is the identity. For sequential composition, $\mathsf{po} = \mathsf{po}_1 \cup \mathsf{po}_2 \cup E_1 \times E_2$. For parallel composition and the conditional, $\mathsf{po} = \mathsf{po}_1 \cup \mathsf{po}_2$. As noted at the beginning of this section, po may contain cycles. To find an acyclic po$'$, we replay the construction of P to get P'. When building P', we require disjoint union in s1 and r1: $E' = E'_1 \uplus E'_2$. If and event is unmerged in P ($e \in E_1 \uplus E_2$) then we choose the same

event name for P'. If an event is merged in P ($e \in E_1 \cap E_2$) then we choose fresh event names—e'_1 and e'_2—and extend π accordingly: $\pi(e'_1) = \pi(e'_2) = e$. In P', we take $\leq' = \mathsf{po}'$.

To arrive at P'', we take (1) $E'' = E \cup E'$, (2) $\lambda'' = \lambda \cup \lambda'$, (3a) if $e \in E$ then $\kappa''(e) = \kappa(e)$, (3b) if $e \in E' \setminus E$ then $\kappa''(e) = \kappa'(e)$, (4) $\tau''^D = \tau^{(\pi^{-1}(D))}$, (5) $\checkmark'' = \checkmark$, (6) $d <'' e$ exactly when $\pi(d) < \pi(e)$, (7) $\mathsf{po}'' = \mathsf{po}'$, and (8) π'' is the constructed merge function. □

Definition 6.3. For a PwT-po, let $\mathsf{extract}(P)$ be the projection of P onto the set $\{e \in E_1 \mid e \text{ is simple and } \kappa_1(e) \text{ is a tautology}\}$.

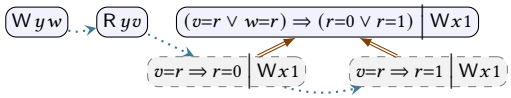
By definition, $\mathsf{extract}(P)$ includes the simple events of P whose preconditions are tautologies. These are already in program order, as per item 7 of the proof. The dependency order is derived from the real events using π, as per item 6.

The following lemma (immediate from M8b) shows that if P is *complete*, then $\mathsf{extract}(P)$ includes at least one simple event for every compound event in P.

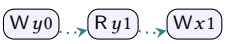
Lemma 6.4. *If P is a complete* PwT-po *with compound event e, then there is a phantom event $c \in \pi^{-1}(e)$ such that $\kappa(c)$ is a tautology.*

A pomset in the image of extract is a C11 *candidate execution*. As an example, consider Java Causality Test Case 6 [Pugh 2004]. Taking $w = 0$ and $v = 1$, the PwT-po on the left below produces the candidate execution on the right.

$y := w; r := y;$ `if(`r`=0){`$x := 1$`};` `if(`r`=1){`$x := 1$`}`



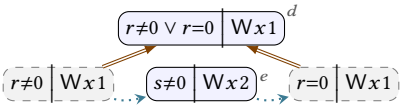
$y := 0; r := y;$ `if(`r`=0){`$x := 1$`};` `if(`r`=1){`$x := 1$`}`



We write $[\![\cdot]\!]^{\mathsf{po}}$ for the semantic function defined by applying the construction of Lemma 6.2 to the base semantics of 1.

The dependency calculation of $[\![\cdot]\!]^{\mathsf{po}}$ is sufficient for C11; however, it ignores synchronization and coherence completely. For example, consider:

`if(`r`){`$x := 1$`};` `if(`s`){`$x := 2$`};` `if(!`r`){`$x := 1$`}` (‡)



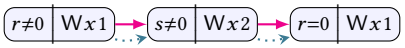
Adding a pair of reads to complete the pomset, we can extract the following candidate executions.

$r := y; s := z;$ `if(`r`){`$x := 1$`};` `if(`s`){`$x := 2$`};` `if(!`r`){`$x := 1$`}`

It is somewhat surprising that the writes are independent of both reads!

In PwT-mca, delay stops the merge in (‡).

`if(`r`){`$x := 1$`};` `if(`s`){`$x := 2$`};` `if(!`r`){`$x := 1$`}`



It is possible to mimic this in PwT-C11, without introducing extra dependencies: one can filter executions post-hoc using the relation $\sqsubseteq$, defined as follows:

$$\pi(d) \sqsubseteq \pi(e) \text{ if } d \overset{\mathsf{po}}{\cdots\!\rightarrow} e \text{ and } \lambda(d) \text{ delays } \lambda(e).$$

In (‡), we have both $d \sqsubseteq e$ and $e \sqsubseteq d$. To rule out (‡), it suffices to require that $\sqsubseteq$ is a partial order.

Table 1. Tool results for supported Java Causality Test Cases [Pugh 2004]. ⊥ indicates the tool failed to run for this test due to a memory overflow. Tests run on an Intel i9-9980HK with 64 GB of memory. For context, results for the MRD, MRD_{IMM}, and MRD_{C11} are also included [Paviotti et al. 2020].

Test	PwT-C11	MRD	MRD_{IMM}	MRD_{C11}
TC1	✔	✔	✔	✔
TC2	✔	✔	✔	✔
TC3	✔	✔	✔	✔
TC4	✔	✔	✔	✔
TC5	✔	✔	✔	✔
TC6	✔	✔	✔	✔
TC7	✔	✔	✔	✔
TC8	✔	✔	✔	✔

Test	PwT-C11	MRD	MRD_{IMM}	MRD_{C11}
TC9	✔	✔	✔	✔
TC10	✔	✔	✔	✔
TC11	⊥	✔	✔	✔
TC12	⊥	–	–	–
TC13	✔	✔	✔	✔
TC17	✔	✘	✔	✘
TC18	✔	✘	✔	✘

Program (‡) shows that the definition of semantic dependency is up for debate in C11. The International Standard Organization's C++ concurrency subgroup acknowledges that semantic dependency (sdep) would address the Out-of-Thin-Air problem: *"Prohibiting executions that have cycles in* (rf ∪ sdep) *can therefore be expected to prohibit Out-of-Thin-Air behaviors"* [McKenney et al. 2016]. PwT-C11 resolves program structure into a dependency relation—not a complex state—that is precise and easily adjusted. As refinements are made to C11, PwT-C11 can accommodate these and test them automatically.

7 PwTer: AUTOMATIC LITMUS TEST EVALUATOR

PwTer automatically and exhaustively calculates the allowed outcomes of litmus tests for the PwT, PwT-po, and PwT-C11 models, obviating the need for error-prone hand evaluation. It is built in OCaml, using Z3 [de Moura and Bjørner 2008] to judge the truth of predicates.

PwTer allows several modes of evaluation: it can evaluate the rules of Fig. 1, implementing PwT; it can generate program order according to §6, implementing PwT-po; and similar to MRD [Paviotti et al. 2020], it can construct C11-style pre-executions and filter them according to the rules of RC11 as described in §6, implementing PwT-C11. Finally, PwTer also allows us to toggle the complete check of Def. 3.4, providing an interface for understanding how fragments of code might compose by exposing preconditions and termination conditions that are not yet tautologies.

We have run PwTer over the Java Causality Tests [Pugh 2004] supported in the input syntax, and tabulated the results in Table 1. For context, we have included the results of MRD for the Java Causality tests [Paviotti et al. 2020]. Note that MRD and MRD_{C11} do not give the correct outcome on TC17–18—the reason is that local invariant reasoning in MRD is too constrained (see §3.8).

For larger test cases, the tool takes exponentially longer to compute, and for the largest tests the memory footprint is too large for even a well-equipped computer. The compositional nature of the semantics makes tool building practical, but it is not enough to make it scalable for large tests. In combination with the rules for reads and writes, the definitions of $SEQ(\mathcal{P}_1, \mathcal{P}_2)$ and $IF(\phi, \mathcal{P}_1, \mathcal{P}_2)$ have exponential complexity. This is compounded by the hidden complexity of calculating the possible merges between pomsets through union in rules s1 and r1. Significant effort has been put into throwing away spurious merges early in PwTer, so that executing the tool remains manageable for small examples. Some further optimizations may be possible within the tool to improve the situation further, such as killing "dead-end" pomsets at each sequence operator, or by doing a directed search for particular execution outcomes. PwTer is available in the supplementary material.

8 REFINEMENTS AND ADDITIONAL FEATURES

In the paper so far, we have assumed that registers are assigned at most once. We have done this primarily for readability. In the first subsection below, we drop this assumption, instead using substitution to rename registers. We use a set of registers indexed by event identifier: $\mathcal{S}_{\mathcal{E}} = \{s_e \mid e \in \mathcal{E}\}$. By assumption (§3.1), these registers do not appear in programs: $S[N/s_e] = S$. The resulting semantics satisfies redundant read elimination.

In the remainder of this section we consider several mostly-orthogonal features: address calculation, if-introduction, and read-modify-write operations. Address calculation and if-introduction do have some interaction, and we spell out the combined semantics in §8.5.

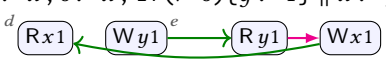
It is worth pointing out that address calculation and if-introduction only affect the semantics of read and write. RMWs introduce new infrastructure in order to ensure atomicity while supporting Arm's load-exclusive and store-exclusive operations.

These extensions preserve all of the program transformation discussed thus far, and apply equally to the various semantics we have discussed: PwT, PwT-MCA$_1$, PwT-MCA$_2$, and PwT-C11. The results discussed in §5 also apply equally, with the exception of RMWs, which are excluded from the proof of DRF-SC and from the proof of lowering to Arm8.

8.1 Register Recycling and Redundant Read Elimination

JMM Test Case 2 [Pugh 2004] states the following execution should be allowed "since redundant read elimination could result in simplification of $r{=}s$ to true, allowing $y := 1$ to be moved early."

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



Under the semantics of Fig. 1, the precondition of e in the independent case is

$$(1{=}r \vee x{=}r) \Rightarrow (1{=}s \vee r{=}s) \Rightarrow (r{=}s), \qquad (*)$$

which is equivalent to $(x{=}r) \Rightarrow (1{=}s) \Rightarrow (r{=}s)$, which is not a tautology, and thus Fig. 1 requires order from d to e in order to complete the pomset.

This execution is allowed, however, if we rename registers using a map from event names to register names. By using this renaming, coalesced events must choose the same register name. In the above example, the precondition of e in the independent case becomes

$$(1{=}s_e \vee x{=}s_e) \Rightarrow (1{=}s_e \vee s_e{=}s_e) \Rightarrow (s_e{=}s_e), \qquad (**)$$

which is a tautology. In $(**)$, the first read resolves the nondeterminism in both the first and the second read. Given the choice of event names, the outcome of the second read is predetermined! In $(*)$, the second read remains nondeterministic, even if the events are destined to coalesce.

Test Cases 17–18 [Pugh 2004] also require coalescing of reads. Contrary to the claim, the semantics of Jagadeesan et al. validates neither redundant load elimination nor these test cases.

Definition 8.1. Let $[\![\cdot]\!]$ be defined as in Fig. 1, changing R4 of *READ*:
(R4a) if $e \in E \cap D$ then $\tau^D(\psi) \equiv (\kappa(e) \Rightarrow v{=}s_e) \Rightarrow \psi[s_e/r]$,
(R4b) if $e \in E \setminus D$ then $\tau^D(\psi) \equiv (\kappa(e) \Rightarrow (v{=}s_e \vee x{=}s_e)) \Rightarrow \psi[s_e/r]$,
(R4c) if $E = \emptyset$ then $\tau^D(\psi) \equiv (\forall s)\ \psi[s/r]$.

With this semantics, it is straightforward to see that redundant load elimination is sound:

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

As a further example, consider Fig. 5 of Sevčík and Aspinall [2008], referenced by Paviotti et al. [2020, §6.4]. Consider the case where the reads are merged, both seeing 1:

$$r := y\,;\ \texttt{if}(r{=}1)\{s := y\,;\ x := s\}\ \texttt{else}\ \{x := 1\} \qquad \boxed{\mathsf{R}\,y1} \quad \boxed{\phi \mid \mathsf{W}x1}$$

In order to be independent of both reads, we take the precondition ϕ to be:

$$(1{=}r \vee y{=}r) \Rightarrow [r{=}1 \wedge \big((1{=}s \vee y{=}s) \Rightarrow s{=}1\big)] \vee [r{\neq}1]$$

Then collapsing r and s and substituting the initial value of y (say 0), we have a tautology:

$$(1{=}r \vee 0{=}r) \Rightarrow [r{=}1 \wedge \big((1{=}r \vee 0{=}r) \Rightarrow r{=}1\big)] \vee [r{\neq}1]$$

Support for register recycling requires predicate transformers, which allow substitution, rather than simple postconditions.

8.2 Read-Modify-Write Operations

To support RMWs, we extend the syntax:

$$S \ ::= \ \cdots \ \mid \ r := \mathsf{CAS}^{\mu,\nu}([L], M, N) \ \mid \ r := \mathsf{FADD}^{\mu,\nu}([L], M) \ \mid \ r := \mathsf{EXCHG}^{\mu,\nu}([L], M)$$

We require that r does not occur in L. Semantically, we add a relation $\xrightarrow{\mathsf{rmw}} \subseteq E \times E$ that relates the read of a successful RMW to the succeeding write.

Definition 8.2. Extend the definition of a pomset as follows.

(M10) $\mathsf{rmw} : E \to E$ is a partial function capturing read-modify-write *atomicity*, such that
(M10a) if $d \xrightarrow{\mathsf{rmw}} e$ then $\lambda(e)$ blocks $\lambda(d)$,
(M10b) if $d \xrightarrow{\mathsf{rmw}} e$ then $d < e$,
(M10c) if $\lambda(c)$ overlaps $\lambda(d)$ and $d \xrightarrow{\mathsf{rmw}} e$ then $c < e$ implies $c \leq d$ and $d < c$ implies $e \leq c$.

Extend the definition of *SEQ*, *IF* and *PAR* to include:

(S10) (I10) (P10) $\mathsf{rmw} = (\mathsf{rmw}_1 \cup \mathsf{rmw}_2)$,

Let *READ′* be defined as for *READ*, adding the constraint:

(R4d) if $(E \cap D) = \emptyset$ then $\tau^D(\psi) \equiv \psi$.

If $P \in \mathit{CAS}(r, x, M, N, \mu, \nu)$ then $P \in \mathit{SEQ}(\mathit{READ}'(r, x, \mu), \mathit{IF}(r{=}M, \mathit{WRITE}(x, N, \nu), \mathit{SKIP}))$ and

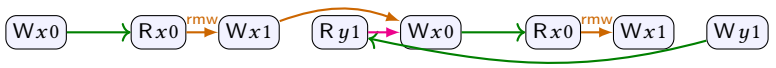
(U10) if $\lambda(e)$ is a write then there is a read $\lambda(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.

$$[\![r := \mathsf{CAS}^{\mu,\nu}(x, M, N)]\!] = \mathit{CAS}(r, x, M, N, \mu, \nu)$$

FADD and EXCHG are similar. These definitions ensure atomicity and support lowering to Arm load/store exclusive operations. See Jagadeesan et al. [2020] for examples.

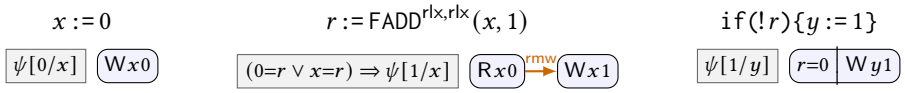
One subtlety of the definition is that we use *READ′* rather than *READ*: for RMWs, the independent case for a read is the same as the empty case. To see why this should be, consider the relaxed variant of the CDRF example from Lee et al. [2020], using *READ* rather than *READ′*.

$$x := 0\,;\ (r := \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x, 1)\,;\ \texttt{if}(!r)\{\texttt{if}(y)\{x := 0\}\} \ \ \|\ r := \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x, 1)\,;\ \texttt{if}(!r)\{y := 1\})$$



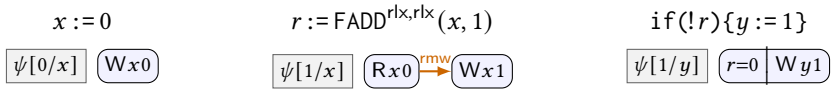
A write should only be visible to one FADD instruction, but here the write of 0 is visible to two! This is allowed because, using *READ* instead of *READ′*, no order is required from $(\mathsf{R}x0)$ to $(\mathsf{W}y1)$ in the last thread.

To see why, consider the independent transformers of the last thread and initializer:



After sequencing, the precondition of $(\mathsf{W}y1)$ is a tautology: $(0{=}r \vee 0{=}r) \Rightarrow r{=}0$.

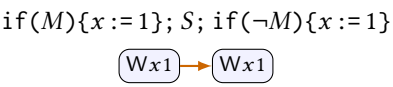
By including R4d, *READ'* constrains the independent predicate transformer of the FADD:



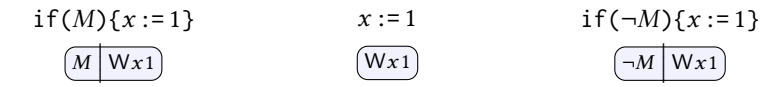
After sequencing, the precondition of $(\mathsf{W}y1)$ is $r{=}0$, which is *not* a tautology. This forces any top-level pomset to include dependency order from $(\mathsf{R}x0)$ to $(\mathsf{W}y1)$.

8.3 If-Introduction (aka Case Analysis)

In order to model sequential composition, we must allow inconsistent predicates in a single pomset, unlike PwP [Jagadeesan et al. 2020]. For example, if $S = (x := 1)$, then the semantics Fig. 1 does *not* allow:

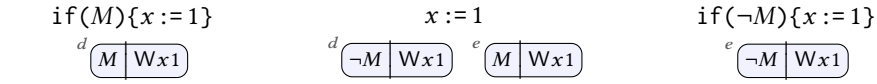


However, if $S = (\texttt{if}(\neg M)\{x := 1\}; \texttt{if}(M)\{x := 1\})$, then it *does* allow the execution. Looking at the initial program:



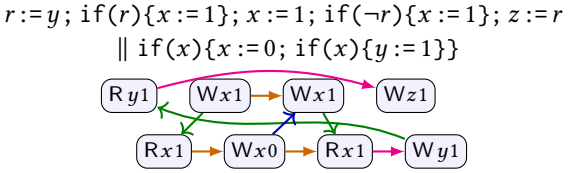
The difficulty is that the middle action can coalesce either with the right action, or the left, but not both. Thus, we are stuck with some non-tautological precondition. Our solution is to allow a pomset to contain many events for a single action, as long as the events have disjoint preconditions.

Def. 8.3 allows the execution, by splitting the middle command:



Coalescing events gives the desired result.

This is not simply a theoretical question; it is observable. For example, the semantics of Fig. 1 does not allow the following, since it must add order in the first thread from the read of y to one of the writes to x.



We show the rules for write and read.[8] The rule for fences requires similar treatment.

[8]The Coq development uses ⊨ rather than ≡ in W3 and R3. Given the quantification over ϕ, these are equivalent.

