The Leaky Semicolon

Compositional Semantic Dependencies for Relaxed-Memory Concurrency

ANONYMOUS AUTHOR(S)

Program logics and semantics tell us that when executing $(S_1; S_2)$ starting in state s_0, we execute S_1 in s_0 to arrive at s_1, then execute S_2 in s_1 to arrive at the final state s_2. This is, of course, an abstraction. Processors execute instructions out of order, due to pipelines and caches, and compilers reorder programs even more dramatically. All of this reordering is meant to be unobservable in single-threaded code, but is observable in multi-threaded code. A formal attempt to understand the resulting mess is known as a "relaxed memory model." The relaxed memory models that have been proposed to date either fail to address sequential composition directly, 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 propose using preconditions and families of predicate transformers. When composing $(S_1; S_2)$, the predicate transformers used to validate the preconditions of events in S_2 are chosen based on the semantic dependencies from events in S_1 to events in S_2. We apply this approach to two existing memory models: "Modular Relaxed Dependencies" for C11 and "Pomsets with Preconditions."

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:
Anonymous Author(s). 2021. The Leaky Semicolon: Compositional Semantic Dependencies for Relaxed-Memory Concurrency. *Proc. ACM Program. Lang.* 0, OOPSLA, Article 0 (October 2021), 46 pages.

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 [Marino et al. 2015]. When these approaches are not available, however, we are left with an enormous gap in our understanding of one of the most basic elements of computing: the humble semicolon. Until recently, existing approaches either

- did not bother tracking dependencies, allowing "thin air" executions — as in C and C++ [Batty et al. 2015],
- tracked dependencies conservatively, using syntax, requiring inefficient implementation of relaxed access [Boehm and Demsky 2014; Kavanagh and Brookes 2018; Lahav et al. 2017;

2475-1421/2021/10-ART0
https://doi.org/

Vafeiadis and Narayan 2013]— a non-starter for safe languages like Java, and an unacceptable cost for low-level languages like C,
- computed dependencies using non-compositional operational models over alternate worlds [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 validate many compiler optimizations, but also fail to validate temporal safety properties (see §A.2).

Recently, two denotational models have been proposed that compute sequential dependencies semantically. Paviotti et al. [2020] defined Modular Relaxed Dependencies (MRD-C11), which use event structures to calculate dependencies for C11. Jagadeesan et al. [2020] defined Pomsets with Preconditions (PwP), which use preconditions and logic to calculate dependencies for a Java-like language on multicopy-atomic (MCA) hardware, such as ARM-V8 [Pulte et al. 2018]. However, neither paper treated sequential composition as a first-class citizen. MRD-C11 encoded sequential composition using continuation-passing, and PwP used prefixing, adding one event at a time on the left. In both cases, the approaches require perfect knowledge of the future.

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. The definition is associative: the denotation of $((S_1\,;\,S_2)\,;\,S_3)$ is the same as the denotation of $(S_1\,;\,(S_2\,;\,S_3))$. It also validates expected laws concerning the interaction of sequencing and conditional execution.

To manage complexity, we have layered the definitions. After an overview, we define sequential dependencies in §4. The next two sections add concurrency. In §5, we define PwT-MCA, which provides a Java-like model for MCA hardware, similar to that of Jagadeesan et al. [2020]. In §6, we summarize the results for this model. In §7, we define PwT-C11, which models C11, adapting the approach of Paviotti et al. [2020]. In §8 we provide a tool for automatic evaluation of litmus tests. In §9, we extend the semantics to include additional features, such as address calculation.

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

Introductory 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*).

(D1) $wp_{\texttt{skip}}(\psi) = \psi$
(D2) $wp_{r:=M}(\psi) = \psi[M/r]$
(D3) $wp_{S_1;S_2}(\psi) = wp_{S_1}(wp_{S_2}(\psi))$
(D4) $wp_{\texttt{if}(M)\{S_1\}\texttt{else}\{S_2\}}(\psi) = ((M{\neq}0) \Rightarrow wp_{S_1}(\psi)) \wedge ((M{=}0) \Rightarrow wp_{S_2}(\psi))$

For this language, 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 D2 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

(D2b) $wp_{x:=M}(\psi) = \psi[M/x]$ (D2c) $wp_{r:=x}(\psi) = \psi[x/r]$

is unsound! In particular, a read from a shared memory location may be fulfilled by a write in another thread, invalidating D2c. (We assume that expressions do *not* include shared variables.)

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

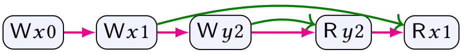
(1) it should not impose too much order, overconstraining the implementation,
(2) it should not impose too little order, allowing bogus executions, and
(3) it should be *compositional* and *direct*, as described in §1.

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

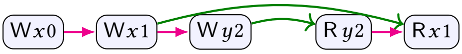
For single-threaded programs, memory can be thought of as you might expect: programs write to, and read from, memory references. This can be thought of as a total order of reads and writes (pink arrows ⟶), where each read has a matching *fulfilling* write (green arrows ⟶), for example:

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



This model naturally extends 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 ∥):

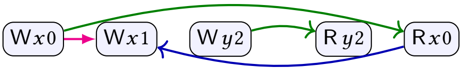
$$x := 0;\ x := 1;\ y := 2 \parallel r := y;\ s := x$$



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.

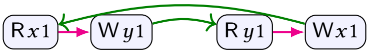
This has led to *relaxed memory models*, in which the requirement of sequential consistency is weakened to only apply *per-location* and not globally over the whole program. This allows executions that are inconsistent with program order, such as:

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



In such models, the causal order between events is important, and includes control and data dependencies, to avoid paradoxical "out of thin air" examples such as:

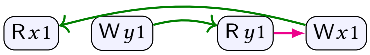
$$r := x;\ \texttt{if}(r)\{y := 1\} \parallel s := y;\ x := s$$



This candidate execution forms a cycle in causal order, so is disallowed, but this depends crucially on the control dependency from (R x 1) to (W y 1), and the data dependency from (R y 1) to (W x 1).

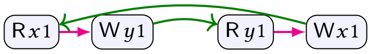
If either is missing, then this execution is acyclic and hence allowed. For example dropping the control dependency results in:

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



While syntactic dependency calculation suffices for hardware models, it is not preserved by common compiler optimizations. For example, if we calculate control dependencies syntactically, then there is a dependency from (Rx1) to (Wy1), and therefore a cycle in, the candidate execution:

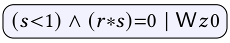
$$r := x ;\ \mathtt{if}(r)\{y := 1\}\,\mathtt{else}\,\{y := 1\} \parallel s := y ;\ x := s$$



A compiler may lift the assignment $y := 1$ out of the conditional, thus removing the dependency.