Definition 8.3. If $P \in \mathit{WRITE}(x, M, \mu)$ then $(\exists v : E \to \mathcal{V})$ $(\exists \phi : E \to \Phi)$
(w1) if $\kappa(d) \wedge \kappa(e)$ is satisfiable then $d = e$,
(w2) $\lambda(e) = \mathsf{W}^{\mu} x v_e$,
(w3) $\kappa(e) \equiv \phi_e \wedge M{=}v_e$,
(w4) $\tau^D(\psi) \equiv \psi[M/x][\mathrm{K}(E)/\mathsf{Q}_x]$,
(w5) $\checkmark \equiv \mathrm{K}(E)$,
(w6) $\phi_e[N/s_d] = \phi_e$.
If $P \in \mathit{READ}(r, x, \mu)$ then $(\exists v : E \to \mathcal{V})$ $(\exists \phi : E \to \Phi)$
(R1) if $\phi_d \wedge \phi_e$ is satisfiable then $d = e$,
(R2) $\lambda(e) = \mathsf{R}^{\mu} x v_e$
(R3) $\kappa(e) \equiv \phi_e \wedge \mathsf{Q}_x$,
(R4) $\tau^D(\psi) \equiv \bigwedge_{e \in E \cap D} \phi_e \Rightarrow (\kappa(e) \Rightarrow v_e{=}s_e) \Rightarrow \psi[s_e/r]$
$\wedge \bigwedge_{e \in E \setminus D} \phi_e \Rightarrow (\kappa(e) \Rightarrow (v_e{=}s_e \vee x{=}s_e)) \Rightarrow \psi[s_e/r]$
$\wedge (\bigwedge_{e \in E} \neg \phi_e) \Rightarrow (\forall s)\ \psi[s/r]$
(R5a) if $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{tt}$,
(R5b) if $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \mathrm{K}(E)$,
(R6) $\phi_e[N/s_d] = \phi_e$.

The definition allows multiple events to represent a single action, with disjoint preconditions. The predicate transformers are derived from those defined for the conditional. w6 and R6 require that the predicates do not mention registers in $\mathcal{S}_{\mathcal{E}}$.

This modification validates Lemma 3.6e, f, and d as equations.

We show how to combine address calculation and if-introduction in §8.5.

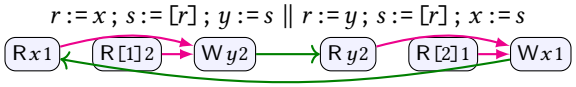
8.4 Address Calculation

Inevitably, address calculation complicates the definitions of *WRITE* and *READ*. In this section, we develop a flat memory model, which does not deal with provenance [Lee et al. 2018].

Definition 8.4. Within a pomset P, let $\mathrm{K}(x) = \bigvee \{\kappa(e) \mid e \in E \wedge \lambda(e) = \mathsf{W}x\}$.
If $P \in \mathit{WRITE}(L, M, \mu)$ then $(\exists \ell \in \mathcal{V})$ $(\exists v \in \mathcal{V})$
(w1) if $|E| \leqslant 1$,
(w2) $\lambda(e) = \mathsf{W}^{\mu}[\ell]v$,
(w3) $\kappa(e) \equiv L{=}\ell \wedge M{=}v$,
(w4) $\tau^D(\psi) \equiv \bigwedge_{k \in \mathcal{V}} L{=}k \Rightarrow \psi[M/[k]][\mathrm{K}([k])/\mathsf{Q}_{[k]}]$,
(w5) $\checkmark \equiv \mathrm{K}(E)$.

If $P \in \mathit{READ}(r, L, \mu)$ then $(\exists \ell \in \mathcal{V})$ $(\exists v \in \mathcal{V})$
(R1) if $|E| \leqslant 1$,
(R2) $\lambda(e) = \mathsf{R}^{\mu}[\ell]v$
(R3) $\kappa(e) \equiv L{=}\ell \wedge \mathsf{Q}_{[\ell]}$,
(R4a) if $e \in E \cap D$ then $\tau^D(\psi) \equiv (\kappa(e) \Rightarrow v{=}s_e) \Rightarrow \psi[s_e/r]$,
(R4b) if $e \in E \setminus D$ then $\tau^D(\psi) \equiv (\kappa(e) \Rightarrow (v{=}s_e \vee [\ell]{=}s_e)) \Rightarrow \psi[s_e/r]$,
(R4c) if $E = \emptyset$ then $\tau^D(\psi) \equiv (\forall s)\ \psi[s/r]$,
(R5a) if $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{tt}$,
(R5b) if $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \mathrm{K}(E)$.

The combination of read-read independency (§3.7) and address calculation is somewhat delicate. Consider the following program, from Jagadeesan et al. [2020, §5], where initially $x{=}0$, $y{=}0$, $[0]{=}0$, $[1]{=}2$, and $[2]{=}1$. It should only be possible to read 0, disallowing the attempted execution below:



(ADDR1)

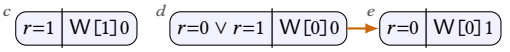
This execution would become possible, however, if we were to remove $(L{=}\ell)$ from R4—it is included in κ. In this case, $(\mathsf{R}y2)$ would not necessarily be dependency ordered before $(\mathsf{W}x1)$.

8.5 Combining Address Calculation and If-Introduction

Def. 8.4 is naive with respect to merging events. Consider the following example:

Merging, we have:

$$\texttt{if}(M)\{[r]:=0;\ [0]:=!r\}\ \texttt{else}\ \{[r]:=0;\ [0]:=!r\}$$

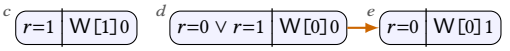


The precondition of $\mathsf{W}[0]0$ is a tautology; however, this is not possible for $([r]:=0;\ [0]:=!r)$ alone. Def. 8.5 enables this execution using if-introduction. Under this semantics, we have:

$$[r]:=0 \qquad\qquad [0]:=!r$$

Sequencing and merging:

$$[r]:=0;\ [0]:=!r$$



The precondition of $(\mathsf{W}[0]0)$ is a tautology, as required.

Def. 8.5 is a mash-up of the Def. 8.3 and Def. 8.4.

Definition 8.5. If $P \in \mathit{WRITE}(L, M, \mu)$ then $(\exists \ell : E \to \mathcal{V})\ (\exists v : E \to \mathcal{V})\ (\exists \phi : E \to \Phi)$

(w1) if $\kappa(d) \wedge \kappa(e)$ is satisfiable then $d = e$,
(w2) $\lambda(e) = \mathsf{W}^{\mu}[\ell_e]v_e$,
(w3) $\kappa(e) \equiv \phi_e \wedge L{=}\ell_e \wedge M{=}v_e$,
(w4) $\tau^D(\psi) \equiv \bigwedge_{k\in\mathcal{V}} L{=}k \Rightarrow \psi[M/k][\mathsf{K}([k])/\mathsf{Q}_{[k]}]$,
(w5) $\checkmark \equiv \mathsf{K}(E)$,
(w6) $\phi_e[N/s_d] = \phi_e$.

If $P \in \mathit{READ}(r, L, \mu)$ then $(\exists \ell : E \to \mathcal{V})\ (\exists v : E \to \mathcal{V})\ (\exists \phi : E \to \Phi)$

(R1) if $\kappa(d) \wedge \kappa(e)$ is satisfiable then $d = e$,
(R2) $\lambda(e) = \mathsf{R}^{\mu}[\ell_e]v_e$
(R3) $\kappa(e) \equiv \phi_e \wedge L{=}\ell_e \wedge \mathsf{Q}_{[\ell_e]}$,
(R4) $\tau^D(\psi) \equiv \bigwedge_{e\in E\cap D} \phi_e \Rightarrow (\kappa(e) \Rightarrow v_e{=}s_e) \Rightarrow \psi[s_e/r]$
$\wedge \bigwedge_{e\in E\setminus D} \phi_e \Rightarrow (\kappa(e) \Rightarrow (v_e{=}s_e \vee [\ell_e]{=}s_e)) \Rightarrow \psi[s_e/r]$
$\wedge(\bigwedge_{e\in E} \neg\phi_e) \Rightarrow (\forall s)\ \psi[s/r]$,
(R5a) if $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{tt}$,
(R5b) if $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \mathsf{K}(E)$,
(R6) $\phi_e[N/s_d] = \phi_e$.

9 RELATED WORK

Marino et al. [2015] argue that the "silently shifting semicolon" is sufficiently problematic for programmers that concurrent languages should guarantee sequential abstraction, despite the performance penalties (see also Liu et al. [2021]). In this paper, we take the opposite approach. We have attempted to find the most intellectually tractable model that encompasses all of the messiness of relaxed memory.

There are two prior studies of relaxed memory that include precise calculation of semantic dependencies—neither gives the semantics of sequential composition in direct style. First, Paviotti et al. [2020] defined MRD, which calculates dependencies using event structures rather than logic. This strategy is brittler than ours, leading to false positives (§3.8). Second, Jagadeesan et al. [2020] defined PwP, using logical entailment to define dependency. Although PwT is based on PwP, there are many differences. Some of these are motivated by requirements unique to PwT (see §3.9). Other differences are stylistic: For example, we use termination *conditions* rather than termination *actions*—our formulation fixes an error in Jagadeesan et al.'s definition of parallel composition. We also fix an error in their treatment of redundant read elimination (§8.1).

Kavanagh and Brookes [2018] define a semantics using pomsets without preconditions. Instead, their model uses syntactic dependencies, thus invalidating many compiler optimizations. They also require a fence after every relaxed read on Arm8. Pichon-Pharabod and Sewell [2016] use event structures to calculate dependencies, combined with an operational semantics that incorporates program transformations. This approach seems to require whole-program analysis.

Other studies of relaxed memory can be categorized by their approach to dependency calculation. Hardware models use syntactic dependencies [Alglave et al. 2014]. Many software models do not bother with dependencies at all [Batty et al. 2011; Cox 2016; Watt et al. 2020, 2019]. Others have strong dependencies that disallow compiler optimizations and efficient implementation, typically requiring fences for every relaxed read on Arm8 [Boehm and Demsky 2014; Dolan et al. 2018; Jeffrey and Riely 2016; Lahav et al. 2017; Lamport 1979]. Many of the most prominent models are operational models based on speculative execution [Chakraborty and Vafeiadis 2019; Cho et al. 2021; Jagadeesan et al. 2010; Kang et al. 2017; Lee et al. 2020; Manson et al. 2005].

Morally, PwT fits between the strong models and the speculative ones. Looking at the details, however, PwT-MCA is incomparable to both RC11 [Lahav et al. 2017] and the promising semantics [Kang et al. 2017], to take two examples. RC11 allows non-MCA behaviors that PwT-MCA disallows. PwT-MCA has a weaker notion of coherence than the promising semantics.

Jagadeesan et al. [2020] argue that the speculative models allow too many executions, resulting in a failure of temporal reasoning and potentially jeopardizing type safety and other security properties. In a similar vein, Cho et al. [2021] argue that local DRF guarantees are violated when read-introduction is followed by if-introduction, branching on the read just introduced. These optimizations are validated by the speculative models—Cho et al. manage to avoid the problem by adopting a sub-optimal lowering for RMWs. PwT does not suffer from this problem, since PwT does not validate read-introduction. There appears to be a genuine tension between temporal reasoning, as supported by PwT, and read-introduction, as supported by the speculative models.

Other work in relaxed memory has shown that tooling is especially useful to researchers, architects, and language specifiers, enabling them to build intuitions experimentally [Alglave et al. 2014; Batty et al. 2011; Cooksey et al. 2019; Paviotti et al. 2020]. Unfortunately, it is not obvious that tools can be built for all thin-air-free models: the calculation of Pichon-Pharabod and Sewell [2016] does not have a termination proof for an arbitrary input; the enormous state space for the operational models of Kang et al. [2017] and Chakraborty and Vafeiadis [2019] is daunting for a tool builder—and as yet no tool exists for automatically evaluating these models. We described a tool for automatically evaluating PwT in §7.

10 LIMITATIONS AND FUTURE WORK

This paper is the first to present a direct denotational semantics for sequential composition that can be efficiently compiled to modern CPUs. We defined two models: PwT-C11 solves the out-of-thin-air problem for C11, and PwT-MCA solves it for safe languages such as Java and Javascript.

Our work has several limitations, providing opportunities for future work:

PwT-C11 can be lowered efficiently to any architecture supported by C11, but inherits the top-level axioms of RC11, compromising compositionality. PwT-MCA is as a compositional as a model of concurrent imperative programming can be, but is limited to MCA architectures for optimal lowering. It would be interesting to explore the middle ground to find a fully compositional model that supports optimal lowering to all modern architectures.

As mentioned in §9, some safety guarantees may be violated when read-introduction is followed by if-introduction, branching on the read just introduced. Nonetheless, read-introduction is ubiquitous in some compilers [Lee et al. 2017]. It would be interesting to know the cost of restricting this optimization. In a similar vein, PwT-MCA_1 is a simpler model than PwT-MCA_2, but requires fences on acquiring reads for Arm8. It would be illuminating to find out what the performance penalty is for these fences.

We have defined the soundness of compiler optimizations in the model, rather than contextually: S' is a sound refinement of S if $[\![S']\!] \subseteq [\![S]\!]$. This approach has several advantages—for example, it is immediate that a sound optimization is sound in any context. It also has a disadvantage: some

optimizations complicate the semantics. For example, PwT-MCA does not validate access elimination, such as store-forwarding and dead-write-removal—consider that complete executions of $[\![x := 1; r := x]\!]$ must include a read action and that complete executions of $[\![x := 1; x := 2]\!]$ must include two write actions. As another example, PwT-MCA does not validate the reverse inclusions for Lemma 3.6g—consider that $[\![\texttt{if}(r)\{x := 1\}\,\texttt{else}\,\{x := 2\}]\!]$ has an augmented (Lemma 3.7) execution with $(r{=}0 \mid \mathsf{W}x2) \rightarrow (r{\neq}0 \mid \mathsf{W}x1)$, whereas $[\![\texttt{if}(r)\{x := 1\}; \texttt{if}(!r)\{x := 2\}\}]\!]$ has no such execution. We expect that these optimizations can be validated, at the cost of complicating the semantics. For access elimination, it is likely sufficient to allow events with different actions to merge. For Lemma 3.6g, it is likely sufficient to encoding delay in the logic—the problem in the execution above is that delay introduces order regardless of whether preconditions are disjoint.

We have not treated loops, although we expect that the usual approach of showing continuity for all the semantic operations with respect to set inclusion would go through. Paviotti et al. [2020] use step-indexing to account for loops; perhaps this approach could be adapted.

While we have mechanized some proofs, most of our proofs are informal. In particular, we have only a pen-and-paper proof showing that PwT-MCA supports optimal lowering to Arm8. The same is true for local data race-freedom (LDRF-SC)—additionally, our proof sketch for LDRF-SC elides RMWs, which have caused complications in other models [Cho et al. 2021].

Supplementary material for this paper is available at https://weakmemory.github.io/pwt.

Acknowledgements

This paper has been greatly improved by the comments of the anonymous reviewers. James Riely was supported by the National Science Foundation under grant No. CCR-1617175. Mark Batty and Simon Cooksey were supported by the EPSRC under grant Nos. EP/V000470/1 and EP/R032971/1, and by VeTSS. Anton Podkopaev was supported by JetBrains Research.

REFERENCES

Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *ACM Trans. Program. Lang. Syst.* 43, 2, Article 8 (July 2021), 54 pages. https://doi.org/10.1145/3458926

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

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

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

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

Hans-J. Boehm. 2019. Out-of-thin-air, Revisited, Again (Revision 2). https://wg21.link/p1217.

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

Stephen D. Brookes. 1996. Full Abstraction for a Shared-Variable Parallel Language. *Inf. Comput.* 127, 2 (1996), 145–163. https://doi.org/10.1006/inco.1996.0056

Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. 2007. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4421)*, Rocco De Nicola (Ed.). Springer, 331–346. https://doi.org/10.1007/978-3-540-71316-6_23

Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 100–110. http://dl.acm.org/citation.cfm?id=

3049844

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

Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. 2021. Modular data-race-freedom guarantees in the promising semantics. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 867–882. https://doi.org/10.1145/3453483.3454082

Simon Cooksey, Sarah Harris, Mark Batty, Radu Grigore, and Mikolás Janota. 2019. PrideMM: Second Order Model Checking for Memory Consistency Models. In *Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part II (Lecture Notes in Computer Science, Vol. 12233)*, Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas (Eds.). Springer, 507–525. https://doi.org/10.1007/978-3-030-54997-8_31

Russ Cox. 2016. Go's Memory Model. http://nil.csail.mit.edu/6.824/2016/notes/gomem.pdf.

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457. https://doi.org/10.1145/360933.360975

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

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

William Ferreira, Matthew Hennessy, and Alan Jeffrey. 1996. A Theory of Weak Bisimulation for Core CML. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 201–212. https://doi.org/10.1145/232627.232649

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

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

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

Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 194:1–194:30. https://doi.org/10.1145/3428262

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

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

Alan Jeffrey and James Riely. 2019. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. *Logical Methods in Computer Science* 15, 1 (2019), 25 pages. https://doi.org/10.23638/LMCS-15(1:33)2019

Jeehoon Kang. 2019. *Reconciling Low-Level Features of C with Compiler Optimizations.* Ph.D. Dissertation. Seoul National University, Seoul, South Korea. https://sf.snu.ac.kr/jeehoon.kang/thesis/

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189. http://dl.acm.org/citation.cfm?id=3009850

Ryan Kavanagh and Stephen Brookes. 2018. A denotational account of C11-style memory. *CoRR* abs/1804.04214 (2018), 13 pages. arXiv:1804.04214 http://arxiv.org/abs/1804.04214

Ori Lahav and Udi Boker. 2020. Decidable verification under a causally consistent shared memory. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 211–226. https://doi.org/10.1145/3385412.3385966

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 649–662. https://doi.org/10.1145/2837614.2837643

Ori Lahav and Viktor Vafeiadis. 2016. Explaining Relaxed Memory Models with Program Transformations. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9995)*, John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer, 479–495. https://doi.org/10.1007/978-3-319-48989-6_29

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

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

Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling high-level optimizations and low-level code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 125:1–125:28. https://doi.org/10.1145/3276495

Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. 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, 633–647. https://doi.org/10.1145/3062341.3062343

Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 362–376. https://doi.org/10.1145/3385412.3386010

Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2019. Accelerating Sequential Consistency for Java with Speculative Compilation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 16–30. https://doi.org/10.1145/3314221.3314611

Lun Liu, Todd Millstein, and Madanlal Musuvathi. 2021. Safe-by-Default Concurrency for Modern Programming Languages. *ACM Trans. Program. Lang. Syst.* 43, 3, Article 10 (Sept. 2021), 50 pages. https://doi.org/10.1145/3462206

Nuno Lopes. 2016. RFC: Killing undef and spreading poison. https://lists.llvm.org/pipermail/llvm-dev/2016-October/106182.html.

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

Daniel Marino, Todd D. Millstein, Madanlal Musuvathi, Satish Narayanasamy, and Abhayendra Singh. 2015. The Silently Shifting Semicolon. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 177–189. https://doi.org/10.4230/LIPIcs.SNAPL.2015.177

Ian A. Mason and Carolyn L. Talcott. 1992. References, Local Variables and Operational Reasoning. In *Proceedings of the Seventh Annual Symposium on Logic in Computer Science (LICS '92), Santa Cruz, California, USA, June 22-25, 1992*. IEEE Computer Society, 186–197. https://doi.org/10.1109/LICS.1992.185532

Paul E. McKenney, Alan Jeffrey, Ali Sezgin, and Tony Tye. 2016. Out-of-Thin-Air Execution is vacuous. http://wg21.link/p0422.

Robin Milner. 1977. Fully Abstract Models of Typed *lambda*-Calculi. *Theor. Comput. Sci.* 4, 1 (1977), 1–22. https://doi.org/10.1016/0304-3975(77)90053-6

Peter O'Hearn. 2007. Resources, Concurrency, and Local Reasoning. *Theor. Comput. Sci.* 375, 1-3 (April 2007), 271–307. https://doi.org/10.1016/j.tcs.2006.12.035

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

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

Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theor. Comput. Sci.* 5, 3 (1977), 223–255. https://doi.org/10.1016/0304-3975(77)90044-5

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

Vaughan R. Pratt. 1985. Some Constructions for Order-Theoretic Models of Concurrency. In *Logics of Programs, Conference, Brooklyn College, New York, NY, USA, June 17-19, 1985, Proceedings (Lecture Notes in Computer Science, Vol. 193)*, Rohit Parikh (Ed.). Springer, 269–283. https://doi.org/10.1007/3-540-15648-8_22

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

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

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

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

Jaroslav Sevčík 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, Vol. 5142)*, Jan Vitek (Ed.). Springer, 27–51. https://doi.org/10.1007/978-3-540-70592-5_3

Joel Spolsky. 2002. The Law of Leaky Abstractions. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.

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

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

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

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

A LOWERING PwT-MCA TO ARM

For simplicity, we restrict to top-level parallel composition.

A.1 Arm executions

Our description of Arm8 follows Alglave et al. [2021], adapting the notation to our setting.

Definition A.1. An *Arm8 execution graph*, G, is tuple $(E, \lambda, \mathsf{poloc}, \mathsf{lob})$ such that

(A1) $E \subseteq \mathcal{E}$ is a set of events,
(A2) $\lambda : E \to \mathcal{A}$ defines a label for each event,
(A3) $\mathsf{poloc} \subseteq E \times E$, is a per-thread, per-location total order, capturing *per-location program order*,
(A4) $\mathsf{lob} \subseteq E \times E$, is a per-thread partial order capturing *locally-ordered-before*, such that
 (A4a) $\mathsf{poloc} \cup \mathsf{lob}$ is acyclic.

The definition of lob is complex. Comparing with our definition of sequential composition, it is sufficient to note that lob includes

(L1) read-write dependencies, required by s3,
(L2) synchronization delay of $\bowtie_{\mathsf{sync}}$, required by s6a,
(L3) sc access delay of $\bowtie_{\mathsf{sc}}$, required by s6a,
(L4) write-write and read-to-write coherence delay of $\bowtie_{\mathsf{co}}$, required by s6a,

and that lob does *not* include

(L5) read-read control dependencies, required by s3,
(L6) write-to-read order of rf, required by m7c,
(L7) write-to-read coherence delay of $\bowtie_{\mathsf{co}}$, required by s6a.

Definition A.2. Execution G is $(\mathsf{co}, \mathsf{rf}, \mathsf{gcb})$*-valid*, under *External Global Consistency* (EGC) if

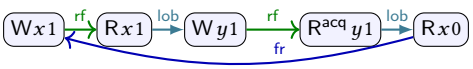
(A5) $\mathsf{co} \subseteq E \times E$, is a per-location total order on writes, capturing *coherence*,
(A6) $\mathsf{rf} \subseteq E \times E$, is a relation, capturing *reads-from*, such that
 (A6a) rf is surjective and injective relation on $\{e \in E \mid \lambda(e) \text{ is a } \mathsf{read}\}$,
 (A6b) if $d \xrightarrow{\mathsf{rf}} e$ then $\lambda(d)$ matches $\lambda(e)$,
 (A6c) $\mathsf{poloc} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}$ is acyclic, where $e \xrightarrow{\mathsf{fr}} c$ if $e \xleftarrow{\mathsf{rf}} d \xrightarrow{\mathsf{co}} c$, for some d,
(A7) $\mathsf{gcb} \supseteq (\mathsf{co} \cup \mathsf{rf})$ is a linear order such that
 (A7a) if $d \xrightarrow{\mathsf{rf}} e$ and $\lambda(c)$ blocks $\lambda(e)$ then either $c \xrightarrow{\mathsf{gcb}} d$ or $e \xrightarrow{\mathsf{gcb}} c$,
 (A7b) if $e \xrightarrow{\mathsf{lob}} c$ then either $e \xrightarrow{\mathsf{gcb}} c$ or $(\exists d)\ d \xrightarrow{\mathsf{rf}} e$ and $d \xrightarrow{\mathsf{poloc}} e$ but not $d \xrightarrow{\mathsf{lob}} c$.

Execution G is $(\mathsf{co}, \mathsf{rf}, \mathsf{cb})$*-valid* under *External Consistency* (EC) if

(A5) and (A6), as for EGC,
(A8) $\mathsf{cb} \supseteq (\mathsf{co} \cup \mathsf{lob})$ is a linear order such that if $d \xrightarrow{\mathsf{rf}} e$ then either
 (A8a) $d \xrightarrow{\mathsf{cb}} e$ and if $\lambda(c)$ blocks $\lambda(e)$ then either $c \xrightarrow{\mathsf{cb}} d$ or $e \xrightarrow{\mathsf{cb}} c$, or
 (A8b) $d \xleftarrow{\mathsf{cb}} e$ and $d \xrightarrow{\mathsf{poloc}} e$ and $(\nexists c)\ \lambda(c)$ blocks $\lambda(e)$ and $d \xrightarrow{\mathsf{poloc}} c \xrightarrow{\mathsf{poloc}} e$.

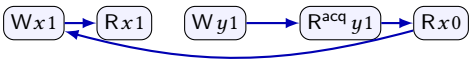
Alglave et al. [2021] show that EGC and EC are both equivalent to the standard definition of Arm8. They explain EGC and EC using the following example, which is allowed by Arm8.[9]

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



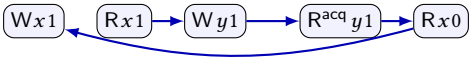
[9] We have changed an address dependency in the first thread to a data dependency.

EGC drops lob-order in the first thread using A7b, since $(\mathsf{W}x1)$ is not lob-ordered before $(\mathsf{W}y1)$.



(gcb)

EC drops rf-order in the first thread using A8b.



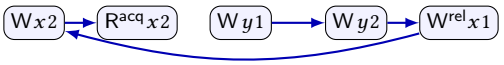
(cb)

A.2 Lowering PwT-MCA1 to Arm

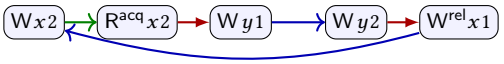
The optimal lowering for Arm8 is unsound for PwT-MCA$_1$. The optimal lowering maps relaxed access to `ldr`/`str` and non-relaxed access to `ldar`/`stlr` [Podkopaev et al. 2019]. In this section, we consider a suboptimal strategy, which lowers non-relaxed reads to (`dmb.sy`; `ldar`). Significantly, we retain the optimal lowering for relaxed access. In the next section we recover the optimal lowering by adopting an alternative semantics for M7c and s6a.

To see why the optimal lowering fails, consider the following attempted execution, where the final values of both x and y are 2.

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



(gcb)

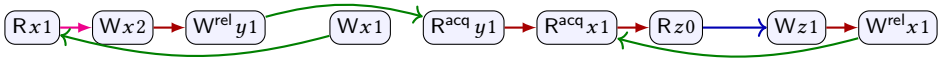


(<)

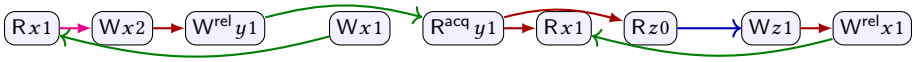
This attempted execution is allowed by Arm8, but disallowed by our semantics.

If the read of x in the execution above is changed from acquiring to relaxed, then our semantics allows the gcb execution, using the independent case for the read and satisfying the precondition of $(\mathsf{W}y1)$ by prepending $(\mathsf{W}x2)$. It may be tempting, therefore, to adopt a strategy of *downgrading* acquires in certain cases. Unfortunately, it is not possible to do this locally without invalidating important idioms such as publication. For example, consider that $(\mathsf{R}^{\mathsf{ra}}x1)$ is *not* possible for the second thread in the following attempted execution, due to publication of $(\mathsf{W}x2)$ via y:

$$x := x + 1;\ y^{\mathsf{rel}} := 1 \parallel x := 1;\ \mathtt{if}(y^{\mathsf{acq}}\ \&\&\ x^{\mathsf{acq}})\{s := z\} \parallel z := 1;\ x^{\mathsf{rel}} := 1$$



Instead, if the read of x is relaxed, then the publication via y fails, and $(\mathsf{R}x1)$ in the second thread is possible.



Using the suboptimal lowering for acquiring reads, our semantics is sound for Arm. The proof uses the characterization of Arm using EGC.

THEOREM A.3. *Suppose G_1 is $(\mathsf{co}_1, \mathsf{rf}_1, \mathsf{gcb}_1)$-valid for S under the suboptimal lowering that maps non-relaxed reads to* (`dmb.sy`; `ldar`). *Then there is a top-level pomset $P_2 \in [\![S]\!]$ such that $E_2 = E_1$, $\lambda_2 = \lambda_1$, $\mathsf{rf}_2 = \mathsf{rf}_1$, and $\leq_2 = \mathsf{gcb}_1$.*

PROOF. First, we establish some lemmas about Arm8.

LEMMA A.4. *Suppose G is $(\mathsf{co}, \mathsf{rf}, \mathsf{gcb})$-valid. Then $\mathsf{gcb} \supseteq \mathsf{fr}$.*

Proof. Using the definition of fr from A6c, we have $e \xleftarrow{\mathsf{rf}} d \xrightarrow{\mathsf{co}} c$, and therefore $\lambda(c)$ blocks $\lambda(e)$. Applying A7a, we have that either $c \xrightarrow{\mathsf{gcb}} d$ or $e \xrightarrow{\mathsf{gcb}} c$. Since gcb includes co, we have $d \xrightarrow{\mathsf{gcb}} c$, and therefore it must be that $e \xrightarrow{\mathsf{gcb}} c$. □

Lemma A.5. *Suppose G is (co, rf, gcb)-valid and $c \xrightarrow{\mathsf{poloc}} e$, where $\lambda(c)$ blocks $\lambda(e)$. Then $c \xrightarrow{\mathsf{gcb}} e$.*

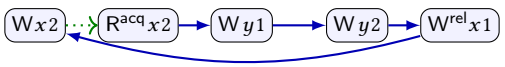
Proof. By way of contradiction, assume $e \xrightarrow{\mathsf{gcb}} c$. If $c \xrightarrow{\mathsf{rf}} e$ then by A7 we must also have $c \xrightarrow{\mathsf{gcb}} e$, contradicting the assumption that gcb is a total order. Otherwise that there is some $d \neq c$ such that $d \xrightarrow{\mathsf{rf}} e$, and therefore $d \xrightarrow{\mathsf{gcb}} e$. By transitivity, $d \xrightarrow{\mathsf{gcb}} c$. By the definition of fr, we have $e \xrightarrow{\mathsf{fr}} c$. But this contradicts A6c, since $c \xrightarrow{\mathsf{poloc}} e$. □

We show that all the order required in the pomset is also required by Arm8. M7b holds since cb_1 is consistent with co_1 and fr_1. As noted above, lob includes the order required by S3 and S6a. We need only show that the order removed from A7b can also be removed from the pomset. In order for A7b to remove order from e to c, we must have $d \xrightarrow{\mathsf{rf}} e$ and $d \xrightarrow{\mathsf{poloc}} e$ but not $d \xrightarrow{\mathsf{lob}} c$. Because of our suboptimal lowering, it must be that e is a relaxed read; otherwise the `dmb.sy` would require $d \xrightarrow{\mathsf{lob}} c$. Thus we know that S6a does not require order from e to c. By chaining R4b and W4, any dependence on the read can by satisfied without introducing order in S3. □

A.3 Lowering PwT-MCA2 to Arm

We can achieve optimal lowering for Arm by weakening the semantics of sequential composition slightly. In particular, we must lose M7c, which states that $d \xrightarrow{\mathsf{rf}} e$ implies $d < e$. Revisiting the example in the last subsection, we essentially mimic the EC characterization:

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



(cb)

Here the rf relation *contradicts* order! We have both $(\mathsf{W}x2) \dashrightarrow (\mathsf{R}^{\mathsf{acq}}x2)$ and $(\mathsf{W}x2) \xleftarrow{\mathsf{cb}} (\mathsf{R}^{\mathsf{acq}}x2)$.

We first show that EC-validity is unchanged if we assume $\mathsf{cb} \supseteq \mathsf{fr}$:

Lemma A.6. *Suppose G is EC-valid via (co, rf, cb). Then there a permutation cb′ of cb such that G is EC-valid via (co, rf, cb′) and cb′ $\supseteq$ fr, where fr is defined in A6c.*

Proof. Suppose $e \xrightarrow{\mathsf{fr}} c$. By definition of fr, $e \xleftarrow{\mathsf{rf}} d \xrightarrow{\mathsf{co}} c$, for some d. We show that either (1) $e \xrightarrow{\mathsf{cb}} c$, or (2) $c \xrightarrow{\mathsf{cb}} e$ and we can reverse the order in cb′ to satisfy the requirements.

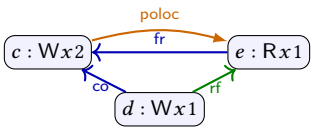
If A8a applies to $d \xrightarrow{\mathsf{rf}} e$, then $e \xrightarrow{\mathsf{cb}} c$, since it cannot be that $c \xrightarrow{\mathsf{co}} d$.

Suppose A8b applies to $d \xrightarrow{\mathsf{rf}} e$ and c is from a different thread than e. Because it is a different thread, we cannot have $e \xrightarrow{\mathsf{lob}} c$, and therefore we can choose $c \xrightarrow{\mathsf{cb}} e$ in cb′.

Suppose A8b applies to $d \xrightarrow{\mathsf{rf}} e$ and c is from the same thread as e. Applying A6c to $e \xrightarrow{\mathsf{fr}} c$, it cannot be that $c \xrightarrow{\mathsf{poloc}} e$. Since poloc is a per-thread-and-per-location total order, it must be that $e \xrightarrow{\mathsf{poloc}} c$. Applying A4a, we cannot have $e \xrightarrow{\mathsf{lob}} c$, and therefore we can choose $c \xrightarrow{\mathsf{cb}} e$ in cb′. □

Here is a contradictory non-example illustrating the last case of the proof:

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



Theorem A.7. *Suppose G_1 is EC-valid for S via $(\mathsf{co}_1, \mathsf{rf}_1, \mathsf{cb}_1)$ and that $\mathsf{cb}_1 \supseteq \mathsf{fr}_1$. Then there is a top-level pomset $P_2 \in [\![S]\!]$ such that $E_2 = E_1$, $\lambda_2 = \lambda_1$, $\mathsf{rf}_2 = \mathsf{rf}_1$, and $\leq_2 = \mathsf{cb}_1$.*

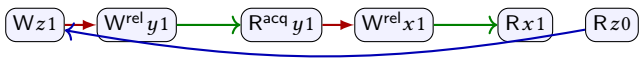
Proof. We show that all the order required in the pomset is also required by Arm8. m7b holds since cb_1 is consistent with co_1 and fr_1. As noted above, lob includes the order required by s3 and s6a′. □

B DISCUSSION

B.1 Read-Read Dependencies, If-Introduction, and Java Final Field Semantics

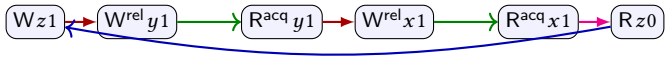
One might worry that the lack of read-read dependencies could cause drf-sc to fail. For example, the following execution has a control dependency between the reads of the last thread, but this order is not enforced, neither by our model, nor Arm8.

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



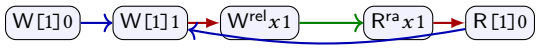
If the first read of the last thread is acquiring, then the execution is disallowed, since acquiring reads are ordered with respect to the reads that follow.

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



Arm8 enforces address dependencies between reads, but not control dependencies. To support if-introduction (aka case-analysis), we drop all dependencies between reads. This, in turn, invalidates Java's final field semantics.

$$(r := 1;\ [r] := 0;\ [r] := 1;\ x^{\mathsf{rel}} := r) \parallel (r := x^{\mathsf{acq}};\ s := [r]) \qquad \text{(ADDR2)}$$

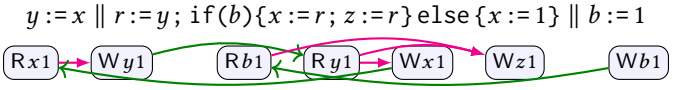


The acquire annotation is required to ensure publication. If address dependencies were enforced between reads then the acquire annotation could be dropped. However, the compiler would need to track address dependencies in order to ensure that if-introduction did not convert them to control dependencies.

B.2 Further Comparison to "Promising Semantics" [POPL 2017]

Recently, Cho et al. [2021] showed that certain combinations of compiler optimizations are inconsistent with local drf guarantees. All of the examples that prove inconsistency have the same shape: they combine read-introduction and if-introduction (aka, case analysis). Effectively, this turns one read into two, where different conditional branches can be taken for the two copies of the read. This is reminiscent of the type of *bait and switch* behavior noted by Jagadeesan et al. [2020]: the promising semantics (ps) [Kang et al. 2017] and related models [Chakraborty and Vafeiadis 2019; Jagadeesan et al. 2010; Manson et al. 2005], fail to validate compositional reasoning of temporal properties. Consider example oota4 from [Jagadeesan et al. 2020]:

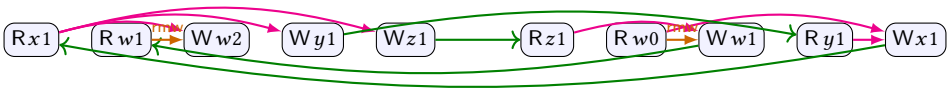
$$y := x \parallel r := y;\ \mathtt{if}(b)\{x := r;\ z := r\}\,\mathtt{else}\,\{x := 1\} \parallel b := 1 \qquad \text{(OOTA4)}$$



Under all variants of PwT, this outcome is disallowed, due to the cycle involving x and y.[10] Under PS, this outcome is allowed by baiting with the `else` branch, then switching to the `then` branch, based on a coin flip (b).

Cho et al. [2021] introduce more complex examples to show that the promising semantics fails LDRF-SC.[11] Here is one, dubbed LDRF-FAIL-PS.

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



Again, all variants of PwT disallow the outcome due to the cycle involving x and y. It is allowed by PS by baiting the second thread with $x := 1$ in the `else` branch, then switching to the `then` branch. This shows some some structural resemblance to OOTA4, with z replacing b.

Cho et al. argue that the outcome of LDRF-FAIL-PS is inevitable due to compiler optimizations. The examples crucially involve the following sequence of operations:

- read-introduction,
- if-introduction, branching on the read just introduced.

We believe this combination of optimizations is unsound. This is obviously the case in C11: read-introduction may cause undefined behavior (UB), due to the possible introduction of a data race.

The situation is more delicate in LLVM. The short version of the story is that load-hoisting followed by case analysis is unsound in LLVM, without freeze. This happens because:

- read-introduction may result in the undefined value **undef**, due to the possible introduction of a data race [Chakraborty and Vafeiadis 2017], and
- branching on an undefined value in LLVM results in UB.

LLVM delays UB using the undefined value. This allows LLVM to perform optimizations such as load hoisting, where $\texttt{if}(C)\{r := x\}$ is rewritten to $s := x\,;\, r := C?s : r$. Despite this, other optimizations regularly performed by LLVM are unsound [Lee et al. 2017]. An example is loop switching, where $\texttt{while}(C_1)\{\texttt{if}(C_2)\{S_1\}\,\texttt{else}\,\{S_2\}\}$ is rewritten to $\texttt{if}(C_2)\{\texttt{while}(C_1)\{S_1\}\}\,\texttt{else}\,\{\texttt{while}\,(C_1)\{S_2\}\}$. Freeze was introduced in LLVM in order to make such optimizations sound by allowing branch on frozen **undef** to give nondeterministic choice rather than UB. In the RFC for freeze, Lopes [2016] says: *"Note that having branch on poison not trigger* UB *has its own problems. We believe this is a good tradeoff."* LDRF-FAIL-PS demonstrates a concrete problem with this tradeoff. Other compilers, such as Compcert, are more conservative [Lee et al. 2017, §9].