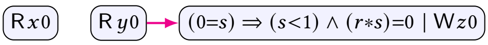
To address this, Jagadeesan et al. [2020] introduced *Pomsets with Preconditions (*PwP*)*, where events are labeled with logical formulae. Nontrivial preconditions are introduced by store actions (modeling data dependencies) and conditionals (modeling control dependencies):

$$\mathtt{if}(s<1)\{z := r*s\}$$

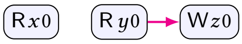


Preconditions are discharged by being ordered after a read (we assume the usual precedence for logical operators—¬, ∧, ∨, ⇒):

$$r := x ;\ s := y ;\ \mathtt{if}(s<1)\{z := r*s\} \tag{$\dagger$}$$



Note that there is dependency order from (Ry0) to (Wz0) so the precondition for (Wz0) only has to be satisfied assuming the hypothesis $(0=s)$. There is no matching order from (Rx0) to (Wz0) which is why we do not assume the hypothesis $(0=r)$. Nonetheless, the precondition on (Wz0) is a tautology, and so can be elided in the diagram:



2.3 Predicate Transformers For Relaxed Memory

Pomsets with Preconditions show how the logical approach to sequential dependency calculation can be mixed into a relaxed memory model. However, Jagadeesan et al. do not provide a model of sequential composition. Instead, their model uses *prefixing*, which requires that the model is built from right to left: events are prepended one at a time, with perfect knowledge of the future. This makes reasoning about sequential program fragments difficult. For example, Jagadeesan et al. state the equivalence allowing reordering independent writes as follows,

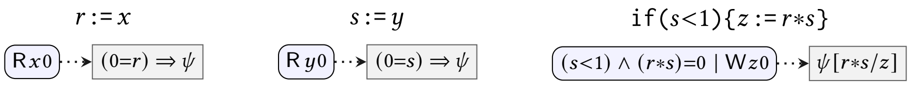
$$[\![x := M ;\ y := N ;\ S]\!] = [\![y := N ;\ x := M ;\ S]\!] \quad \text{if } x \neq y$$

where S is the entire future computation! By formalizing sequential composition, we can show:

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

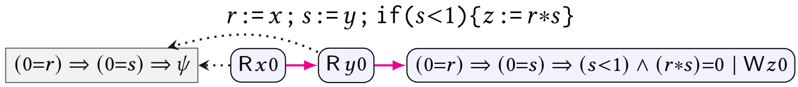
Then the equivalence holds in any context.

Predicate transformers are a good fit for logical models of dependency calculation, since both are concerned with preconditions and how they are transformed by sequential composition. 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:



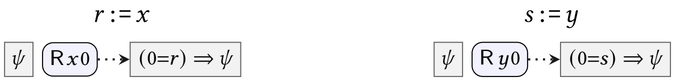
The predicate transformer from the write matches Dijkstra's d2b. For the reads, however, d2c defines the transformer of $r:=x$ to be $\psi[x/r]$, which is equivalent to $(x{=}r) \Rightarrow \psi$ under the assumption that registers are assigned at most once. Instead, we use $(0{=}r) \Rightarrow \psi$, reflecting the fact that 0 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 the preconditions of subsequent events, composing the predicate transformers. (In subsequent diagrams, we only show predicate transformers for reads.)


r := x; s := y; if(s<1){z := r∗s}
(0=r) ⇒ (0=s) ⇒ ψ
R x0
R y0
(0=r) ⇒ (0=s) ⇒ (s<1) ∧ (r∗s)=0 | W z0

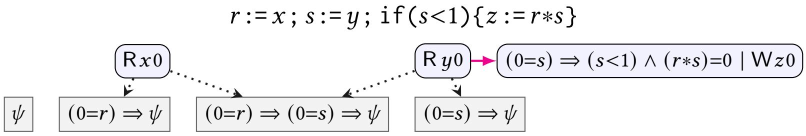
This model works for the sequentially consistent case, but needs to be weakened for the relaxed case. The key observation of this paper is that rather than working with one predicate transformer, we should work with a *family* of predicate transformers, indexed by sets of events.

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 an edge leading to the dependent one. For example:

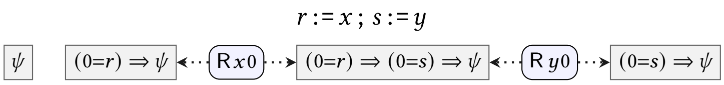


The model of sequential composition then picks which predicate transformer to apply to an event's precondition by picking the one indexed by all the events before it in causal order.

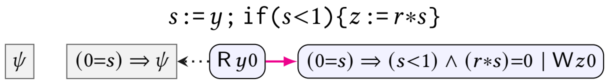
For example, we can recover the expected semantics for (†) by choosing the predicate transformer which is independent of $(\mathsf{R}\,x0)$ but dependent on $(\mathsf{R}\,y0)$, which is the transformer which maps ψ to $(0{=}s) \Rightarrow \psi$.


r := x; s := y; if(s<1){z := r∗s}
R x0
R y0
(0=s) ⇒ (s<1) ∧ (r∗s)=0 | W z0
ψ
(0=r) ⇒ ψ
(0=r) ⇒ (0=s) ⇒ ψ
(0=s) ⇒ ψ

In the diagram, the dotted lines indicate set inclusion into the index of the transformer-family. As a sanity check, we can see that sequential composition is associative in this case, since it does not matter whether we associate to the left, with intermediate step:



or to the right, with intermediate step:



This is an instance of the general result that sequential composition forms a monoid.

3 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. 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 few prior studies of relaxed memory that include sequential composition and/or precise calculation of semantic dependencies. Jagadeesan et al. [2020] give a denotational semantics, using prefixing rather than sequential compositions. Paviotti et al. [2020] give a denotational semantics, calculating dependencies using event structures rather than logic. They give the semantics of sequential composition in continuation passing style, whereas we give it in direct style. This paper provides a general technique for computing sequential dependencies and applies it to these two approaches. We provide a detailed comparison with [Jagadeesan et al. 2020] in §A.3.

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 ARM-V8. 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 Arm [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, whole-program 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]. We provide a detailed comparison with these approaches in §A.2.

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, and the enormous state space for the operational models of Kang et al. [2017] and Chakraborty and Vafeiadis [2019] is a daunting prospect for a tool builder – and as yet no tool exists for automatically evaluating these models. We describe a tool, PwTER, for automatically evaluating PwT in §8.