Thus, the difference between PS and PwT can be understood in terms of the valid program transformations. PS allows reads to be introduced, with subsequent case analysis on the value read. PwT validates case analysis, but invalidates read-introduction.

Allowing executions such as OOTA4 and LDRF-FAIL-PS also invalidates compositional reasoning for temporal safety properties (see §5).

These differences highlight the subtle tensions between compiler optimizations and program logics that are revealed by relaxed memory models. It is not possible to have everything one wants. Thus, one is forced to choose which optimizations and reasoning principles are most important.[12]

[10] All of the reads in OOTA4 are cross-thread, so there is no difference between PwT-MCA$_1$ and PwT-MCA$_2$. For PwT-C11, there is a cycle in $\mathsf{rf} \cup <$.

[11] Cho et al. [2021] show that by restricting RMW-store reorderings, one can establish LDRF-SC for PS. We speculate that no such restriction is required for PwT. (We did not treat RMWs in our proof of LDRF-SC.)

[12] Another example is the tension between load hoisting—forbidden in C11 but allowed by LLVM—and common subexpression elimination over an acquiring lock—allowed by C11 but forbidden by LLVM [Chakraborty and Vafeiadis 2017].

Finally, we note that it is possible that PS is properly weaker than PwT.

B.3 Further Comparison to "Pomsets with Preconditions" [OOPSLA 2020]

PwT-MCA is closely related to PwP model of [Jagadeesan et al. 2020]. The major difference is that PwT-MCA supports sequential composition. In the remainder of this section, we discuss other differences. We also point out some errors in [Jagadeesan et al. 2020], all of which have been confirmed by the authors.

Substitution. PwP uses substitution rather than Skolemizing. Indeed our use of Skolemization is motivated by disjunction closure for predicate transformers, which do not appear in PwP. In Fig. 1, we gave the semantics of read for nonempty pomsets as:

(R4a) if $(E \cap D) \neq \emptyset$ then $\tau^D(\psi) \equiv v{=}r \Rightarrow \psi$,
(R4b) if $(E \cap D) = \emptyset$ then $\tau^D(\psi) \equiv (v{=}r \vee x{=}r) \Rightarrow \psi$.

In PwP, the definition is roughly as follows:

(R4a′) if $(E \cap D) \neq \emptyset$ then $\tau^D(\psi) \equiv \psi[v/r][v/x]$,
(R4b′) if $(E \cap D) = \emptyset$ then $\tau^D(\psi) \equiv \psi[v/r][v/x] \wedge \psi[x/r]$

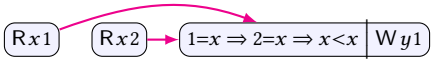
The use of conjunction in R4b′ causes disjunction closure to fail because the predicate transformer $\tau(\psi) = \psi' \wedge \psi''$ does not distribute through disjunction, even assuming that the prime operations do:[13] $\tau(\psi_1 \vee \psi_2) = (\psi_1' \vee \psi_2') \wedge (\psi_1'' \vee \psi_2'') \neq (\psi_1' \wedge \psi_1'') \vee (\psi_2' \wedge \psi_2'') = \tau(\psi_1) \vee \tau(\psi_2)$. See also §3.9.

The substitutions collapse x and r, allowing local invariant reasoning (LIR), as required by JMM causality test case 1, discussed in §3.8. Without Skolemizing it is necessary to substitute $[x/r]$, since the reverse substitution $[r/x]$ is useless when r is bound—compare with §B.8. As discussed below (B.3), including this substitution affects the interaction of LIR and downset closure.

Removing the substitution of $[x/r]$ in the independent case has a technical advantage: we no longer require *extended* expressions (which include memory references), since substitutions no longer introduce memory references.

The substitution $[x/r]$ does not work with Skolemization, even for the dependent case, since we lose the unique marker for each read. In effect, this forces all reads of a location to see the same values. Using this definition, consider the following:

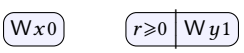
$$r := x \,;\, s := x \,;\, \texttt{if}(r{<}s)\{y := 1\}$$



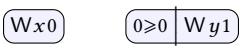
Although the execution seems reasonable, the precondition on the write is not a tautology.

Downset closure. PwP enforces downset closure in the prefixing rule. Even without this, downset closure would be different for the two semantics, due to the use of substitution in PwP. Consider the final pomset in the last example of §B.9 under the semantics of this paper, which elides the middle read event:

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



In PwP, the substitution $[x/r]$ is performed by the middle read regardless of whether it is included in the pomset, with the subsequent substitution of $[0/x]$ by the preceding write, we have $[x/r][0/x]$, which is $[0/r][0/x]$, resulting in:

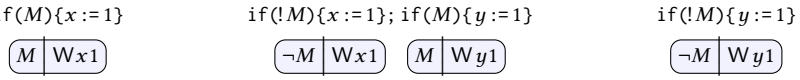


[13] $(\psi_1 \vee \psi_2)' = (\psi_1' \vee \psi_2')$ and $(\psi_1 \vee \psi_2)'' = (\psi_1'' \vee \psi_2'')$.

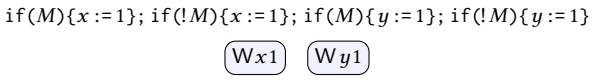
Consistency. PwP imposes *consistency*, which requires that for every pomset P, $\bigwedge_e \kappa(e)$ is satisfiable. Associativity requires that we allow pomsets with inconsistent preconditions. Consider a variant of the example from §8.3.

Associating left and right, we have:

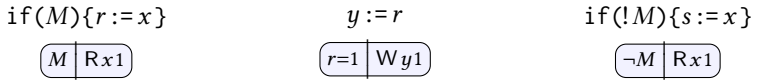
Associating into the middle, instead, we require:



Joining left and right, we have:

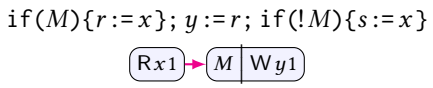


Causal Strengthening. PwP imposes *causal strengthening*, which requires for every pomset P, if $d < e$ then $\kappa(e) \vDash \kappa(d)$. Associativity requires that we allow pomsets without causal strengthening. Consider the following.



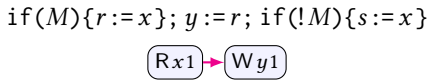
Associating left, with causal strengthening:

Finally, merging:



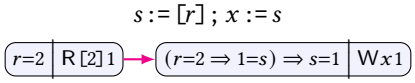
Instead, associating right:

Merging:



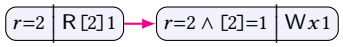
With causal strengthening, the precondition of $\mathsf{W}y1$ depends upon how we associate. This is not an issue in PwP, which always associates to the right.

One use of causal strengthening is to ensure that address dependencies do not introduce thin-air reads. Associating to the right, the intermediate state of ADDR2 (§8.4) is:



In PwP, we have, instead:

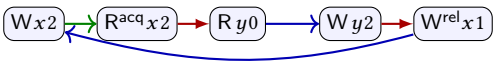
$$s := [r] \,;\, x := s$$



Without causal strengthening, the precondition of $(\mathsf{W}x1)$ would be simply [2]=1. The treatment in this paper, using implication rather than conjunction, is more precise.

Internal Acquiring Reads. The proof of compilation to Arm in PwP assumes that all internal reads can be eliminated. However, this is not the case for acquiring reads. For example, PwP disallows the following execution, where the final values of x is 2 and the final value of y is 2. This execution is allowed by Arm8 and TSO.

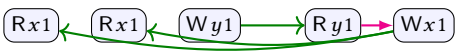
$$x := 2\,;\, r := x^{\mathsf{acq}}\,;\, s := y \parallel y := 2\,;\, x^{\mathsf{rel}} := 1$$



We discuss two approaches to this problem in §A.

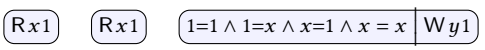
Redundant Read Elimination. Contrary to the claim, redundant read elimination fails for PwP. We discuss redundant read elimination in §8.1. Consider JMM Causality Test Case 2, which we describe there.

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



Under the semantics of PwP, we have

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



The precondition of $(\mathsf{W}y1)$ is *not* a tautology, and therefore redundant read elimination fails. (It is a tautology in $r := x\,;\, s := r\,;\, \texttt{if}(r{=}s)\{y := 1\}$.) PwP(§3.1) incorrectly stated that the precondition of $(\mathsf{W}y1)$ was $1{=}1 \wedge x{=}x$.

Termination Conditions and Parallel Composition. In PwP(§2.4), parallel composition is defined allowing coalescing of events. Here we have forbidden coalescing. This difference appears to be arbitrary. In PwP, however, there is a mistake in the handling of termination actions. The predicates should be joined using $\wedge$, not $\vee$. Here we have used termination conditions rather than termination actions so that termination is handled separately.

Read-Modify-Write Actions. In PwP, the atomicity axioms M10c erroneously applies only to overlapping writes, not overlapping reads. The difficulty can be seen in Example C.2.

In addition, PwP uses *READ* instead of *READ′* when calculating of dependency for RMWs. For a discussion, see the example at the end of §8.2.

Data Race Freedom. The definition of data race is wrong in PwP. It should require that that at least one action is relaxed.

Note that the definition of *L-stable* applies in the case that conflicting writes are totally ordered. This gives a result more in the spirit of [Dolan et al. 2018]. In particular, this special case of the theorem clarifies the discussion of the PAST example in PwP;

Augmentation of Preconditions. PwP allows arbitrary augmentation of preconditions. Here we are more conservative, only allowing augmentation of preconditions in the semantics of primitive actions, as in §8.3. As discussed in §B.10, allowing arbitrary augmentation causes associativity to fail when encoding delay logically.

B.4 Further Comparison with Sequential Predicate Transformers

We compare traditional transformers to the dependent-case transformers of Fig. 1.

All programs in our language are strongly normalizing, so we need not distinguish strong and weak correctness. In this setting, the Hoare triple $\{\phi\}\ S\ \{\psi\}$ holds exactly when $\phi \Rightarrow wp_S(\psi)$.

Hoare triples do not distinguish thread-local variables from shared variables. Thus, the assignment rule applies to all types of storage. The rules can be written as on the left below:

$$wp_{x:=M}(\psi) = \psi[M/x] \qquad \tau_{x:=M}(\psi) = \psi[M/x]$$
$$wp_{r:=M}(\psi) = \psi[M/r] \qquad \tau_{r:=M}(\psi) = \psi[M/r]$$
$$wp_{r:=x}(\psi) = x{=}r \Rightarrow \psi \qquad \tau_{r:=x}(\psi) = v{=}r \Rightarrow \psi \quad \text{where } \lambda(e) = \mathsf{R}xv$$

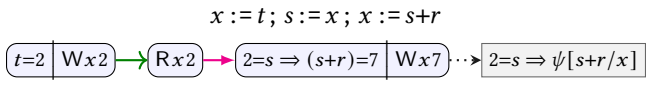
Here we have chosen an alternative formulation for the read rule, which is equivalent to the more traditional $\psi[x/r]$, as long as registers are assigned at most once in a program. Our predicate transformers for the dependent case are shown on the right above. Only the read rule differs from the traditional one.

For programs where every register is bound and every read is fulfilled, our dependent transformers are the same as the traditional ones. Thus, when comparing to weakest preconditions, let us only consider totally-ordered executions of our semantics where every read could be fulfilled by prepending some writes. For example, we ignore pomsets of $x := 2\,;\, r := x$ that read 1 for x.

For example, let S_i be defined:

$$S_1 = s := x\,;\, x := s{+}r \qquad S_2 = x := t\,;\, S_1 \qquad S_3 = t := 2\,;\, r := 5\,;\, S_2$$

The following pomset appears in the semantics of S_2. A pomset for S_3 can be derived by substituting $[2/t, 5/r]$. A pomset for S_1 can be derived by eliminating the initial write.



The predicate transformers are:

$$wp_{S_1}(\psi) = x{=}s \Rightarrow \psi[s{+}r/x] \qquad \tau_{S_1}(\psi) = 2{=}s \Rightarrow \psi[s{+}r/x]$$
$$wp_{S_2}(\psi) = t{=}s \Rightarrow \psi[s{+}r/x] \qquad \tau_{S_2}(\psi) = 2{=}s \Rightarrow \psi[s{+}r/x]$$
$$wp_{S_3}(\psi) = 2{=}s \Rightarrow \psi[s{+}5/x] \qquad \tau_{S_3}(\psi) = 2{=}s \Rightarrow \psi[s{+}5/x]$$

B.5 Register Consistency

In addition to the three criteria of Def. 3.2 Dijkstra [1975] requires

(x4′) $\tau(\mathsf{ff}) \equiv \mathsf{ff}$.

Unfortunately, our transformer for read actions (R4a) does not obey x4′, since ff is not equivalent to $v{=}r \Rightarrow \mathsf{ff}$.

In this subsection, we refine this requirement to one that does hold. The main insight is to pull values for registers from the actions of pomset itself. Thus, we define θ_λ to capture the *register state* of a pomset.

Definition B.1. Let $\theta_\lambda = \bigwedge_{\{(e,v)\in(E\times\mathcal{V})\mid\lambda(e)=(\mathsf{R}v)\}}(s_e{=}v)$ where $E = \mathsf{dom}(\lambda)$.
We say that ϕ is λ-*consistent* if $\phi \wedge \theta_\lambda$ is satisfiable. We say that it is λ-*inconsistent* otherwise.

Using this, we define the constraint on predicate transformers that we want. We also need to update the definition of predicate transformer families to carry the labeling.

Definition B.2. A *λ-predicate transformer* is a function $\tau : \Phi \to \Phi$ such that

(x1) (x2) (x3) as in Def. 3.2,

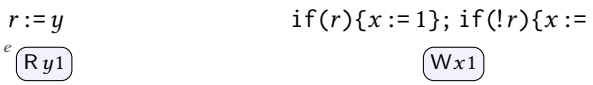
(x4) if ψ is λ-inconsistent then $\tau(\psi)$ is λ-inconsistent.

A *family of λ-predicate transformers* over consists of a λ-predicate transformer τ^D for each $D \subseteq \mathcal{E}$, such that if $C \cap E \subseteq D$ then $\tau^C(\psi) \vDash \tau^D(\psi)$.

(M4) $\tau : 2^{\mathcal{E}} \to \Phi \to \Phi$ is a *family of λ-predicate transformers*,

It would seem reasonable to require that $\kappa(e)$ be λ-consistent. However, this breaks associativity. Compare the following, where $s_e = r$:

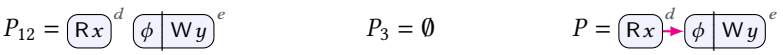
and



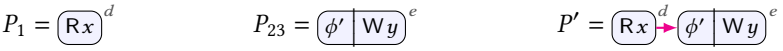
It would also seem reasonable to require that ✓ be λ-consistent in all pomsets. However, doing so is incompatible with our approach to untaken conditionals. Consider that the empty pomset is in the semantics of `if(ff){x := 1}`. In order to construct the final pomset with ✓ ≡ tt, we must allow the intermediate pomset with ✓ ≡ ff.

B.6 The Need for Respect

In Fig. 1, we choose the weakest precondition. Because of this, associativity requires that S6 is ($<$ respects $<_1$ and $<_2$) rather than ($<$ ⊇ ($<_1 \cup <_2$)). Consider ($r := x$; $y := M$; `skip`). Associating to the left, we might have:



When building P_{12}, the dependent set of e would be the empty set, and thus ϕ must have been constructed using the independent transformer R4b. Attempting to repeat this, associating to the right:

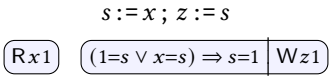


In P', however, now the dependent set of e is the singleton $\{d\}$; thus ϕ' must be constructed using the dependent transformer R4a. Since $((v{=}r \vee x{=}r) \Rightarrow \psi) \not\equiv (v{=}r \Rightarrow \psi)$, associativity fails.

If we allow stronger preconditions, as in [Jagadeesan et al. 2020], then we could use inclusion rather than *respects*. To arrive at this semantics, one would replace every occurrence of ≡ in Fig. 1 with ⊨. Then ($<$ respects $<_1$ and $<_2$) can be replaced by ($<$ ⊇ ($<_1 \cup <_2$)).

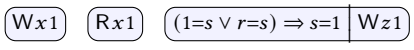
B.7 Write Substitutions

In the predicate transformer for $x := M$, we substitute M for x. In an alternative semantics, one could substitute the value chosen for the action. This alternative semantics looses dependencies. Consider:

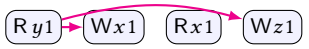


Prepending a write and then a read, our semantics gives the following:

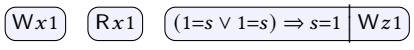
$x := r\,;\ s := x\,;\ z := s$



$r := y\,;\ x := r\,;\ s := x\,;\ z := s$



With the alternative semantics, instead, we would have:



$r := y\,;\ x := r\,;\ s := x\,;\ z := s$

The dependency from $\mathsf{R}y1$ to $\mathsf{W}z1$ has been lost.

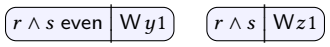
B.8 Read Substitutions

In *READ*, it is also possible to collapse x and r via substitution:

(R4a′) if $(E \cap D) \neq \emptyset$ then $\tau^D(\psi) \equiv v{=}r \Rightarrow \psi[r/x]$,
(R4b′) if $E \neq \emptyset$ and $(E \cap D) = \emptyset$ then $\tau^D(\psi) \equiv (v{=}r \vee x{=}r) \Rightarrow \psi[r/x]$,
(R4c′) if $E = \emptyset$ then $\tau^D(\psi) \equiv \psi[r/x]$,

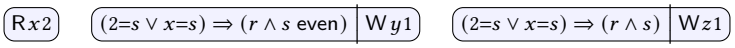
Perhaps surprisingly, this semantics is incomparable with that of Fig. 1. Consider the following:

$\texttt{if}(r \wedge s\ \text{even})\{y := 1\};\ \texttt{if}(r \wedge s)\{z := 1\}$



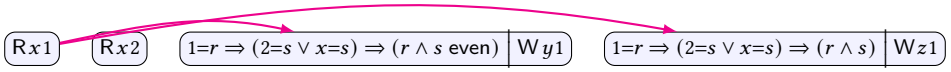
Prepending $(s := x)$, we get the same result regardless of whether we substitute $[s/x]$, since x does not occur in either precondition. Here we show the independent case:

$s := x\,;\ \texttt{if}(r \wedge s\ \text{even})\{y := 1\};\ \texttt{if}(r \wedge s)\{z := 1\}$

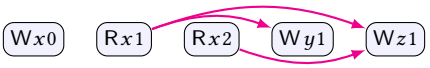


Since the preconditions mention x, prepending $(r := x)$, we now get different results depending on whether we perform the substitution. Without any substitution, we have:

$r := x\,;\ s := x\,;\ \texttt{if}(r \wedge s\ \text{even})\{y := 1\};\ \texttt{if}(r \wedge s)\{z := 1\}$

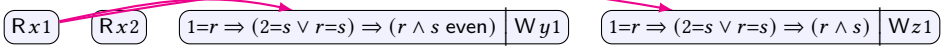


Prepending $(x := 0)$, which substitutes $[0/x]$, the precondition of $(\mathsf{W}y1)$ becomes $(1{=}r \Rightarrow (2{=}s \vee 0{=}s) \Rightarrow (r \wedge s\ \text{even}))$, which is a tautology, whereas the precondition of $\mathsf{W}z1$ becomes $(1{=}r \Rightarrow (2{=}s \vee 0{=}s) \Rightarrow (r \wedge s))$, which is not. In order to be top-level, $(\mathsf{W}z1)$ must be dependency ordered after $(\mathsf{R}x2)$; in this case the precondition becomes $(1{=}r \Rightarrow 2{=}s \Rightarrow (r \wedge s))$, which is a tautology.

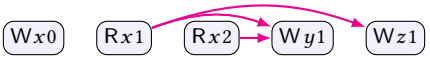


The situation reverses with the substitution $[r/x]$:

$r := x\,;\ s := x\,;\ \texttt{if}(r \wedge s\ \text{even})\{y := 1\};\ \texttt{if}(r \wedge s)\{z := 1\}$



Prepending $(x := 0)$:



The dependency has changed from $(\mathsf{R}x2) \rightarrow (\mathsf{W}z1)$ to $(\mathsf{R}x2) \rightarrow (\mathsf{W}y1)$. The resulting sets of pomsets are incomparable.

Thinking in terms of hardware, the difference is whether reads update the cache, thus clobbering preceding writes. With $[r/x]$, reads clobber the cache, whereas without the substitution, they do not. Since most caches work this way, the model with $[r/x]$ is likely preferred for modeling hardware. However, this substitution only makes sense in a model with read-read coherence and read-read dependencies, which is not the case for Arm8.

B.9 Downset Closure

We would like the semantics to be closed with respect to *downsets*. Downsets include a subset of initial events, similar to *prefixes* for strings.

Definition B.3. P_2 is an *downset* of P_1 if

(1) $E_2 \subseteq E_1$,
(2) $(\forall e \in E_2)\ \lambda_2(e) = \lambda_1(e)$,
(3) $(\forall e \in E_2)\ \kappa_2(e) \equiv \kappa_1(e)$,
(4) $(\forall e \in E_2)\ \tau_2^D(e) \equiv \tau_1^D(e)$,
(5) $\checkmark_2 \vDash \checkmark_1$,
(6a) $(\forall d \in E_2)\ (\forall e \in E_2)\ d <_2 e$ iff $d <_1 e$,
(6b) $(\forall d \in E_1)\ (\forall e \in E_2)$ if $d <_1 e$ then $d \in E_2$,
(7) $(\forall d \in E_2)\ (\forall e \in E_2)\ d\ \mathsf{rf}_2\ e$ iff $d\ \mathsf{rf}_1\ e$.

Downset closure fails due to for two reasons. The key property is that the empty set transformer should behave the same as the independent transformer.

First, downset closure fails for read-read independency §3.7. Consider

$$r := x\,;\ \texttt{if}(!r)\{s := y\}$$

$(\mathsf{R}x0)$ $(\mathsf{R}y0)$

The semantics of this program includes the singleton pomset $(\mathsf{R}x0)$, but not the singleton pomset $(\mathsf{R}y0)$. To get $(\mathsf{R}x0)$, we combine:

$r := x$ | $\texttt{if}(!r)\{s := y\}$

$(\mathsf{R}x0)$ | $\emptyset$

Attempting to get $(\mathsf{R}y0)$, we instead get:

$r := x$ | $\texttt{if}(!r)\{s := y\}$

$\emptyset$ | $r{=}0 \mid \mathsf{R}y0$

Since r appears only once in the program, this pomset cannot contribute to a top-level pomset.

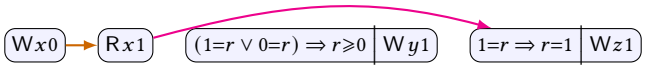
Second, the semantics is not downset closed because the independency reasoning of R4b is only applicable for pomsets where the ignored read is present! Revisiting JMM causality test case 1 from the end of §3.6:

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

$(\mathsf{W}x0)$ | $(\mathsf{R}x1)$ | $r \geqslant 0 \mid \mathsf{W}y1$ $\quad$ $r{=}1 \mid \mathsf{W}z1$

$\psi[0/x]$ | $(1{=}r \vee x{=}r) \Rightarrow \psi$

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



The precondition of $(\mathsf{W}y1)$ is a tautology.

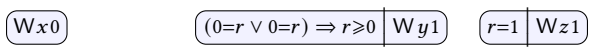
Taking the empty set for the read, however, the precondition of $(\mathsf{W}y1)$ is not a tautology:

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

$(\mathsf{W}x0)$ $\quad$ $r \geqslant 0 \mid \mathsf{W}y1$ $\quad$ $r{=}1 \mid \mathsf{W}z1$

One way to deal with the second issue would be to allow general access elimination to merge $(\mathsf{W}x0)$ and $(\mathsf{R}x0)$:

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



We leave the elaboration of this idea to future work.

B.10 Logical Encoding of Delay for PwT-MCA

In this subsection, we develop a logical encoding of delay, which can replace s6a in PwT-MCA$_1$. It is not obvious how to repeat this trick for PwT-MCA$_2$, due to thread-local reads-from (s6a′ in Def. 4.2).

As motivation, recall that we stated Lemma 3.6(g) using inclusions:

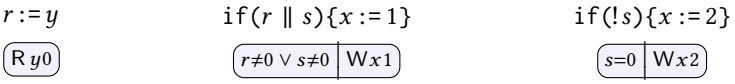
(g) $[\![\mathtt{if}(\neg\phi)\{S_2\};\ \mathtt{if}(\phi)\{S_1\}]\!] \subseteq [\![\mathtt{if}(\phi)\{S_1\}\,\mathtt{else}\,\{S_2\}]\!] \supseteq [\![\mathtt{if}(\phi)\{S_1\};\ \mathtt{if}(\neg\phi)\{S_2\}]\!]$.

PwT-MCA does not satisfy the reverse inclusion. The culprit is delay, which introduces order regardless of whether preconditions are disjoint. As an example, $[\![\mathtt{if}(r)\{x:=1\}\,\mathtt{else}\,\{x:=2\}]\!]$ has an execution with $(r{=}0 \mid \mathsf{W}x2) \to (r{\neq}0 \mid \mathsf{W}x1)$, (using augmentation), whereas $[\![\mathtt{if}(r)\{x:=1\};\ \mathtt{if}(!r)\{x:=2\}\}]\!]$ has no such execution.

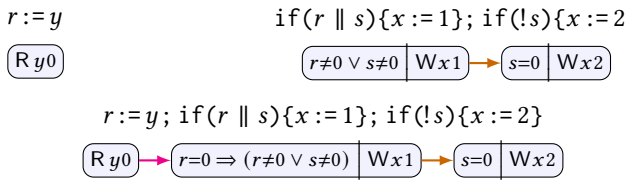
In order to validate the reverse inclusions, we could require that s6a not impose order when $\kappa_1(d) \wedge \kappa_2(e)$ is unsatisfiable. Thus, following on §B.5, we would also like this:

(s6b′) if $\lambda_1(d)$ delays $\lambda_2(e)$ and $\kappa_1(d) \wedge \kappa_2'(e)$ is λ-consistent then $d \leq e$.

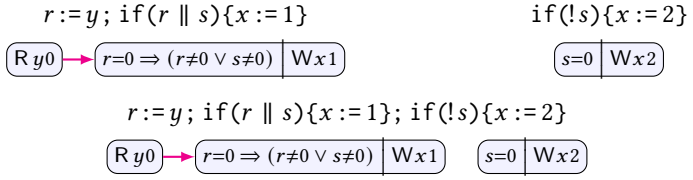
However, (s6b′) fails associativity. Example where $\theta_\lambda = (r{=}0)$



Associating right, order is required since $((r{\neq}0 \vee s{\neq}0) \wedge s{=}0)$ is satisfiable (take $r{=}1$ and $s{=}0$):



Associating left, order is not required between the writes since $(s{\neq}0 \wedge s{=}0)$ is unsatisfiable:



This motivates the logic-based presentation of delay. We make the following changes to the data model:

- actions need not include access modes—for readability, we color synchronizing events in example diagrams throughout this section,
- there exists a symbol W, indicating a write action—this is needed to handle read-read independency (§3.7),
- there exist symbols Q^{SC}, $\mathsf{Q}^{\mathsf{R}}_x$, and $\mathsf{Q}^{\mathsf{W}}_x$—we refer to these collectively as *quiescence symbols*. Roughly, the old Q_x correspond to $\mathsf{Q}^{\mathsf{W}}_x$.

We define some shorthand, using the symbols S for *stores* (aka writes) and L for *loads* (aka reads).

Definition B.4. Let $\mathsf{Q}^{\mathsf{R}}_* = \bigwedge_y \mathsf{Q}^{\mathsf{R}}_y$, and similarly for $\mathsf{Q}^{\mathsf{W}}_*$. Let $\mathsf{Q}^*_* = \mathsf{Q}^{\mathsf{R}}_* \wedge \mathsf{Q}^{\mathsf{W}}_* \wedge \mathsf{Q}^{\mathsf{SC}}$.
Let $[\phi/\mathsf{Q}^{\mathsf{R}}_*]$ substitute ϕ for every $\mathsf{Q}^{\mathsf{R}}_y$, and similarly for $\mathsf{Q}^{\mathsf{W}}_*$. Let $[\phi/\mathsf{Q}^*_*] = [\phi/\mathsf{Q}^{\mathsf{R}}_*][\phi/\mathsf{Q}^{\mathsf{W}}_*][\phi/\mathsf{Q}^{\mathsf{SC}}]$.
Let formulae $\mathsf{Q}^{\mathsf{F}\mu}$, $\mathsf{Q}^{\mathsf{S}\mu}_x$, and $\mathsf{Q}^{\mathsf{L}\mu}_x$ be defined:

$$\mathsf{Q}^{\mathsf{Frel}} = \mathsf{Q}^{\mathsf{R}}_* \wedge \mathsf{Q}^{\mathsf{W}}_* \qquad \mathsf{Q}^{\mathsf{Srlx}}_x = \mathsf{Q}^{\mathsf{R}}_x \wedge \mathsf{Q}^{\mathsf{W}}_x \qquad \mathsf{Q}^{\mathsf{Lrlx}}_x = \mathsf{Q}^{\mathsf{W}}_x$$
$$\mathsf{Q}^{\mathsf{Facq}} = \mathsf{Q}^{\mathsf{R}}_* \qquad \mathsf{Q}^{\mathsf{Srel}}_x = \mathsf{Q}^{\mathsf{R}}_* \wedge \mathsf{Q}^{\mathsf{W}}_* \qquad \mathsf{Q}^{\mathsf{Lacq}}_x = \mathsf{Q}^{\mathsf{W}}_x$$
$$\mathsf{Q}^{\mathsf{Fsc}} = \mathsf{Q}^*_* \qquad \mathsf{Q}^{\mathsf{Ssc}}_x = \mathsf{Q}^*_* \qquad \mathsf{Q}^{\mathsf{Lsc}}_x = \mathsf{Q}^{\mathsf{W}}_x \wedge \mathsf{Q}^{\mathsf{SC}}$$

Let substitutions $[\phi/\mathsf{Q}^{\mathsf{F}\mu}]$, $[\phi/\mathsf{Q}^{\mathsf{S}\mu}_x]$, and $[\phi/\mathsf{Q}^{\mathsf{L}\mu}_x]$ be defined:

$$[\phi/\mathsf{Q}^{\mathsf{Frel}}] = [\phi/\mathsf{Q}^{\mathsf{W}}_*] \qquad [\phi/\mathsf{Q}^{\mathsf{Srlx}}_x] = [\phi/\mathsf{Q}^{\mathsf{W}}_x] \qquad [\phi/\mathsf{Q}^{\mathsf{Lrlx}}_x] = [\phi/\mathsf{Q}^{\mathsf{R}}_x]$$
$$[\phi/\mathsf{Q}^{\mathsf{Facq}}] = [\phi/\mathsf{Q}^{\mathsf{R}}_*, \phi/\mathsf{Q}^{\mathsf{W}}_*] \qquad [\phi/\mathsf{Q}^{\mathsf{Srel}}_x] = [\phi/\mathsf{Q}^{\mathsf{W}}_x] \qquad [\phi/\mathsf{Q}^{\mathsf{Lacq}}_x] = [\phi/\mathsf{Q}^{\mathsf{R}}_*, \phi/\mathsf{Q}^{\mathsf{W}}_*]$$
$$[\phi/\mathsf{Q}^{\mathsf{Fsc}}] = [\phi/\mathsf{Q}^*_*] \qquad [\phi/\mathsf{Q}^{\mathsf{Ssc}}_x] = [\phi/\mathsf{Q}^{\mathsf{W}}_x, \phi/\mathsf{Q}^{\mathsf{SC}}] \qquad [\phi/\mathsf{Q}^{\mathsf{Lsc}}_x] = [\phi/\mathsf{Q}^*_*]$$

With these notations in hand, we can modify the semantics of §3 as follows. (We leave the generalization to the semantics of §8 as future work.)

Definition B.5. Update the following rules from Fig. 1.

(F3) $\kappa(e) \equiv \mathsf{Q}^{\mathsf{F}\mu}$,
(F4a) if $E \cap D \neq \emptyset$ then $\tau^D(\psi) \equiv \psi$,
(F4b) if $E \cap D = \emptyset$ then $\tau^D(\psi) \equiv \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{F}\mu}]$.
(W3) $\kappa(e) \equiv \mathsf{Q}^{\mathsf{S}\mu}_x \wedge M{=}v$,
(W4a) if $E \cap D \neq \emptyset$ then $\tau^D(\psi) \equiv \psi[M/x][(\mathsf{Q}^{\mathsf{W}}_x \wedge M{=}v)/\mathsf{Q}^{\mathsf{W}}_x]$,
(W4b) if $E \cap D = \emptyset$ then $\tau^D(\psi) \equiv \psi[M/x][\mathsf{ff}/\mathsf{Q}^{\mathsf{S}\mu}_x]$.
(R3) $\kappa(e) \equiv \mathsf{Q}^{\mathsf{L}\mu}_x$,
(R4a) if $e \in E \cap D$ then $\tau^D(\psi) \equiv (\mathsf{Q}^{\mathsf{W}}_x \Rightarrow v{=}r) \Rightarrow \psi$,
(R4b) if $e \in E \setminus D$ then $\tau^D(\psi) \equiv (\mathsf{Q}^{\mathsf{W}}_x \Rightarrow (v{=}r \vee x{=}r \vee \mathsf{W})) \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}\mu}_x]$,
(R4c) if $E = \emptyset$ then $\tau^D(\psi) \equiv \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}\mu}_x]$.

A PwT is *complete* if

(C3a) if $\lambda(e)$ is a write then $\kappa(e)[\mathsf{tt}/\mathsf{W}][\mathsf{tt}/\mathsf{Q}^*_*]$ is a tautology,
(C3b) if $\lambda(e)$ is a read then $\kappa(e)[\mathsf{ff}/\mathsf{W}][\mathsf{tt}/\mathsf{Q}^*_*]$ is a tautology,
(C5) $\checkmark[\mathsf{tt}/\mathsf{Q}^*_*]$ is a tautology.

The preconditions and the independent transformers have changed. With the exception of write, the dependent transformers are unchanged. For writes, the interpretation of $\mathsf{Q}^{\mathsf{W}}_x$ of subtly different from that of the old Q_x—the transformer strengthens $\mathsf{Q}^{\mathsf{W}}_x$ to $(\mathsf{Q}^{\mathsf{W}}_x \wedge M{=}v)$ rather than replacing it by $M{=}v$. In order to ensure coherence, we have given up on initialization.

The precondition indicates which sequentially preceding events must be ordered before. For example, all preceding accesses must be ordered before a releasing write, whereas only writes to the same location must be ordered before a acquiring read—the latter is due to coherence.

Symmetrically, the transformer indicates which sequentially following must be ordered after. For example, all following accesses must be ordered after an acquiring read, whereas only writes to the same location must be ordered after a releasing write read—again, the latter is due to coherence.

Fig. 2 shows the effect of quiescence for each access mode.

Example B.6. The definition enforces publication. Consider:

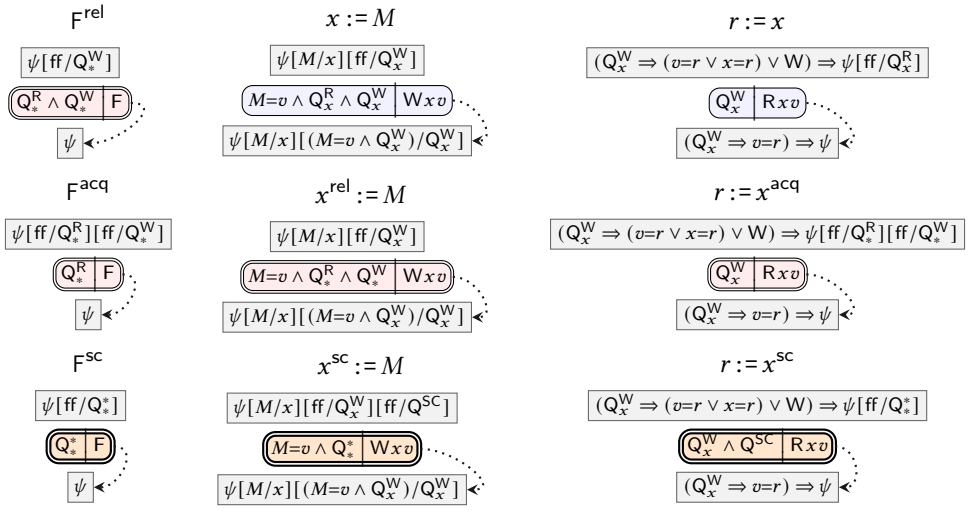
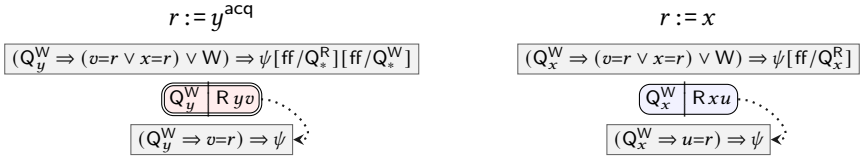


Fig. 2. The Effect of Quiescence for Each Access Mode

Since $\mathsf{Q}^{\mathsf{W}}_{*}[\mathsf{ff}/\mathsf{Q}^{\mathsf{W}}_{x}]$ is ff, we must introduce order to get a satisfiable precondition for $(\mathsf{W}yu)$.

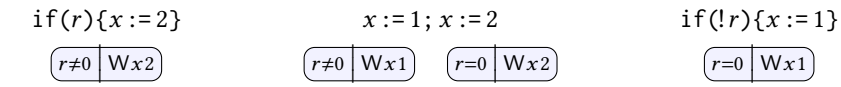
Example B.7. The definition enforces subscription. Consider:



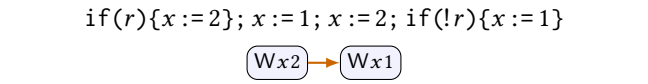
Since $\mathsf{Q}^{\mathsf{W}}_{x}[\mathsf{ff}/\mathsf{Q}^{\mathsf{W}}_{*}]$ is ff, we must introduce order to get a satisfiable precondition for $(\mathsf{R}xu)$.

Example B.8. Even in its logical form, s6b′ is incompatible with the ability to strengthen preconditions using augment closure, which is allowed in [Jagadeesan et al. 2020]. Consider the following.