4 SEQUENTIAL SEMANTICS

After some preliminaries (§4.1–4.2), we define the basic model and establish some basic properties (§4.3 and Fig. 1). We then explain the model using examples (§4.4–4.11). We encourage readers to skim the definitions and then skip to §4.4, coming back as needed.

4.1 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 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 include at least the constants 0 and 1,
- expressions include at least registers and values,
- expressions do *not* include references: $M[N/x] = M$.

We model the following language.

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

$$S \;::=\; r \mathbin{:=} M \mid r \mathbin{:=} [\![L]\!]^{\mu} \mid [\![L]\!]^{\mu} \mathbin{:=} M \mid \mathsf{F}^{\mu} \mid \texttt{skip} \mid S_1 \mathbin{;} S_2 \mid \texttt{if}(M)\{S_1\}\texttt{else}\{S_2\} \mid S_1 \mathbin{\rceil\!\!|} S_2$$

Access modes, μ, are relaxed (rlx), release (rel), acquire (acq), and sequentially consistent (sc). Let expressions ($r \mathbin{:=} M$) only affect thread-local state and thus do not have a mode. Reads ($r \mathbin{:=} [\![L]\!]^{\mu}$) support rlx, acq, sc. Writes ($[\![L]\!]^{\mu} \mathbin{:=} r$) support rlx, rel, sc. Fences (F^{μ}) support rel, acq, sc.

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

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

Throughout §1–8 we require that

- each register is assigned at most once in a program.

In §9, we drop this restriction, requiring instead that

- there are registers $\mathcal{S}_{\mathcal{E}} = \{s_e \mid e \in \mathcal{E}\}$, that do not appear in programs: $S[N/s_e] = S$.

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 ($x{=}M$),
- 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 $\mathcal{A} \times \mathcal{A}$: *overlaps*, *matches*, *blocks*, and delays.

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.

4.2 Actions in This Paper

In this paper, we let actions be reads and writes and fences:

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

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 4.1. Actions (R) are *read* actions.
We say a *overlaps* b if they access the same location.
We say a *matches* b if $a = (\mathsf{W}xv)$ and $b = (\mathsf{R}xv)$.
We say a *blocks* b if $a = (\mathsf{W}x)$ and $b = (\mathsf{R}x)$, regardless of value.
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: $\ltimes_{\mathsf{sync}} = \{(a, \mathsf{W}^{\sqsupseteq\mathsf{rel}}),\ (a, \mathsf{F}^{\sqsupseteq\mathsf{rel}}),\ (\mathsf{R}, \mathsf{F}^{\sqsupseteq\mathsf{acq}}),$ $(\mathsf{R}^{\sqsupseteq\mathsf{acq}}, a),\ (\mathsf{F}^{\sqsupseteq\mathsf{acq}}, a),\ (\mathsf{F}^{\sqsupseteq\mathsf{rel}}, \mathsf{W}),\ (\mathsf{W}^{\sqsupseteq\mathsf{rel}}x, \mathsf{W}x)\}$.
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 $a \bowtie_{\mathsf{sc}} b$.

4.3 PwT: Pomsets with Predicate Transformers

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

Definition 4.2. A *predicate transformer* is a function $\tau : \Phi \longrightarrow \Phi$ such that
(x1) if $\phi \vDash \psi$, then $\tau(\phi) \vDash \tau(\psi)$,
(x2) $\tau(\psi_1 \land \psi_2) \equiv \tau(\psi_1) \land \tau(\psi_2)$,
(x3) $\tau(\psi_1 \lor \psi_2) \equiv \tau(\psi_1) \lor \tau(\psi_2)$.

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. In sequential composition, we will use $\tau^{\downarrow e}$ as the predicate transformer applied to event e where $d \in (\downarrow e)$ if $d < e$.

Definition 4.3. 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 $\tau^C(\psi) \vDash \tau^D(\psi)$.
We write $\tau(\psi)$ as an abbreviation of $\tau^E(\psi)$.

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 $\tau(\psi)$—the transformer of the biggest set—is the *best*. (Note that the definition is written to be insensitive to events outside E.)

In sequential composition, adding more order can only increase the size of $\downarrow e$. Following Def. 4.3, the larger $\downarrow e$ is, the better, at least in terms of satisfying preconditions. Thus more order means weaker preconditions.

Definition 4.4. A *pomset with predicate transformers* (PwT) is a tuple $(E, \lambda, \kappa, \tau, \checkmark, \leq)$ where
(M1) $E \subseteq \mathcal{E}$ is a set of *events*,
(M2) $\lambda : E \longrightarrow \mathcal{A}$ defines a *label* for each event,
(M3) $\kappa : E \longrightarrow \Phi$ defines a *precondition* for each event,
(M4) $\tau : 2^{\mathcal{E}} \longrightarrow \Phi \longrightarrow \Phi$ is a *family of predicate transformers* over E,
(M5) $\checkmark : \Phi$ is a *termination condition*, such that
  (M5a) $\checkmark \vDash \tau(\mathsf{tt})$,
(M6) $\leq \subseteq E \times E$, is a 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 give the semantics of programs $[\![\cdot]\!]$ in Fig. 1.

Let P range over pomsets, and $\mathcal{P}$ over sets of pomsets.

The model has 6 components, which can be daunting at first glance. To aid the reader, we use consistent numbering throughout. For example, item 6 always refers to the order relation.

The core of the model is a pomset, which includes 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. Sequential dependency is calculated by κ_2' in the semantics of sequential composition.

Before discussing the details of the model, we note that the semantics satisfies the expected monoid laws and is closed with respect to *augmentation*. Augments include more order and stronger formulae; in examples, we typically consider pomsets that are augment-minimal. One intuitive reading of augment closure is that adding order can only cause preconditions to weaken.

Lemma 4.5. *(a)* $\mathcal{P} = (\mathcal{P}\,;\mathit{SKIP}) = (\mathit{SKIP}\,;\mathcal{P})$.
(b) $(\mathcal{P}_1\,;\mathcal{P}_2)\,;\mathcal{P}_3 = \mathcal{P}_1\,;(\mathcal{P}_2\,;\mathcal{P}_3)$.