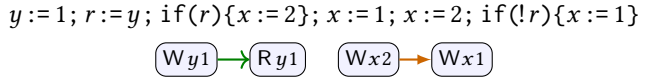
If $r{=}0$ then x is 1, 2, 1. If $r{\neq}0$ then x is 2, 1, 2. Augmenting the middle preconditions and then using sequential composition, we have:



Note that s6b′ does not require any order between the two writes of the middle pomset. Merging left and right, we have:

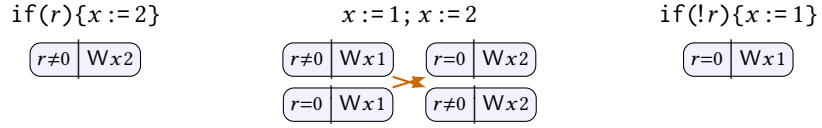


As shown by the following single-threaded code, allowing this outcome would violate DRF-SC.

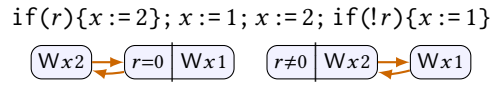


This is one reason that we use *weakest* preconditions, rather than preconditions.

The same problem does not occur due to if-introduction, since complete pomsets require that the termination condition is a tautology; therefore we cannot arbitrarily strengthen preconditions without introducing a second event to cover.



Merging left and right, we have



B.11 Optimizations Not Considered

We have not considered the following optimizations advocated by Manson et al. [2005]:

- synchronization on thread local objects can be ignored or removed altogether (the caveat to this is the fact that invocations of methods like wait and notify have to obey the correct semantics – for example, even if the lock is thread local, it must be acquired when performing a wait),
- volatile fields of thread local objects can be treated as normal fields,
- redundant synchronization (e.g., when a synchronized method is called from another synchronized method on the same object) can be ignored or removed.

Nor have we attempted to capture the following:

- read introduction,
- *monotonicity*, which allows the access mode to strength, for example from rlx to acq to sc,
- access elimination, such as store forwarding, dead-write-removal, redundant write after read elimination [Sevčík and Aspinall 2008, §4.1].

One approach to elimination would be to allow merging of actions with different labels. A list of safe merges can be found in [Chakraborty and Vafeiadis 2017, §E] and [Kang 2019, §7.1]. For examples of unsafe merges and reorderings, see [Chakraborty and Vafeiadis 2017, §D]. See also [Chakraborty and Vafeiadis 2019, §6.2]

Certain combinations of optimizations are quite delicate. For example, consider if-introduction and dead-write-removal. With if-introduction, the following equation should hold:

$$\begin{aligned}&[\![\texttt{if}(r)\{x := 2\};\ x := 1;\ x := 2;\ \texttt{if}(!r)\{x := 1\};\ x := 3]\!]\\ =\ &[\![\texttt{if}(!r)\{x := 1\};\ x := 2;\ x := 1;\ \texttt{if}(r)\{x := 2\};\ x := 3]\!]\end{aligned}$$

Using dead write removal naively, these could be refined, respectively, to:

$$\begin{aligned}&[\![x := 1;\ x := 2;\ x := 3]\!]\\ \stackrel{?}{=}\ &[\![x := 2;\ x := 1;\ x := 3]\!]\end{aligned}$$

Depending upon the details of the model, these may be observably different.

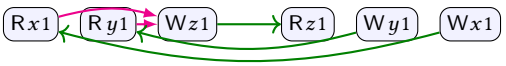
What has become of coherence?

B.12 The State of the Art Circa 2021

Pugh [1999] noticed that the semantics of Java 1.0 disabled common subexpression elimination. This lead to a repaired model five years later [Manson et al. 2005]. Shortly thereafter, Cenciarelli

et al. [2007, §7] noticed that the repaired model disabled the reordering of independent statements. Here is the example:

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



Quoting Cenciarelli et al. [2007, §7]:

> After reordering the independent statements in the else branch, a compiler may execute assignments $x := 1$ and $y := 1$ early, so that [the execution is allowed]. However, such a behaviour is not legal according to the current JMM, as it violates the condition that the happens-before orders during validation be consistent with the final happens-before on the committed actions. In fact, the latter will have the write to x before the write to y, but during validation the write to y happens before the write to x.

Since then, several models have been proposed. Many have been revised repeatedly to repair bugs. (For example, this paper fixes several errors of Jagadeesan et al. [2020].)

In this subsection, we provide series of quotations from a discussion on the OpenJDK mailing lists, which provides an excellent summary of the state of the art in 2021, when this paper was written. (The quotes are ordered for readability, with hyperlinks to the original discussion.)

> Raffaello Giulietti: "JEP 188: Java Memory Model Update" [1], the JMM wiki [2] and the jmm-dev mailing list [3] seem quite inactive. (The latter point explains why I'm posting to this list instead.)
>
> The introduction of `j.l.i.VarHandle` [4] brought more access modes to Java, but in a narrative and informal way. A paper by Bender & Palsberg [5], addressing the formalization of the concurrent access modes, has been published in 2019 but I'm not sure if it caught the attention of the OpenJDK community.
>
> So what is the current thinking for progressing the JMM spec?
>
> Hans Boehm: I think it's safe to say that it has been slow going, not just for Java, but for other languages as well.
>
> In my view, the core problem, shared by pretty much all of them, is that we don't have an established way to give well-defined semantics to potentially racing unordered accesses, like ordinary variable accesses in Java, or `memory_order_relaxed` accesses in C and C++. That's particularly essential with the traditional Java language-based-security model, since we can't just give up on racing accesses to ordinary variables.
>
> I'm aware of a number of proposed solutions. But I don't think we currently have enough confidence that they
>
> (a) Are correct, and don't have issues similar to the older models,
> (b) Don't have unintended consequences, particularly for compilation, and
> (c) Are sufficiently comprehensible by programmers to actually be useful.
>
> (a) is hard because the models have gotten complex enough that reviewers are scarce. (A problem that I gather you're familiar with.) The authors are commonly experts at formally analyzing the models, but it's hard to analyze whether the model conflicts with some well-known, but perhaps not well-written-down compilation technique.
>
> Probably even more controversially, I think we've realized that existing compiler technology can compile such racing code in ways that some of us are not 100% sure should really be allowed. Demonstrably unexecuted code can affect

the semantics in ways that strike me as scary. (See https://wg21.link/p1217 for a down-to-assembly C++ version; [if I understand correctly], Lochbihler and others earlier came up with some closely related observations for Java.)

It might be possible to do what we've involuntarily done for C++: Punt the hard cases for now, and define what the model is for programs without racing ordinary accesses.

Andrew Haley:

> (Quoting Hans Boehm) Probably even more controversially, I think we've realized that existing compiler technology can compile such racing code in ways that some of us are not 100% sure should really be allowed.

This implies, does it not, that the problem is not formalization as such, but that we don't really understand what the language is supposed to mean? That's always been my problem with OOTA: I'm unsure whether the problem is due to the inadequacy of formal models, in which case the formalists can fix their own problem, or something we all have to pay attention to.

Hans Boehm: In some sense, I'm not sure either. The p1217 examples [formalized below as RFUB and RFUB-NC] bother me in that they seem to violate some global programming rules ("if x is only ever null or refers to an object properly constructed by the same thread, then x should never appear to refer to an incompletely constructed object"). And there seems to be disagreement about whether the currently allowed behavior is "correct."

On the other hand, in practice the weirdness doesn't seem to break things. If you ask people advocating the current behavior, the answer will be that it doesn't matter because nobody writes code that way. If you ask people trying to analyzer or verify code, they'll probably be unhappy. And I haven't been able to convince myself that you cannot get yourself into these situations just by linking components together, each of which does something perfectly reasonable.

And there are very common code patterns (like the standard implementation of reentrant locks used by all Java implementations) that break if you allow general OOTA behavior. Which at least means that you can't currently formally verify such code. The theorem you'd be trying to prove is false with respect to the part of the language spec we know how to formalize.

It's a mess.

Andrew Haley:

> (Quoting Hans Boehm) Demonstrably unexecuted code can affect the semantics in ways that strike me as scary. (See wg21.link/p1217 for a down-to-assembly C++ version; [if I understand correctly], Lochbihler and others earlier came up with some closely related observations for Java.)

Looking again at p1217, it seems to me that enforcing load-store ordering would have severe effects on compilers, at least without new optimization techniques. We hoist loads before loops and sink stores after them. When it all works out, there are no memory accesses in the loop. A load-store barrier in a loop would have the effect of forcing succeeding stores out to memory, and forcing preceding loads to reload from memory. It's not hard to imagine that this would cause an order-of-margnitude performance reduction in common cases.

I suppose one could argue that such optimizations would continue to be valid, so only those stores which would have been emitted anyway would be affected.

But that's not how compilers work, as far as I know. In our IR for C2, memory accesses are not pinned in any way, so the only way to make unrelated accesses execute in any particular order is to add a dependency between all loads and stores.

Hans Boehm: I think it would be a fairly pervasive change to optimizers. It has also become clear in WG21, the C++ committee, that there is not enough support for requiring this. In that case, Ou and Demsky have a paper saying that the overhead is likely to be on the order of 1% or less. For Java if it were applied everywhere, it would probably be appreciably higher.

On the other hand, it's a bit harder than that to come up with examples where the generated x86 code has to be worse. Moving loads earlier in the code, or delaying stores, as you suggest, would still be fine. The only issue is with delaying loads past stores, which seems less common, though it can certainly be beneficial for reducing live ranges, probably some vectorization etc.

But it seems unlikely that such a restriction will be applied even to C++ `memory_order_relaxed`, much less Java ordinary variables.

Doug Lea: My stance in the less formal account (http://gee.cs.oswego.edu/dl/html/j9mm.html) as well as Shuyang Liu et al's ongoing formalization (see links from http://compilers.cs.ucla.edu/people/) is that the most you want to say about racy Java programs is that they are typesafe. As in: you can't see a String when expecting an int. Even this looser constraint is challenging to specify, prove, and extend. But it is a path for Java that might not apply to languages like C that are not guaranteed typesafe anyway, and so enter Undefined Behavior territory (as opposed to possibly-unexpected but still typesafe behavior).

Han Boehm: But this now breaks some common idioms, right? In particular, I think a bunch of code assumes that racing assignments of equivalent primitive values or immutable objects to the same field are OK.

If, in 2004, our view of language-based security had been the same as it is now, then I completely agree that this would have been the right approach. But I think doing it now would require significant user code changes. Which might still be the best way forward ...

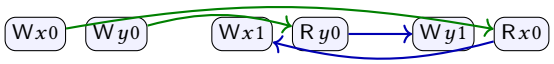
C ADDITIONAL EXAMPLES (PwT-MCA)

This appendix includes additional examples. They all apply equally to PwT-MCA$_1$ and PwT-MCA$_2$. Many of these are taken directly from [Jagadeesan et al. 2020]; see there for further discussion.

C.1 Buffering

Store buffering is allowed, as required by TSO.

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



Load buffering is allowed, as required by Arm8.

$$r := y;\ x := 1 \parallel r := x;\ y := 1 \tag{LB}$$

C.2 Thin-Air

Thin air is disallowed. [Pugh 2004, TC4]:

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

(OOTA1)

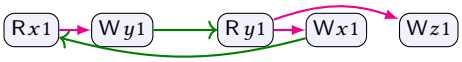
The control variant ([Pugh 2004, TC13]) is also disallowed:

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

(OOTA2)

[Jagadeesan et al. 2020, §2]

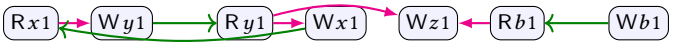
$$y := x \parallel r := y\,;\, \texttt{if}(r)\{x := r\,;\, z := r\}\,\texttt{else}\,\{x := 2\}$$



(OOTA3)

[Jeffrey and Riely 2019, §8] and [Jagadeesan et al. 2020, §6]:

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



(OOTA4)

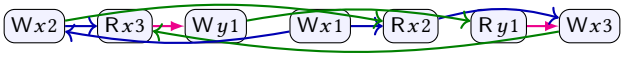
[Svendsen et al. 2018, RNG] is disallowed since there is no write to fulfill $(\mathsf{R}y1)$.

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

(OOTA6)

OOTA7 is allowed by PS, but not WEAKESTMO [Chakraborty and Vafeiadis 2019, Fig. 3]:

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



(OOTA7)

OOTA4 is similar to TC5 [Pugh 2004]:

$$y := x \parallel x := y \parallel z := 0\,;\, z := 1 \parallel x := z$$

(TC5)

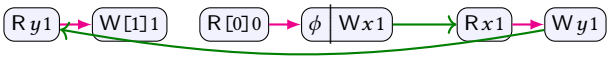
The justification for forbidding this execution states:

> values are not allowed to come out of thin air, even if there are other executions in which the thin-air value would have been written to that variable by some not out-of-thin-air means.

OOTA4 is an interesting border case, since it is allowed by speculative models (§B.2).

We presented a thin-air behavior involving address calculation in §8.4. TC12 provides another example—eliding initializing writes, all 0:

$$r := y\,;\, [r] := 1\,;\, s := [0]\,;\, x :=\! ! s \parallel y := x$$



(TC12)

Building the precondition ϕ from right to left:

$\phi_1 \equiv s{=}0$ ($x :=\! ! s$)

$\phi_2 \equiv (\mathsf{Q}_{[0]} \Rightarrow 0{=}s) \Rightarrow s{=}0$ (Prepending $s := [0]$)

$\phi_3 \equiv (r{=}1 \Rightarrow \phi_2[1/[1]][\mathsf{tt}/\mathsf{Q}_{[1]}]) \wedge (r{=}0 \Rightarrow \phi_2[1/[0]][\mathsf{ff}/\mathsf{Q}_{[0]}])$ (Prepending `if`)

$\equiv (r{=}1 \Rightarrow (\mathsf{Q}_{[0]} \Rightarrow 0{=}s) \Rightarrow s{=}0) \wedge (r{=}0 \Rightarrow s{=}0)$

Dependent case:

$$\phi_4 \equiv (\mathsf{Q}_y \Rightarrow 1{=}r) \Rightarrow \phi_3 \qquad \text{(Prepending } r := y\text{)}$$
$$\phi_5 \equiv 1{=}r \Rightarrow (r{=}1 \Rightarrow (0{=}s \Rightarrow s{=}0)) \wedge (r{=}0 \Rightarrow s{=}0) \qquad \text{(Prepending Initializers)}$$

Independent case:

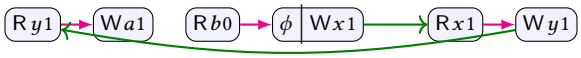
$$\phi'_4 \equiv (\mathsf{Q}_y \Rightarrow 1{=}r \vee y{=}r) \Rightarrow \phi_3 \qquad \text{(Prepending } r := y\text{)}$$
$$\phi'_5 \equiv (1{=}r \vee 0{=}r) \Rightarrow (r{=}1 \Rightarrow (0{=}s \Rightarrow s{=}0)) \wedge (r{=}0 \Rightarrow s{=}0) \qquad \text{(Prepending Initializers)}$$

The justification for forbidding TC12 states:

> Since no other thread accesses [either [0] or [1]], the code for thread one should be equivalent to:
>
> $$r := y\,;\ [r] := 1\,;\ \texttt{if}(r{=}0)\{s := 1\}\,\texttt{else}\,\{s := 0\}\,;\ x := !s$$
>
> With this code, it is clear that this is the same situation as [OOTA1].

Here is the same example with control dependencies—again eliding initializing writes, all 0:

$$r := y\,;\ \texttt{if}(r)\{a := 1\}\,\texttt{else}\,\{b := 1\}\,;\ s := b\,;\ x := !s \parallel y := x \qquad \text{(TC12}')$$



Building the precondition ϕ from right to left:

$$\phi_1 \equiv s{=}0 \qquad (x := !s)$$
$$\phi_2 \equiv (\mathsf{Q}_b \Rightarrow 0{=}s) \Rightarrow s{=}0 \qquad \text{(Prepending } s := b\text{)}$$
$$\phi_3 \equiv (r{\neq}0 \wedge \phi_2[1/a][\mathtt{tt}/\mathsf{Q}_a]) \vee (r{=}0 \wedge \phi_2[1/b][\mathtt{ff}/\mathsf{Q}_b]) \qquad \text{(Prepending if)}$$
$$\equiv (r{\neq}0 \wedge ((\mathsf{Q}_b \Rightarrow 0{=}s) \Rightarrow s{=}0)) \vee (r{=}0 \wedge s{=}0)$$

Dependent case:

$$\phi_4 \equiv (\mathsf{Q}_y \Rightarrow 1{=}r) \Rightarrow \phi_3 \qquad \text{(Prepending } r := y\text{)}$$
$$\phi_5 \equiv 1{=}r \Rightarrow (r{\neq}0 \wedge (0{=}s \Rightarrow s{=}0)) \vee (r{=}0 \wedge s{=}0) \qquad \text{(Prepending Initializers)}$$

Independent case:

$$\phi'_4 \equiv (\mathsf{Q}_y \Rightarrow 1{=}r \vee y{=}r) \Rightarrow \phi_3 \qquad \text{(Prepending } r := y\text{)}$$
$$\phi'_5 \equiv (1{=}r \vee 0{=}r) \Rightarrow (r{\neq}0 \wedge (0{=}s \Rightarrow s{=}0)) \vee (r{=}0 \wedge s{=}0) \qquad \text{(Prepending Initializers)}$$

Jagadeesan et al. [2020, §6] provide the following analysis of RFUB:

> Boehm's [2019] RFUB example presents another potential form of OOTA behavior. Our analysis shows that there is no OOTA behavior in RFUB, only a false dependency:
>
> $$[\![r := y\,;\ x := r]\!] \not\supseteq [\![r := y\,;\ \texttt{if}(r{\neq}1)\{z := 1\,;\ r := 1\}\,;\ x := r]\!] \qquad \text{(RFUB)}$$
>
> The left command is half of OOTA3 ($y := x$). The right command is dubbed RFUB, for *Register assignment From an Unexecuted Branch*. Boehm observes that in the context $x := y \parallel [-]$, these programs have different behaviors. Yet the OOTA example on the left never writes 1. Why should the unexecuted branch change that? Because of the conditional, the write to x in RFUB is independent of the read from y. It useful to considering the Hoare logic formulas satisfied by the two threads above: we have $\{\mathtt{tt}\}$ RFUB $\{x = 1\}$ for the right thread of RFUB, but not $\{\mathtt{tt}\}$ OOTA3 $\{x = 1\}$ for the right thread of OOTA3. The change in the thread from OOTA3 to RFUB is not a valid refinement under Hoare logic; thus, it is expected that RFUB may have additional behaviors.

RFUB New Constructor:

$$y := x \parallel r := y;\ \texttt{if}(r{=}\texttt{null})\{r := \texttt{new C}()\};\ x := r;\ r.\texttt{f}() \qquad \text{(RFUB-NC)}$$

This is similar to:

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

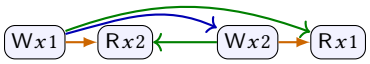
And different from the following, which is similar to TC18:

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

C.3 Coherence

The following execution is disallowed by fulfillment (M7a and M7b). It is also disallowed by C11 and Java.

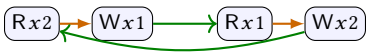
$$x := 1;\ r := x \parallel x := 2;\ s := x \qquad \text{(COH)}$$



M7b requires that we order one write with respect to the other, either before the write or after the read (and therefore after the write). Suppose we pick 1 before 2, as shown. This satisfies M7b for $(\mathsf{R}x2)$. But to satisfy the requirement for $(\mathsf{R}x1)$ we must have either $(\mathsf{W}x2) < (\mathsf{W}x1)$ or $(\mathsf{R}x1) < (\mathsf{W}x2)$. Either way, we have a cycle.

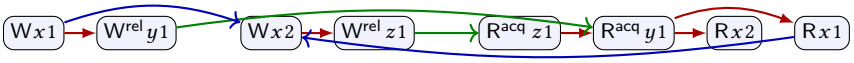
Our model is more coherent than Java, which permits the following:

$$r := x;\ x := 1 \parallel s := x;\ x := 2 \qquad \text{(TC16)}$$



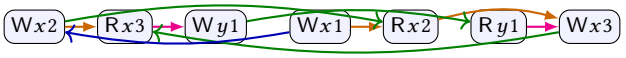
We also forbid the following, which Java allows:

$$x := 1;\ y^{\mathsf{rel}} := 1 \parallel x := 2;\ z^{\mathsf{rel}} := 1 \parallel r := z^{\mathsf{acq}};\ r := y^{\mathsf{acq}};\ r := x;\ r := x \qquad \text{(CO3)}$$



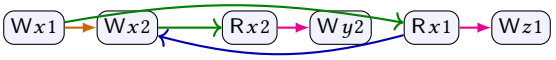
The following outcome is allowed by the promising semantics [Kang et al. 2017], but not in WEAKESTMO [Chakraborty and Vafeiadis 2019, Fig. 3]. We disallow it:

$$x := 2;\ \texttt{if}(x{\neq}2)\{y := 1\} \parallel x := 1;\ r := x;\ \texttt{if}(y)\{x := 3\} \qquad \text{(COH-CYC)}$$



C11 includes read-read coherence between relaxed atomics in order to forbid the following. We do not order reads by intra-thread coherence, and this allow the following:

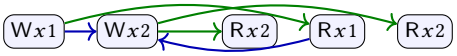
$$x := 1;\ x := 2 \parallel y := x;\ z := x \qquad \text{(CO2)}$$



Here, the reader sees 2 then 1, although they are written in the reverse order.

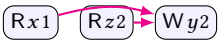
We also allow the following, similar execution:

$$x := 1;\ x := 2 \parallel r_1 := x;\ r_2 := x;\ r_3 := x;$$



Pugh [1999, §2.3] presented the following example to show that Java's original memory model required alias analysis to validate common subexpression elimination (CSE).

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



Coalescing the two read of x is obviously allowed if $z \neq x$. But if $z{=}x$, coalescing is only permitted because we do not include read-read pairs in $\bowtie_{\mathsf{co}}$ (§3.2):

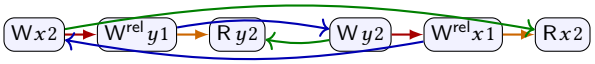
$$\bowtie_{\mathsf{co}} = \{(\mathsf{W}x, \mathsf{W}x),\ (\mathsf{R}x, \mathsf{W}x),\ (\mathsf{W}x, \mathsf{R}x)\}$$

C11 has read-read coherence, and therefore CSE is only valid up to alias analysis in C11.

C.4 RA

Our model is closer to strong RA (SRA) [Lahav and Boker 2020; Lahav et al. 2016], than RA, as in C11 and RC11. For example, RC11 allows the following, which we disallow:

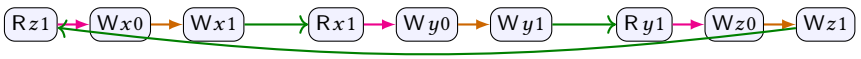
$$x := 2\,;\ y^{\mathsf{rel}} := 1\,;\ r := y \parallel y := 2\,;\ x^{\mathsf{rel}} := 1\,;\ s := x \tag{SRA}$$



C.5 MCA

Here are a few litmus tests that distinguish MCA architectures from non-MCA architectures. MCA1 is an example of *write subsumption* [Pulte et al. 2018, §3]:

$$\texttt{if}(z)\{x := 0\};\ x := 1 \parallel \texttt{if}(x)\{y := 0\};\ y := 1 \parallel \texttt{if}(y)\{z := 0\};\ z := 1 \tag{MCA1}$$

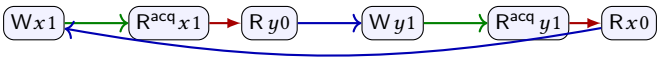


Two thread variant:

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

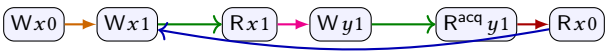
IRIW is allowed if all accesses are relaxed, but not if the initial reads are acquiring:

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



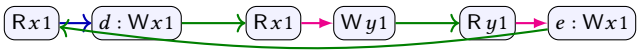
MCA2 is a simplified version of IRIW

$$x := 0\,;\ x := 1 \parallel y := x \parallel r := y^{\mathsf{acq}}\,;\ s := x \tag{MCA2}$$



[Flur et al. 2016] and [Lahav and Vafeiadis 2016, Fig. 4] discuss the following, which is not valid in Arm8, although it was valid under some earlier sketches of the model:

$$r := x\,;\ x := 1 \parallel y := x \parallel x := y \tag{MCA3}$$

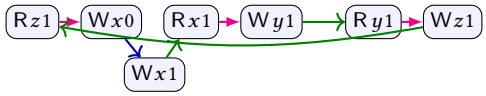


These candidate executions are invalid, due to cycles.

C.6 Detour

The following example [Podkopaev et al. 2019, Ex. 3.7] is disallowed by IMM by including a detour relation. It is also disallowed by PS.

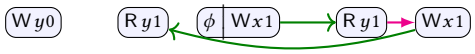
$$x := z-1;\ y := x \parallel x := 1 \parallel z := y$$



C.7 Local Invariant Reasoning and Value Range Analysis

We have already seen TC1 in §3.8, TC2 in §8.1 and TC6 in §6. Here is the complete program for TC6:

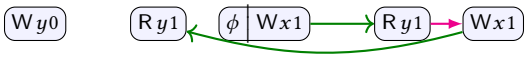
$$y := 0;\ (r := y;\ \texttt{if}(r{=}0)\{x := 1\};\ \texttt{if}(r{=}1)\{x := 1\}) \parallel (\texttt{if}(x{=}1)\{y := 1\}) \tag{TC6}$$



$$\phi = (1{=}r \vee 0{=}r) \Rightarrow (r{=}0 \vee r{=}1)$$

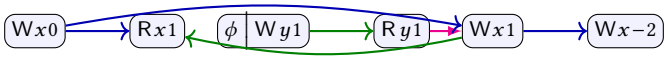
Here are some additional examples from [Jagadeesan et al. 2020]:

$$y := 0;\ (r := y;\ x := 1{+}r{*}r{-}r) \parallel (y := x) \tag{TC8}$$



$$\phi = (1{=}r \vee 0{=}r) \Rightarrow 1{+}r{*}r{-}r = 1$$

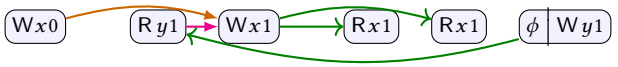
$$x := 0;\ (r := x;\ \texttt{if}(r{\geqslant}0)\{y := 1\} \parallel x := y \parallel x := -2) \tag{TC9}$$



$$\phi = (1{=}r \vee 0{=}r) \Rightarrow r{\geqslant}0$$

Java Causality Test Case 18 asks that we justify the following execution:

$$x := 0;\ (x := y \parallel r := x;\ \texttt{if}(r{=}0)\{x := 1\};\ s := x;\ y := s) \tag{TC18}$$



Before we prefix $x := 0$, the precondition of $\mathsf{W}y1$ is:

$$\phi \equiv (1{=}r \vee x{=}r) \Rightarrow ([r{=}0 \wedge ((1{=}s \vee 1{=}s) \Rightarrow s{=}1)] \vee [r{\neq}0 \wedge ((1{=}s \vee x{=}s) \Rightarrow s{=}1)])$$

Simplifying:

$$\phi \equiv (1{=}r \vee x{=}r) \Rightarrow (r{=}0 \vee [r{\neq}0 \wedge ((1{=}s \vee x{=}s) \Rightarrow s{=}1)])$$

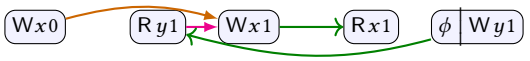
Prefixing $x := 0$:

$$\phi \equiv (1{=}r \vee 0{=}r) \Rightarrow (r{=}0 \vee [r{\neq}0 \wedge ((1{=}s \vee 0{=}s) \Rightarrow s{=}1)])$$

Drilling into the interesting part:

$$\phi \equiv 1{=}r \Rightarrow ((1{=}s \vee 0{=}s) \Rightarrow s{=}1)$$

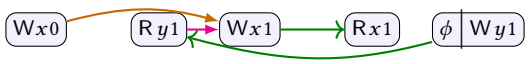
This is not a tautology. But we get one by coalescing s and r:



$$\phi \equiv 1{=}r \Rightarrow ((1{=}r \vee 0{=}r) \Rightarrow r{=}1)$$

TC20 splits the first thread of TC18:

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



(TC20)

Because we take register state from the right, the example is the same as for TC18 above.

TC17 replaces the condition $r{=}0$ by $r{\neq}1$ in TC18:

$$\phi \equiv (1{=}r \vee x{=}r) \Rightarrow ([r{\neq}1 \wedge ((1{=}s \vee 1{=}s) \Rightarrow s{=}1)] \vee [r{=}1 \wedge ((1{=}s \vee x{=}s) \Rightarrow s{=}1)])$$

Simplifying and prefixing $x := 0$:

$$\phi \equiv (1{=}r \vee 0{=}r) \Rightarrow (r{\neq}1 \vee [r{=}1 \wedge ((1{=}s \vee 0{=}s) \Rightarrow s{=}1)])$$

Again, we have:

$$\phi \equiv 1{=}r \Rightarrow ((1{=}s \vee 0{=}s) \Rightarrow s{=}1)$$

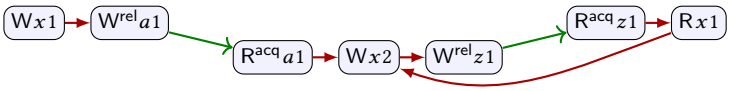
which is not a tautology. But we get one by coalescing s and r.

TC19 makes the same change for TC20, and follows for the same reason.

C.8 Release/Acquire and Internal Reads

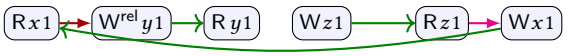
From [Jagadeesan et al. 2020]:

$$x := 1;\ a^{\mathsf{rel}} := 1;\ \texttt{if}(z^{\mathsf{acq}})\{r := x\} \parallel \texttt{if}(a^{\mathsf{acq}})\{x := 2;\ z^{\mathsf{rel}} := 1\}$$



(INTERNAL1)

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

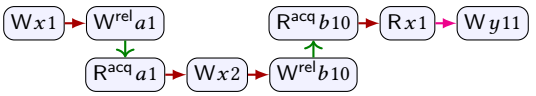


(INTERNAL2)

C.9 Roach Motel: Commuting Release and Acquire

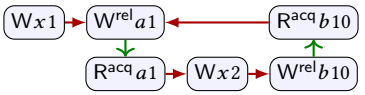
The following is impossible, since $\mathsf{R}x1$ unfulfilled.

$$x := 1;\ a^{\mathsf{rel}} := 1;\ r := b^{\mathsf{acq}};\ s := x;\ y := r{+}s \parallel r := a^{\mathsf{acq}};\ x := 2;\ b^{\mathsf{rel}} := 10$$



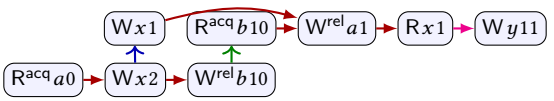
If you swap the release and acquire, then it is impossible for the second thread to get in the middle.

$$x := 1;\ r := b^{\mathsf{acq}};\ a^{\mathsf{rel}} := 1;\ \parallel r := a^{\mathsf{acq}};\ x := 2;\ b^{\mathsf{rel}} := 10$$



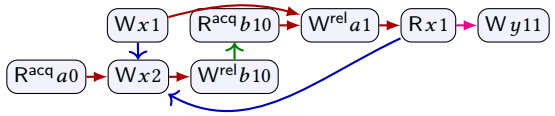
In this case, the following execution is possible:

$$x := 1;\ r := b^{\mathsf{acq}};\ a^{\mathsf{rel}} := 1;\ s := x;\ y := r{+}s \parallel r := a^{\mathsf{acq}};\ x := 2;\ b^{\mathsf{rel}} := 10$$



But not:

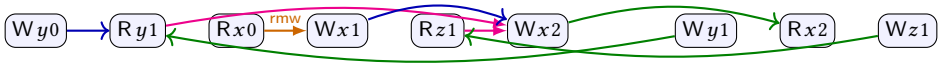
$$x := 1;\ r := b^{\mathsf{acq}};\ a^{\mathsf{rel}} := 1;\ s := x;\ y := r{+}s \ \|\ r := a^{\mathsf{acq}};\ x := 2;\ b^{\mathsf{rel}} := 10$$



C.10 RMWs

If RMWs simply use the same semantics as read and write, then we allow LDRF-PF-FAIL, which is used to show failure of LDRF-SC for the promising semantics in [Cho et al. 2021].

$$y := 0;\ \mathtt{if}(y)\{\mathtt{if}(!\mathsf{CAS}(x,0,1))\{\mathtt{if}(z)\{x := 2\}\}\} \ \|\ y := 1;\ \mathtt{if}(1{\neq}\mathsf{CAS}(x,0,3))\{z := 1\}$$

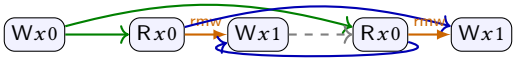


(LDRF-PF-FAIL)

To disallow this, we need to retain the dependency $(\mathsf{R}x2) \to (\mathsf{W}z1)$. For this, we need to avoid the substitution for x. This is why we use *READ′* instead of *READ* in the independent case for RMWs.

It is not possible for two RMWs to see the same write.

$$x := 0;\ \left(\mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x,1) \ \|\ \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x,1)\right)$$

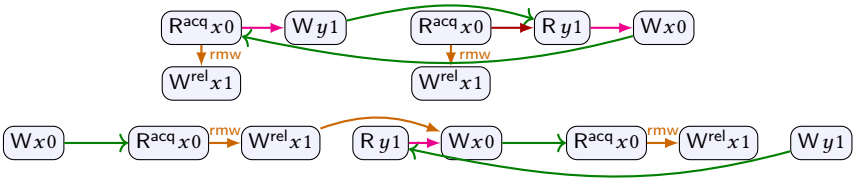


(RMW0)

The gray arrow is required the RMW atomicity axioms.

Lee et al. [2020] introduce PS2.0 to refine the treatment of RMWs in the promising semantics (PS). Their examples have the expected results here, with far less work. First they recall that PS requires quantification over multiple futures in order to disallow executions such as CDRF. (We showed the relaxed variant (CDRF-RLX) in §8.2.)

$$r := \mathsf{FADD}^{\mathsf{acq},\mathsf{rel}}(x,1);\ \mathtt{if}(r{=}0)\{y := 1\} \ \|\ r := \mathsf{FADD}^{\mathsf{acq},\mathsf{rel}}(x,1);\ \mathtt{if}(r{=}0)\{\mathtt{if}(y)\{x := 0\}\}$$

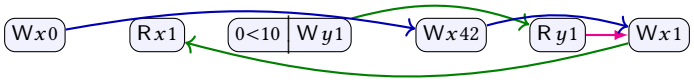


(CDRF)

This execution is clearly impossible, due to the cycle above. In this diagram, we have not drawn order adjacent to the writes of the RMWs, since this is not necessary to produce the cycle. If CDRF is allowed then DRF-RA fails.

PS does not support global value range analysis, as modeled by GA+E below. Our semantics permits GA+E:

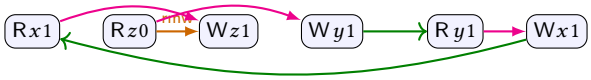
$$x := 0;\ \left(r := \mathsf{CAS}^{\mathsf{rlx},\mathsf{rlx}}(x,0,1);\ \mathtt{if}(r{<}10)\{y := 1\} \ \|\ x := 42;\ x := y\right)$$



(GA+E)

PS also does not support register promotion, as modeled by RP below. Our semantics permits RP. It is allowed by Arm8 and by WEAKESTMO.

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

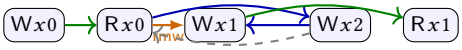


(RP)

Example C.1. Recall M10c: if $\lambda(c)$ overlaps $\lambda(d)$ and $d \xrightarrow{\mathsf{rmw}} e$ then (1) $c < e$ implies $c \le d$ and (2) $d < c$ implies $e \le c$.

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

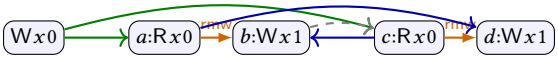
$$x := 0 \,;\, \mathsf{INC}^{\mathsf{rlx},\mathsf{rlx}}(x) \;\|\; x := 2 \,;\, r := x$$



By 1, since $(\mathsf{W}x2) \longrightarrow (\mathsf{W}x1)$, it must be that $(\mathsf{W}x2) \longrightarrow (\mathsf{R}x0)$, creating a cycle.

Example C.2. Two successful RMWs cannot see the same write:

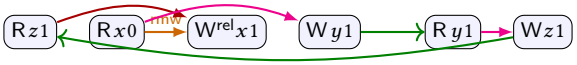
$$x := 0 \,;\, (\mathsf{INC}^{\mathsf{rlx},\mathsf{rlx}}(x) \;\|\; \mathsf{INC}^{\mathsf{rlx},\mathsf{rlx}}(x))$$



The order from read-to-write is required by fulfillment. Apply 1 of the second RMW to $a \longrightarrow d$, we have that $a \longrightarrow c$. Subsequently applying 2 of the first RMW, we have $b \longrightarrow c$, creating a cycle.

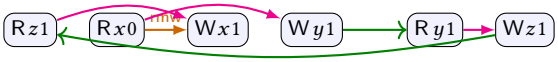
Example C.3. By using two actions rather than one, the definition allows examples such as the following, which is allowed by Arm8 [Podkopaev et al. 2019, Ex. 3.10]:

$$r := z \,;\, s := \mathsf{INC}^{\mathsf{rlx},\mathsf{rel}}(x) \,;\, y := s{+}1 \;\|\; r := y \,;\, z := r$$



A similar example, also allowed by Arm8 [Chakraborty and Vafeiadis 2019, Fig. 6]:

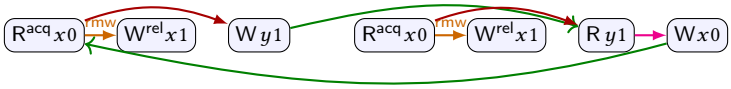
$$r := z \,;\, s := \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x, r) \,;\, y := s{+}1 \;\|\; r := y \,;\, z := r$$



This is allowed by WEAKESTMO, but not PS.

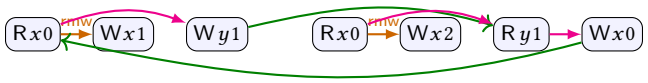
Example C.4. Consider the CDRF example from [Lee et al. 2020]:

$$r := \mathsf{INC}^{\mathsf{acq},\mathsf{rel}}(x) \,;\, \mathtt{if}(r{=}0)\{y := 1\}$$
$$\|\;\; r := \mathsf{INC}^{\mathsf{acq},\mathsf{rel}}(x) \,;\, \mathtt{if}(r{=}0)\{\mathtt{if}(y)\{x := 0\}\}$$



Example C.5. Consider this example from [Lee et al. 2020, §C]:

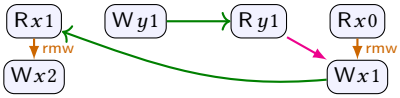
$$r := \mathsf{CAS}^{\mathsf{rlx},\mathsf{rlx}}(x,0,1)\,;\ \mathtt{if}(r{\leqslant}1)\{y := 1\}$$
$$\parallel\ r := \mathsf{CAS}^{\mathsf{rlx},\mathsf{rlx}}(x,0,2)\,;\ \mathtt{if}(r{=}0)\{\mathtt{if}(y)\{x := 0\}\}$$



The following examples are from [Cho et al. 2021].