Proof. Straightforward calculation. (a) requires M5a for the termination condition in $(\mathcal{P}\,;\mathit{SKIP})$. (b) requires both conjunction closure (X2, for the termination condition) and disjunction closure (X3, for the predicate transformers themselves). □

Lemma 4.6. *(d)* $\texttt{if}(\phi)\{\texttt{if}(\psi)\{\mathcal{P}\}\} = \texttt{if}(\phi \land \psi)\{\mathcal{P}\}$.
(e) $\texttt{if}(\phi)\{\mathcal{P}_1\,;\mathcal{P}_3\}\,\texttt{else}\,\{\mathcal{P}_2\,;\mathcal{P}_3\} \supseteq \texttt{if}(\phi)\{\mathcal{P}_1\}\,\texttt{else}\,\{\mathcal{P}_2\}\,;\mathcal{P}_3$.
(f) $\texttt{if}(\phi)\{\mathcal{P}_1\,;\mathcal{P}_2\}\,\texttt{else}\,\{\mathcal{P}_1\,;\mathcal{P}_3\} \supseteq \mathcal{P}_1\,;\texttt{if}(\phi)\{\mathcal{P}_2\}\,\texttt{else}\,\{\mathcal{P}_3\}$.
(g) $\texttt{if}(\phi)\{\mathcal{P}\}\,\texttt{else}\,\{\mathcal{P}\} \supseteq \mathcal{P}$.
(h) $\texttt{if}(\phi)\{\mathcal{P}_1\}\,\texttt{else}\,\{\mathcal{P}_2\} \supseteq \mathcal{P}_1$ *if* ϕ *is a tautology.*
(i) $\texttt{if}(\phi)\{\mathcal{P}_1\}\,\texttt{else}\,\{\mathcal{P}_2\} = \texttt{if}(\phi)\{\mathcal{P}_1\}\,;\texttt{if}(\neg\phi)\{\mathcal{P}_2\}$.
(j) $\texttt{if}(\phi)\{\mathcal{P}_1\}\,\texttt{else}\,\{\mathcal{P}_2\} = \texttt{if}(\neg\phi)\{\mathcal{P}_2\}\,;\texttt{if}(\phi)\{\mathcal{P}_1\}$.

Proof. Straightforward calculation. □

In §9.4, we refine the semantics to validate the reverse inclusions for (e), (f), and (g). In §4.11, we refine the semantics to validate the reverse inclusion for (h).

Definition 4.7. P_2 is an *augment* of P_1 if all fields are equal except, perhaps, the order, where we require $\le_2 \supseteq \le_1$.

Lemma 4.8. *If* $P_1 \in [\![S]\!]$ *and* P_2 *augments* P_1 *then* $P_2 \in [\![S]\!]$.

Proof. Induction on the definition of $[\![\cdot]\!]$. □

4.4 Pomsets and Complete Pomsets

Ignoring the logic, the definitions are straightforward. Reads and writes map to pomsets with at most one event. `skip` maps to the empty pomset. Note only that $[\![x := 1]\!]$ can write any value v; the fact that v must be 1 is captured in the logic.

The structural rules combine pomsets: *SEQ* and *IF* perform a union, inheriting labeling and order from the two sides. 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

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

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 $\kappa'_2(e) = \tau_1^{\downarrow e}(\kappa_2(e))$, where $\downarrow e = \{c \mid c < e\}$,

(s1) $E = (E_1 \cup E_2)$,
(s2) $\lambda = (\lambda_1 \cup \lambda_2)$,
(s3a) if $e \in E_1 \setminus E_2$ then $\kappa(e) \equiv \kappa_1(e)$,
(s3b) if $e \in E_2 \setminus E_1$ then $\kappa(e) \equiv \kappa'_2(e)$,
(s3c) if $e \in E_1 \cap E_2$ then $\kappa(e) \equiv \kappa_1(e) \vee \kappa'_2(e)$,
(s4) $\tau^D(\psi) \equiv \tau_1^D(\tau_2^D(\psi))$,
(s5) $\checkmark \equiv \checkmark_1 \wedge \tau_1(\checkmark_2)$,
(s6) $\leq\ \supseteq\ \leq_1 \cup \leq_2$.

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) $\lambda = (\lambda_1 \cup \lambda_2)$,
(I3a) if $e \in E_1 \setminus E_2$ then $\kappa(e) \equiv \phi \wedge \kappa_1(e)$,
(I3b) if $e \in E_2 \setminus E_1$ then $\kappa(e) \equiv \neg\phi \wedge \kappa_2(e)$,
(I3c) if $e \in E_1 \cap E_2$ then $\kappa(e) \equiv (\phi \wedge \kappa_1(e)) \vee (\neg\phi \wedge \kappa_2(e))$,
(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) $\leq\ \supseteq\ \leq_1 \cup \leq_2$.

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

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

(W1) $|E| \leqslant 1$,
(W2) $\lambda(e) = \mathsf{W}^{\mu}xv$,
(W3) $\kappa(e) \equiv M{=}v$,
(W4a) if $E \neq \emptyset$ then $\tau^D(\psi) \equiv \psi[M/x][M{=}v/\mathsf{Q}_x]$,
(W4b) if $E = \emptyset$ then $\tau^D(\psi) \equiv \psi[M/x][\mathsf{ff}/\mathsf{Q}_x]$,
(W5a) if $E \neq \emptyset$ then $\checkmark \equiv M{=}v$,
(W5b) if $E = \emptyset$ then $\checkmark \equiv \mathsf{ff}$.

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

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

$$\begin{aligned}
[\![r := M]\!] &= \mathit{LET}(r, M) & [\![\mathtt{skip}]\!] &= \mathit{SKIP}\\
[\![r := x^{\mu}]\!] &= \mathit{READ}(r, x, \mu) & [\![S_1\,;\,S_2]\!] &= \mathit{SEQ}([\![S_1]\!], [\![S_2]\!])\\
[\![x^{\mu} := M]\!] &= \mathit{WRITE}(x, M, \mu) & [\![\mathtt{if}(M)\{S_1\}\,\mathtt{else}\,\{S_2\}]\!] &= \mathit{IF}(M{\neq}0, [\![S_1]\!], [\![S_2]\!])
\end{aligned}$$

Fig. 1. PwT Semantics

$[\![x := 1]\!] \not\supseteq [\![x := 1\,;\,x := 1]\!]$ is not a valid refinement, since the latter includes a two-element pomset, but the former does not.[1]

In complete pomsets, c5 requires that $\checkmark$ is a tautology, capturing termination. In *WRITE*, w5b ensures that all writes are included in complete pomsets. This also ensures $[\![x := 1]\!] \not\supseteq [\![\mathtt{if}(M)\{x := 1\}]\!]$, since $[\![\mathtt{if}(M)\{x := 1\}]\!]$ includes the empty set with termination condition $\neg M$, but $[\![x := 1]\!]$ can only include the empty set with termination condition ff.