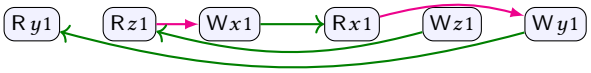
CDRF shows that PwT semantics is not too permissive for rel/acq-RMWs. But what about rlx-RMWs. The following execution is allowed by Arm8, and PS2.0, but disallowed by PS2.1.

$$r := \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x,1)\,;\ y := 1 \parallel r := y\,;\ s := \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x,r) \qquad \text{(RMW-W)}$$



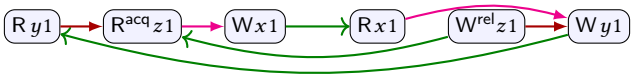
If this $\{z\}$-DRF-RA?

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



(NAIVE-LDRF-RA-FAIL)

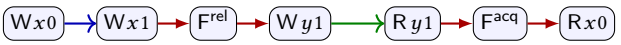
Interpreting $\{z\}$ as rel/acq:



C.11 Fences

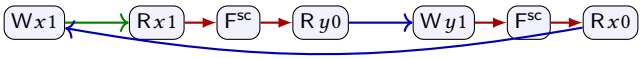
From [Jagadeesan et al. 2020]:

$$x := 0\,;\ x := 1\,;\ \mathsf{F}^{\mathsf{rel}}\,;\ y := 1 \parallel r := y\,;\ \mathsf{F}^{\mathsf{acq}}\,;\ s := x \qquad \text{(PUB2)}$$



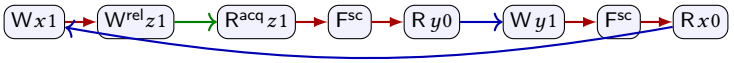
[Lahav et al. 2017, Fig. 5]:

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



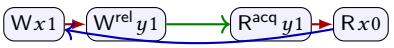
[Lahav et al. 2017, Fig. 6]

$$x := 1\,;\ z^{\mathsf{rel}} := 1\,;\ \parallel r := z^{\mathsf{acq}}\,;\ \mathsf{F}^{\mathsf{sc}}\,;\ r := y \parallel y := 1\,;\ \mathsf{F}^{\mathsf{sc}}\,;\ r := x \qquad \text{(SC4)}$$

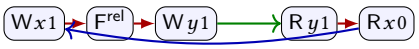


Here are several examples mixing fencing with release/acquire:

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



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



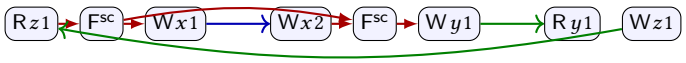
$x := 1; y^{\mathsf{rel}} := 1 \parallel r := y; \mathsf{F}^{\mathsf{acq}}; s := x$

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

[Podkopaev et al. 2019, §D]:

> The following execution graph is not consistent in the promise-free declarative model of [Kang et al. 2017]. Nevertheless, its mapping to POWER (obtained by simply replacing Fsc with Fsync) is POWER-consistent and po ∪ rf is acyclic (so it is Strong-POWER-consistent). Note that, using promises, the promising semantics allows this behavior.

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



[Podkopaev et al. 2019, §8]:

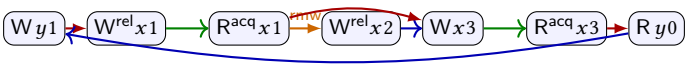
> To establish the correctness of compilation of the promising semantics to POWER, Kang et al. [2017] followed the approach of Lahav and Vafeiadis [2016]. This approach reduces compilation correctness to POWER to (i) the correctness of compilation to the POWER model strengthened with po ∪ rf acyclicity; and (ii) the soundness of local reorderings of memory accesses. To establish (i), Kang et al. [2017] wrongly argued that the strengthened POWER-consistency of mapped promise-free execution graphs imply the promise-free consistency of the source execution graphs. This is not the case due to SC fences, which have relatively strong semantics in the promise-free declarative model (see [Podkopaev et al. 2018, Appendix D] for a counter example). Nevertheless, our proof shows that the compilation claim of Kang et al. [2017] is correct.

C.12 Fences and RMW

Aim: allow the splitting of release writes and RMWs into release fences followed by relaxed operations. [Podkopaev et al. 2019, Remark 2, After example 3.1]:

> In RC11 [Lahav et al. 2017], as well as in C/C++11 [Batty et al. 2011], this rather intuitive transformation, as we found out, is actually unsound.

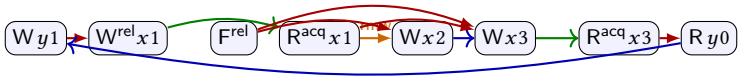
$y := 1; x^{\mathsf{rel}} := 1 \parallel \mathtt{INC}^{\mathsf{acq,rel}}(x); x := 3 \parallel r := x^{\mathsf{acq}}; s := y$



> (R)C11 disallows the annotated behavior, due in particular to the release sequence formed from the release exclusive write to x in the second thread to its subsequent relaxed write. However, if we split the increment to fencerel; a := FADDacq,rlx(x, 1) (which intuitively may seem stronger), the release sequence will no longer exist, and

the annotated behavior will be allowed. IMM overcomes this problem by strengthening sw in a way that ensures a synchronization edge for the transformed program as well

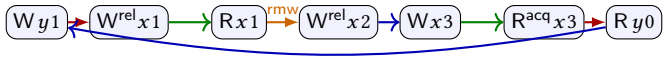
$$y := 1;\ x^{\mathsf{rel}} := 1 \parallel \mathsf{F}^{\mathsf{rel}};\ \mathsf{INC}^{\mathsf{acq,rlx}}(x);\ x := 3 \parallel r := x^{\mathsf{acq}};\ s := y$$



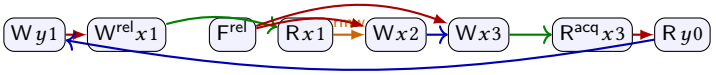
We seem to disallow both of these out of the box.

In the case of a relaxed read in the RMW, the outcome is allowed in both cases:

$$y := 1;\ x^{\mathsf{rel}} := 1 \parallel \mathsf{INC}^{\mathsf{rlx,rel}}(x);\ x := 3 \parallel r := x^{\mathsf{acq}};\ s := y$$



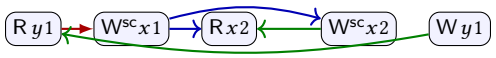
$$y := 1;\ x^{\mathsf{rel}} := 1 \parallel \mathsf{F}^{\mathsf{rel}};\ \mathsf{INC}^{\mathsf{rlx,rlx}}(x);\ x := 3 \parallel r := x^{\mathsf{acq}};\ s := y$$



C.13 SC Access and Volatiles

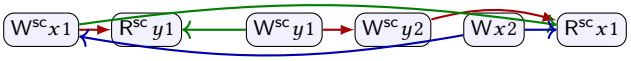
[Dolan et al. 2018, §8.2]:

$$r := y;\ x^{\mathsf{sc}} := 1;\ s := x \parallel x^{\mathsf{sc}} := 2;\ y := 1 \tag{sc1}$$



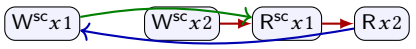
Watt et al. [2020, §3.1]:

$$x^{\mathsf{sc}} := 1;\ r := y^{\mathsf{sc}} \parallel y^{\mathsf{sc}} := 1;\ y^{\mathsf{sc}} := 2;\ x := 2;\ s := x^{\mathsf{sc}} \tag{sc2}$$



Violation of SC-DRF from [Watt et al. 2020, Fig. 9]. The following program is DRF. It should not be possible for the second thread to read 1 then 2

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



(Additionally, fulfillment of the read of 1 requires that $\mathsf{W}^{\mathsf{sc}}x2 \longrightarrow \mathsf{W}^{\mathsf{sc}}x1$, which we have elided.)

The following example is from https://bugs.openjdk.java.net/browse/JDK-8262877: One implementation strategy for volatiles maps a volatile read to a full fence followed by acquire and a volatile write to a release followed by full fence. On power, this is not enough to guarantee that all-volatile programs only have SC executions. This implementation strategy on Power allows the following execution, which is disallowed by our semantics.

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

D PROOF SKETCH: LDRF-SC FOR PwT-MCA

In this appendix, we sketch a proof of DRF-SC for PwT-MCA$_2$. We prove an *external* result, where the notion of *data-race* is independent of the semantics itself. Since every PwT-MCA$_2$ is also a PwT-MCA$_1$, the result also applies there. Our result is also *local*. Using Dolan et al.'s [2018] notion of *Local Data Race Freedom (LDRF)*.

We do not address PwT-C11. The internal DRF-SC result for C11 [Batty 2015] does not rely on dependencies and thus applies to PwT-C11. In internal DRF-SC, data-races are defined using the semantics of the language itself. Using the notion of dependency defined here, it should be possible to prove an stronger external result for C11, similar to that of [Lahav et al. 2017]—we leave this as future work.

Jagadeesan et al. [2020] prove LDRF-SC for Pomsets with Preconditions (PwP). PwT-MCA generalizes PwP to account for sequential composition. Most of the machinery of LDRF-SC, however, has little to do with sequential semantics. Thus, we have borrowed heavily from the text of [Jagadeesan et al. 2020]; indeed, we have copied directly from the LaTeX source, which is publicly available. We indicate substantial changes or additions using a change-bar on the right.

There are several changes:

- PwP imposes several conditions that we have dropped: *consistency*, *causal strengthening*, *downset closure* (see §B.3).
- PwP allows preconditions that are stronger than the weakest precondition.
- PwP imposes M7c (rf implies <) and thus is similar to PwT-MCA$_1$. PwT-MCA$_2$ is a weaker model that is new to this paper.
- PwP did not provide an accurate account of program order for merged actions. We use Lemma 6.2 to correct this deficiency.

The first two items require us to define gen differently, below.

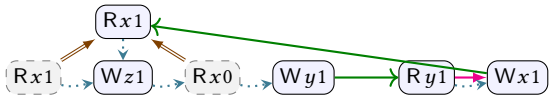
The result requires that locations are properly initialized. We assume a sufficient condition: that programs have the form "$x_1 := v_1\,;\ \cdots x_n := v_n\,;\ S$" where every location mentioned in S is some x_i. To simplify the definition of *happens-before*, we ban fences and RMWs.

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

D.1 Definitions

Program Order. Let $[\![\cdot]\!]^{\mathsf{po}}_{\mathsf{mca2}}$ be defined by applying the construction of Lemma 6.2 to $[\![\cdot]\!]_{\mathsf{mca2}}$. We consider only *complete* pomsets. For these, we derive program order on compound events as follows. By Lemma 6.4, if there is a compound event e, then there is a phantom event $c \in \pi^{-1}(e)$ such that $\kappa(c)$ is a tautology. If there is exactly one tautology, we identify e with c in program order. If there is more than one tautology, Lemma D.1, below, shows that it suffices to pick an arbitrary one—we identify e with the $c \in \pi^{-1}(e)$ that is minimal in program order. For example, consider JMM causality test case 2, with an added write to z:

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



Data Race. Data races are defined using *program* order (po), not *pomset* order (<).

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

Let $\mathsf{hb} = (\mathsf{po} \cup \mathsf{sw})^+$ be the *happens-before* relation.

Let $L \subseteq \mathcal{X}$ be a set of locations. We say that d *has an L-race with* e (notation $d \overset{L}{\rightsquigarrow} e$) when (1) at least one is relaxed, (2) at least one is a write, (3) they access the same location in L, and (4) they are unordered by hb: neither $d \xrightarrow{\mathsf{hb}} e$ nor $e \xrightarrow{\mathsf{hb}} d$.

Generators. We say that $P' \in \nabla(\mathcal{P})$ if there is some $P \in \mathcal{P}$ such that P is *complete* (Def. 4.1) and P' is a *downset* of P (Def. B.3).

Let P be *augmentation-minimal* in $\mathcal{P}$ if $P \in \mathcal{P}$ and there is no $P \neq P' \in \mathcal{P}$ such that P augments P'.

Let $\mathsf{gen}[\![S]\!] = \{P \in \nabla[\![S]\!]^{\mathsf{po}}_{\mathsf{mca2}} \mid P \text{ is augmentation-minimal in } \nabla[\![S]\!]^{\mathsf{po}}_{\mathsf{mca2}}\}$.

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

Similarity. We say that P' *is e-similar to* P if they differ at most in (1) pomset order adjacent to e, (2) the value associated with event e, if it is a read, and (3) the addition and removal of read events po-after e.

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

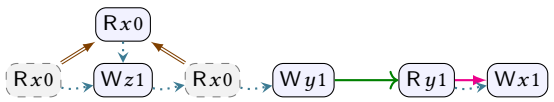
Sequentiality. Let $\lessdot_L = <_L \cup \mathsf{po}$, where $<_L$ is the restriction of $<$ to events that access locations in L. We say that P' is *L-sequential after* P if (1) P' is po-convex, (2) $\lessdot_L$ is acyclic in $E' \setminus E$.

Simplicity. We say that P' is *L-simple after* P if all of the events in $E' \setminus E$ that access locations in L are *simple* (Def. 6.1).

Lemma D.1. *Suppose $P' \in gen[\![S]\!]$ and P is L-sequential after P. Let P'' be the restriction of P' that is L-simple after P (throwing out compound L-events after P). Then $P'' \in gen[\![S]\!]$.*

As a negative example, note that (‡‡) is not L-sequential—in fact there is no execution of the program that results in the simple events of (‡‡): without merging the reads, there would be a dependency $(\mathsf{R}x1) \rightarrow (\mathsf{W}y1)$. L-sequential executions of this code must read 0 for x:

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



D.2 Theorem and Proof Sketch

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

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

Proof Sketch. We show L-sequentiality. L-simplicity then follows from Lemma D.1.

In order to develop a technique to find P''' from P'', we analyze pomset order in generation-minimal top-level pomsets. First, we note that $<_*$ (the transitive reduction $<$) can be decomposed into three disjoint relations. Let $\mathsf{ppo} = (<_* \cap \mathsf{po})$ denote *preserved* program order, as required by sequential composition and conditional. The other two relations are cross-thread subsets of $(<_* \setminus$

po): rfe (reads-from-external) orders writes before reads, satisfying P6a and P6b; cae (coherence-after-external) orders read and write accesses before writes, satisfying M7b. (Within a thread, S6a′ induces order that is included in ppo.)

Using this decomposition, we can show the following.

Lemma D.3. *Suppose $P'' \in gen[\![S]\!]$ has an external read $d \xrightarrow{\mathsf{rf}''} e$ that is maximal in (ppo ∪ rfe). Further suppose that there another write d' that could fulfill e. Then there exists an e-similar P''' with $d' \xrightarrow{\mathsf{rf}'''} e$ such that $P''' \in gen[\![S]\!]$.*

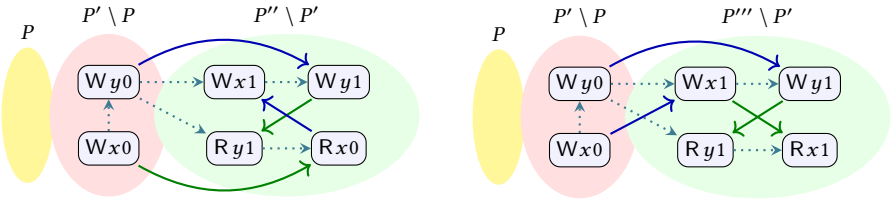
The proof of the lemma follows an inductive construction of gen$[\![S]\!]$, starting from a large set with little order, and pruning the set as order is added: We begin with all pomsets generated by the semantics without imposing the requirements of fulfillment (including only ppo). We then prune reads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

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

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

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

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

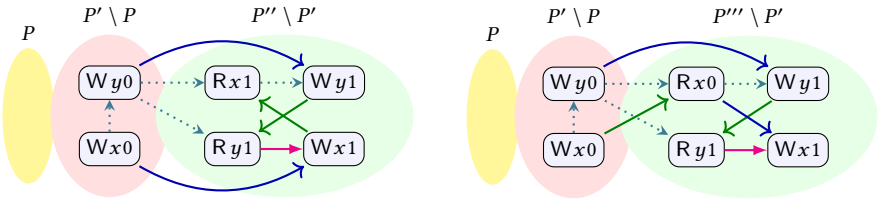


It is important that e be (po ∪ rfe)-maximal, not just (ppo ∪ rfe)-maximal. The latter criterion would allow us to choose e to be the read of y, but then there would be no e-similar pomset: if an execution reads 0 for y then there is no read of x, due to the conditional.

In the above argument, it is unimportant whether e reads-from an internal or an external write; thus the argument applies to PwT-MCA$_2$ and PwT-MCA$_1$ as it does for PwT-MCA$_1$.

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

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

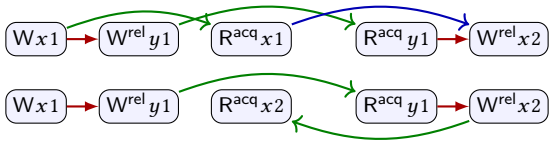


This example requires $(\mathsf{W}x0)$. Proper initialization ensures the existence of such "older" writes. □

D.3 A Note on Prior Work

In preparing this paper, we came across the following example, which appears to invalidate Theorem 4.1 of [Dongol et al. 2019].

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



The program is data-race free. The two executions shown are the only top-level executions that include $(\mathsf{W}^{\mathsf{rel}}x2)$.

Theorem 4.1 of [Dongol et al. 2019] is stated by extending execution sequences. In the terminology of [Dongol et al. 2019], a read is *L-weak* if it is sequentially stale. Let $\rho = (\mathsf{W}x1)(\mathsf{W}^{\mathsf{rel}}y1)(\mathsf{R}^{\mathsf{acq}}y1)(\mathsf{W}^{\mathsf{rel}}x2)$ be a sequence and $\alpha = (\mathsf{R}^{\mathsf{acq}}x1)$. ρ is L-sequential and α is L-weak in $\rho\alpha$. But there is no execution of this program that includes a data race, contradicting the theorem. The error seems to be in Lemma A.4 of [Dongol et al. 2019], which states that if α is L-weak after an L-sequential ρ, then α must be in a data race. That is clearly false here, since $(\mathsf{R}^{\mathsf{acq}}x1)$ is stale, but the program is data race free.

In proving the SC-LDRF result in [Jagadeesan et al. 2020, §8], we noted that our proof technique is more robust than that of [Dongol et al. 2019], because it limits the prefixes that must be considered. In (¶), the induction hypothesis requires that we add $(\mathsf{R}^{\mathsf{acq}}x1)$ before $(\mathsf{W}^{\mathsf{rel}}x2)$ since $(\mathsf{R}^{\mathsf{acq}}x1) \longrightarrow (\mathsf{W}^{\mathsf{rel}}x2)$. In particular,

is not a downset of (¶), because $(\mathsf{R}^{\mathsf{acq}}x1) \longrightarrow (\mathsf{W}^{\mathsf{rel}}x2)$. As noted in [Jagadeesan et al. 2020, §8], this affects the inductive order in which we move across pomsets, but does not affect the set of pomsets that are considered. In particular,

is a downset of (¶).