In addition, w5a ensures that complete pomsets do not include bogus writes. Suppose $P \in [\![x := 1]\!]$. As we noted above, P can include $(1{=}v \mid \mathsf{W}xv)$, for any value v. In complete pomsets,

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

however, W5a requires that $\checkmark$ implies $1{=}v$. In this case, M3a would filter the pomset, since preconditions must be satisfiable. However, unsatisfiable writes can be become satisfiable via merging:

$$\begin{array}{ccc} x := 1 & x := 2 & \texttt{if}(M)\{x := 3\} \\ \boxed{\mathsf{W}x1} & \boxed{2{=}3 \mid \mathsf{W}x3} & \boxed{M \mid \mathsf{W}x3} \end{array}$$

By merging, the semantics allows the following:

$$\begin{array}{c} x := 1;\ x := 2;\ \texttt{if}(M)\{x := 3\} \\ \boxed{\mathsf{W}x1} \quad \boxed{M \mid \mathsf{W}x3} \end{array}$$

This pomset is incomplete, however, since $\checkmark \equiv 2{=}3$.

In *READ*, $\checkmark$ depends on the mode. R5b ensures that all acquiring reads are included in complete pomsets. Instead R5a states that relaxed reads are optional: $\checkmark$ is alway true for relaxed reads. From this, it is easy to see that $[\![r := x]\!] \supseteq [\![\texttt{skip}]\!]$ is a valid refinement (where the default mode is rlx).

Ignoring predicate transformers, the *SEQ* rule S5 takes $\checkmark$ to be $\checkmark_1 \land \checkmark_2$. This is as expected: the program terminates if both subprograms terminate.

In $\mathit{IF}(\phi, \mathcal{P}_1, \mathcal{P}_2)$, the termination condition (I5) is $(\phi \land \checkmark_1) \lor (\neg\phi \land \checkmark_2)$: the program terminates as long as the "true" branch terminates. Thus $[\![\texttt{if(tt)}\{x := 1\}\texttt{else}\{y := 1\}]\!]$ 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 event at top-level, use the empty set.

4.5 Preconditions, Predicate Transformers, and Data Dependencies

Preconditions are used to calculate dependencies. They also determine which events can appear in a pomset. In a complete pomset, C3 requires that every precondition $\kappa(e)$ is a tautology. Using W3, $[\![x := 2]\!]$ cannot include a pomset with event $(\mathsf{W}x3)$, since $2{=}3$ is not a tautology. The symbols Q_x that occur in R3 and W5 serve similar purpose. We defer discussion of these until §4.8.

Preconditions are discharged during sequential composition by applying predicate transformers τ_1 from the left to preconditions $\kappa_2(e)$ on the right. The specific rules are S3b and S3c, which use the transformed predicate $\kappa_2'(e) = \tau_1^{\downarrow e}(\kappa_2(e))$, where $\downarrow e = \{c \mid c < e\}$ is the set of events that precede e in causal order. We call $\downarrow e$ the *dependent set* for e. Then $E \setminus (\downarrow e)$ 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. (We discuss address dependencies in §9.2.)

Preconditions are introduced in:	Predicate transformers are introduced in:
(I3) for control dependencies,	(R4a) for reads in the dependent set,
(W3) for data dependencies on writes.	(R4b) for reads in the independent set,
	(W5) for writes.

The rules track dependencies. We discuss data dependencies (W3) here and control dependencies (I3) in §4.6. Unless otherwise noted, we assume pomsets are *complete* and *augment-minimal*. We do not discuss S3 further. It simply ensures that all writes are present before a release, even for incomplete pomsets (see §4.4).

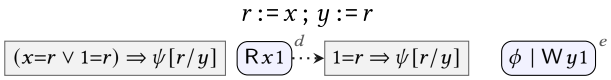
A simple example of a data dependency is a pomset $P \in [\![r := x\,;\, y := r]\!]$. If P is complete, it must have two events. Then *SEQ* requires that there are $P_1 \in [\![r := x]\!]$ and $P_2 \in [\![y := r]\!]$ of the form:

$$\begin{array}{cc} r := x & y := r \\ \boxed{(x{=}r \lor v{=}r) \Rightarrow \psi}\ \ \boxed{\mathsf{R}xv} \overset{d}{\dashrightarrow} \boxed{v{=}r \Rightarrow \psi} & \boxed{\psi[r/y]}\ \ \boxed{r{=}w \mid \mathsf{W}yw} \overset{e}{\dashrightarrow} \boxed{\psi[r/y]} \end{array} \tag{††}$$

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

$$\boxed{(x{=}r \lor 1{=}r) \Rightarrow \psi}\ \ \boxed{\mathsf{R}x1} \overset{d}{\dashrightarrow} \boxed{1{=}r \Rightarrow \psi} \qquad\qquad \boxed{\psi[r/y]}\ \ \boxed{r{=}1 \mid \mathsf{W}y1} \overset{e}{\dashrightarrow} \boxed{\psi[r/y]}$$

For the read, the dependent transformer $\tau_1^{\{d\}}$ is $1{=}r \Rightarrow \psi$; the independent transformer $\tau_1^{\emptyset}$ is $(x{=}r \vee 1{=}r) \Rightarrow \psi$. 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 W5. Combining these into a single pomset, we have:

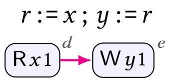


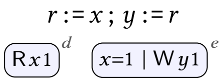
By S4, predicate transformers are determined by composition; thus $\tau^D(\psi)$ is $\tau_1^D(\tau_2^D(\psi))$. Since the transformer does not depend on whether the write is included, we do not draw dependencies for the write in the diagram.

Turning to the precondition ϕ on 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.

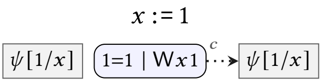
- If $d \leq e$ then we apply the dependent transformer and $\phi = (1{=}r \Rightarrow r{=}1)$, a tautology.
- If $d \not\leq e$ then we apply the independent transformer and $\phi = ((x{=}r \vee 1{=}r) \Rightarrow r{=}1)$. Under the assumption that r is bound, this is logically equivalent to $(x{=}1)$. (We make this more precise in §9.1.)

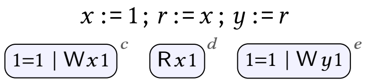
Eliding transformers, the two outcomes are:





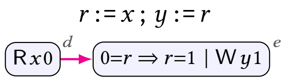
The independent case on the right can only participate in a top-level pomset if the precondition $(x{=}1)$ is discharged. To do so, we must prepend a pomset P_0 that writes 1 to x:

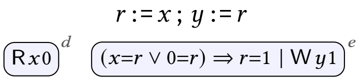




Here we apply the predicate transformer $\tau_0^{\emptyset}$ to $(x{=}1)$, resulting in the tautology $(1{=}1)$.

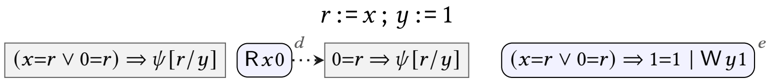
Now suppose that $v \neq w$ in (††). Again there are two possibilities, where we take $v = 0$ and $w = 1$:





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:



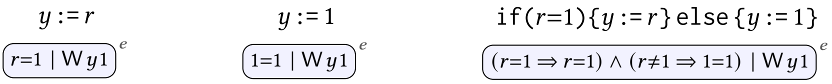
4.6 Control Dependencies

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: $(\phi \Rightarrow \tau_1^D(\psi)) \wedge (\neg\phi \Rightarrow \tau_2^D(\psi))$.

This semantics validates dead code elimination: if $M{\neq}0$ is a tautology then $[\![\texttt{if}(M)\{S_1\}\texttt{else}\{S_2\}]\!] \supseteq [\![S_1]\!]$. The reverse inclusion does not hold.

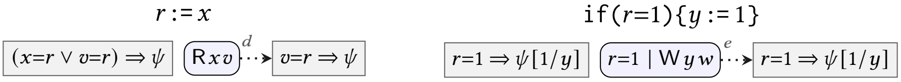
For events from E_1, I3a requires $\phi \wedge \kappa_1(e)$. For events from E_2, I3b requires $\neg\phi \wedge \kappa_2(e)$. For coalescing events in $E_1 \cap E_2$, I3c requires $(\phi \wedge \kappa_1(e)) \vee (\neg\phi \wedge \kappa_2(e))$. This semantics allows common code to be lifted out of a conditional, validating the transformation $[\![\texttt{if}(M)\{S\}\texttt{else}\{S\}]\!] \supseteq [\![S]\!]$.

By allowing events to coalesce, I3C ensures that control dependencies are calculated semantically. For example, consider $P \in [\![\texttt{if}(r{=}1)\{y := r\}\,\texttt{else}\,\{y := 1\}]\!]$, which is build from $P_1 \in [\![y := r]\!]$ and $P_2 \in [\![y := 1]\!]$ such as:

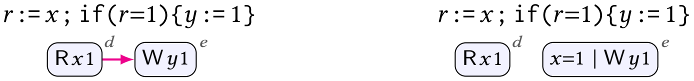

$y := r$ | $y := 1$ | $\texttt{if}(r{=}1)\{y := r\}\,\texttt{else}\,\{y := 1\}$

Here, the precondition in the combined pomset is a tautology, independent of r.

Control dependencies are eliminated in the same way as data dependencies. For example:


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

Reasoning as we did for (††) in §4.5, there are two possibilities:


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

4.7 A Refinement: No Dependencies into Reads

To avoid stalling the CPU pipeline unnecessarily, hardware does not enforce control dependencies between reads. To support if-closure (§9.4), 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.

$$\kappa'_2(e) = \begin{cases} \tau_1(\kappa_2(e)) & \text{if } \lambda(e) \text{ is a read} \\ \tau_1^{\downarrow e}(\kappa_2(e)) & \text{otherwise, where } {\downarrow}e = \{c \mid c < e\} \end{cases}$$

Thus reads always use the "best" transformer, τ_1. In order for non-reads to get a good transformer, they need to add order.

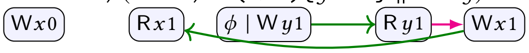
Throughout the remainder of the paper, we use this definition. (The lack of dependencies into reads is one of the factors complicating downset closure; see §A.5 for a discussion.)

4.8 Subtleties

There are two aspects of Fig. 1 that are fairly obscure: local invariant reasoning and Q_x.

First, consider local invariant reasoning, as in JMM causality test case 1 [Pugh 2004]:

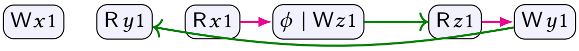
$$x := 0;\ (r := x\,;\, \texttt{if}(r \geqslant 0)\{y := 1\} \parallel x := y)$$



In order to allow this execution, the precondition ϕ must be a tautology. Using R4B and W4A, the precondition 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. Intuitively, R4B says that, to be independent of the read action, subsequent preconditions must be tautological under both $[v/r]$ and $[x/r]$. Here v is the value read, and x tracks the "local state" of the variable. This idea is borrowed from [Jagadeesan et al. 2020], which includes further examples. (See §4.9 for a discussion of Skolemization.)

Second, consider Q_x, which we motivate using the following example [Paviotti et al. 2020, §6.3]:

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



Note that the two branches of the conditional are the same after the first assignment in the else branch. Without Q_x, the precondition ϕ is $(1{=}r \Rightarrow r \neq 0)$, which is a tautology, and the execution is allowed, resulting in a violation of DRF-SC. To construct this pomset, we have chosen the empty pomset for $[\![x := 0]\!]$. The constraints on complete pomsets do not filter out this pomset, since $x := 0$ is in the false-branch of the conditional. The problem here is that we have forgotten the local state of x in the false-branch of the execution. Nonetheless, we are using the subsequent read.

With Q_x, the precondition ϕ is $(1{=}r \Rightarrow (r{\neq}0 \land \mathsf{Q}_x))[\mathsf{ff}/\mathsf{Q}_x]$, which is unsatisfiable. Intuitively, Q_x requires that the most recent prior write to x must be in the pomset in order to read x.

Q_x also guarantees initialization in complete pomsets: (c3) requires tautologies, which means that all variables must be initialized sequentially in order to get rid of Q_x.

The use of Q_x is not too restrictive. As a sanity check, consider the following pomsets,

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

$$\boxed{\mathsf{R}\,y1} \rightarrow \boxed{1{=}r \Rightarrow r{\neq}0 \mid \mathsf{W}x1} \quad \boxed{1{=}r \Rightarrow \tau_1(\mathsf{Q}_x) \mid \mathsf{R}x2} \qquad\qquad \boxed{\mathsf{R}\,y0} \quad \boxed{0{=}r \Rightarrow \tau_2(\mathsf{Q}_x) \mid \mathsf{R}x2}$$

where $\tau_1(\psi) = (r{\neq}0 \land \psi[1/x][1{=}1/\mathsf{Q}_x]) \lor (r{=}0 \land \psi)$ and $\tau_2(\psi) = (r{\neq}0 \land \psi[1/x][\mathsf{ff}/\mathsf{Q}_x]) \lor (r{=}0 \land \psi)$ are the predicate transformers for the conditional writes, using w4a and w4b respectively. The executions are complete since all preconditions are tautologies.

4.9 Associativity and Skolemization

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 $E \neq \emptyset$ and $(E \cap D) \neq \emptyset$ then $\tau^D(\psi) \equiv \psi[v/r]$,
(R4b′) if $E \neq \emptyset$ and $(E \cap D) = \emptyset$ then $\tau^D(\psi) \equiv \psi[v/r] \land \psi[x/r]$.
Sadly, this definition fails associativity.

Consider the following, eliding transformers for the writes ("!" represents logical negation):

$$\begin{array}{ccc} r := y & x := !r & x := !!r \\ \boxed{\psi[1/r] \land \psi[y/r]}\ \ \boxed{\mathsf{R}\,y1} \dashrightarrow \boxed{\psi[1/r]} & \boxed{r{=}0 \mid \mathsf{W}x1} & \boxed{r{\neq}0 \mid \mathsf{W}x1} \end{array}$$

Coalescing the writes and associating to the right, we have the following, since $(r{=}0 \lor r{\neq}0) \equiv \mathsf{tt}$:

$$\begin{array}{ccc} r := y & x := !r\,;\ x := !!r & r := y\,;\ (x := !r\,;\ x := !!r) \\ \boxed{\mathsf{R}\,y1} & \boxed{\mathsf{W}x1} & \boxed{\mathsf{R}\,y1}\ \ \boxed{\mathsf{W}x1} \end{array}$$

The precondition of $(\mathsf{W}x1)$ is a tautology. Associating to the left and the coalescing, instead:

$$\begin{array}{ccc} r := y\,;\ x := !r & x := !!r & (r := y\,;\ x := !r);\ x := !!r \\ \boxed{\mathsf{R}\,y1}\ \ \boxed{1{=}0 \land y{=}0 \mid \mathsf{W}x1} & \boxed{r{\neq}0 \mid \mathsf{W}x1} & \boxed{\mathsf{R}\,y1}\ \ \boxed{\phi \mid \mathsf{W}x1} \end{array}$$

The precondition $\phi = (1{=}0 \land y{=}0) \lor (1{\neq}0 \land y{\neq}0)$, which is equivalent to $y{\neq}0$, which is not a tautology. This pomset can never be complete due to the bogus write of $(x := !r)$, which will show up in the termination condition $(1{=}0)$. Nevertheless, this is a problem, since associativity must hold for incomplete pomsets.

Our solution is to Skolemize, replacing substitution by implication, with uniquely chosen registers. Using Fig. 1, we compute a tautology in the example above: $\phi = ((y{=}r \lor 1{=}r) \Rightarrow r{=}0) \lor (r{\neq}0)$.

The proof of associativity requires that predicate transformers distribute through disjunction (Def. 4.2). The attempt to define predicate transformers using substitution fails for R4c because the predicate transformer $\tau(\psi) = (\forall r)\psi$ does not distribute through disjunction: $\tau(\psi_1 \lor \psi_2) = (\forall r)(\psi_1 \lor \psi_2) \neq ((\forall r)(\psi_1)) \lor ((\forall r)(\psi_2)) = \tau(\psi_1) \lor \tau(\psi_2)$. Since $\tau(\psi) = (\forall r)\psi$ does not distribute through disjunction, we use $\tau(\psi) = \psi$ instead (which trivially distributes through disjunction). This change means we cannot use substitution, since ψ does not imply $\psi[v/r]$. Fortunately, Skolemizing solves this problem, since ψ implies $(r{=}v) \Rightarrow \psi$.

4.10 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:

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

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.

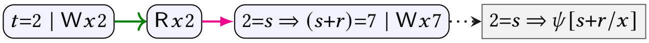
For programs where every register is bound and every read is fulfilled, our dependent transformers are the same as the traditional ones. Thus, when comparing to weakest preconditions, let us only consider totally-ordered executions of our semantics where every read could be fulfilled by prepending some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

For example, let S_i be defined:

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

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.

$$x := t;\ s := x;\ x := s{+}r$$



The predicate transformers are:

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

4.11 Register Consistency

If a precondition is false, you can be pretty sure it's useless. In this subsection, we develop a criterion for eliminating such useless pomsets.

To achieve this, we would like to bolt a requirement into the definition of pomsets in order to weed out the useless ones. Something like this:

(M3a′) $\kappa(e)$ is satisfiable.

For associativity, (M3a′) would in turn require

(X4′) $\tau(\mathsf{ff}) \equiv \mathsf{ff}$.

Dijkstra [1975] requires exactly X4′. Problem solved! 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 these requirements into ones that do hold. The main insight is to pull values for registers from the labels of pomset itself. Thus, we define θ_λ to capture the *register state* of a pomset.

Definition 4.9. Let $\theta_\lambda = \bigwedge_{\{(e,v)\in(E\times\mathcal{V})\,|\,\lambda(e)=(\mathsf{R}\,v)\}}(s_e{=}v)$ where $E = \mathsf{dom}(\lambda)$.
We say that ϕ is *λ-consistent* if $\phi \land \theta_\lambda$ 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 4.10. A *λ-predicate transformer* is a function $\tau : \Phi \to \Phi$ such that

(x1) (x2) (x3) as in Def. 4.2,

(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*,

Given these definitions, we can add the following requirement to the model, which enables us to prune pomsets that include λ-inconsistent preconditions.

(M3a) $\kappa(e)$ is λ-consistent.

5 PwT-MCA: POMSETS WITH PREDICATE TRANSFORMERS FOR MCA

We derive a model of concurrent computation by adding *parallel composition* and *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 ARM-V8, but requires extra synchronization for acquiring reads.

A variant, PwT-MCA$_2$, supports optimal lowering for all access modes on ARM-V8. 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.

The lowering proofs can be found in §B. The proofs use recent alternative characterizations of ARM-V8 [Alglave et al. 2021].

5.1 PwT-MCA1

We define PwT-MCA$_1$ by extending Def. 4.4 and Fig. 1. The definition uses several relations over actions—matches, blocks and delays—as well a distinguished set of read actions; see §4.2.

Definition 5.1. A PwT-MCA$_1$ is a PwT (Def. 4.4) equipped 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 $c \le d$ or $e \le c$,
- (M7c) if $d \xrightarrow{\mathsf{rf}} e$ then $d \le e$.

A PwT-MCA is *complete* if

(c3) (c5) as in Def. 4.4, (c7) if $\lambda(e)$ is a read then there is some $d \xrightarrow{\mathsf{rf}} e$.

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

(P1) $E = (E_1 \uplus E_2)$, (P4) $\tau^D(\psi) \equiv \tau_1^D(\psi)$,
(P2) $\lambda = (\lambda_1 \cup \lambda_2)$, (P5) $\checkmark \equiv \checkmark_1 \wedge \checkmark_2$,
(P3a) if $e \in E_1$ then $\kappa(e) \equiv \kappa_1(e)$, (P6) $\leq\, \supseteq (\leq_1 \cup \leq_2)$,
(P3b) if $e \in E_2$ then $\kappa(e) \equiv \kappa_2(e)$, (P7) $\mathsf{rf} \supseteq (\mathsf{rf}_1 \cup \mathsf{rf}_2)$.

If $P \in \mathit{SEQ}(\mathcal{P}_1, \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)$ $(\exists P_2 \in \mathcal{P}_2)$
(S1) (S2) (S3) (S4) (S5) (S6) as in Fig. 1, (S7) $\mathsf{rf} \supseteq (\mathsf{rf}_1 \cup \mathsf{rf}_2)$.
(S6a) if $\lambda_1(d)$ delays $\lambda_2(e)$ then $d \leq e$,

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) (I2) (I3) (I4) (I5) (I6) as in Fig. 1, (I7) $\mathsf{rf} \supseteq (\mathsf{rf}_1 \cup \mathsf{rf}_2)$.

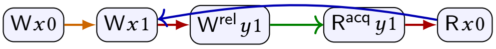
Let $[\![S_1 \parallel S_2]\!] = \mathit{PAR}([\![S_1]\!], [\![S_2]\!])$. We write $[\![\cdot]\!]_{\mathsf{mca1}}$ for the semantic function when it is unclear from context.

In complete pomsets, 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].

The semantic rules are mostly straightforward: Parallel composition is disjoint union, and all constructs respect reads-from. The monoid laws (Lemma 4.5) extend to parallel composition, with `skip` as right unit only due to the asymmetry of P4.

Only S6a requires explanation. From Def. 4.1, 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 and M7c), these ensure coherence, publication, subscription and other idioms. For example, consider the following:[2]

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



(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: (Wx0) $\rightarrow$ (Wx1) is required by $\bowtie_{\mathsf{co}}$. (Wx1) $\rightarrow$ (W$^{\mathsf{rel}}y$1) and (R$^{\mathsf{acq}}y$1) $\rightarrow$ (Rx0) 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 (W$^{\mathsf{rel}}y$1) $\xrightarrow{\mathsf{rf}}$ (R$^{\mathsf{acq}}y$1), and M7c subsequently ensures that (W$^{\mathsf{rel}}y$1) $\leq$ (R$^{\mathsf{acq}}y$1). The *antidependency* (Rx0) $\longrightarrow$ (Wx1) is required by M7b. (Alternatively, we could have (Wx1) $\longrightarrow$ (Wx0), 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 §D and [Jagadeesan et al. 2020, §3.1].

Lemmas 4.5 and 4.6 mostly hold for PwT-MCA$_1$. The exceptions are items (i) and (j), which become inclusions. For example, (i) becomes:

$$\mathtt{if}(\phi)\{\mathcal{P}_1\}\,\mathtt{else}\,\{\mathcal{P}_2\} \supseteq \mathtt{if}(\phi)\{\mathcal{P}_1\};\ \mathtt{if}(\neg\phi)\{\mathcal{P}_2\}$$

The culprit is delay, which introduces order regardless whether preconditions are disjoint. As an example, $[\![\mathtt{if}(r)\{x := 1\}\,\mathtt{else}\,\{x := 2\}]\!]$ has an execution with (r=0 | Wx2) $\rightarrow$ (r≠0 | Wx1), (using augmentation), whereas $[\![\mathtt{if}(r)\{x := 1\};\ \mathtt{if}(!r)\{x := 2\}]\!]$ has no such execution.

For further discussion, see §A.6.

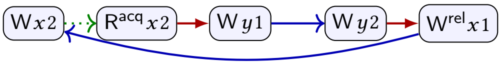
[2] We use different colors for arrows representing order:
- $d \rightarrow e$ arises from $\bowtie_{\mathsf{co}}$ (S6a),
- $d \rightarrow e$ arises from $\ltimes_{\mathsf{sync}}$ or $\bowtie_{\mathsf{sc}}$ (S6a),
- $d \rightarrow e$ arises from control/data/address *dependency* (S3, definition of $\kappa_2'(d)$),
- $d \longrightarrow e$ arises from *reads-from* (M7a),
- $d \longrightarrow e$ arises from *blocking* (M7b).

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

5.2 PwT-MCA2

Lowering PwT-MCA1 to ARM-V8 requires a full fence after every acquiring read. To see why, consider the following attempted execution, where the final values of both x and y are 2.

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



The execution is allowed by ARM-V8, but disallowed by PwT-MCA$_1$, due to the cycle.

ARM-V8 allows the execution because the read of x is internal to the thread. This aspect of ARM-V8 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*. Rather than ensuring that there is no *global* blocker for a sequentially fulfilled read (M7c), we require only that there is no *thread-local* blocker (S6b). For PwT-MCA$_2$, internal reads don't necessarily contribute to order, and thus the above execution is allowed.

Definition 5.2. A PwT-MCA$_2$ is a PwT (Def. 4.4) equipped with an injective relation rf that satisfies requirements M7a and M7b of Def. 5.1.

A A PwT-MCA$_2$ is *complete* if it satisfies C3, C5, and C7.

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) (P2) (P3) (P4) (P5) (P6) (P7) as in Def. 5.1,

(P6a) if $d \in E_1$, $e \in E_2$ and $d \xrightarrow{\mathsf{rf}} e$ then $d \le e$,

(P6b) if $d \in E_1$, $e \in E_2$ and $e \xrightarrow{\mathsf{rf}} d$ then $e \le d$,

(P7a) $\mathsf{rf}_i = \mathsf{rf} \cap (E_i \times E_i)$, for $i \in \{1, 2\}$.

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

(S1) (S2) (S3) (S4) (S5) (S6) (S7) as in Def. 5.1,

(S6a) if $\lambda_1(d)$ delays $\lambda_2(e)$ then either $d \xrightarrow{\mathsf{rf}} e$ or $d \le e$,

(S6b) if $\lambda_1(c)$ blocks $\lambda_2(e)$ and $d \xrightarrow{\mathsf{rf}} e$ then $c \le d$,

(S7a) $\mathsf{rf}_i = \mathsf{rf} \cap (E_i \times E_i)$, for $i \in \{1, 2\}$.

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) (I2) (I3) (I4) (I5) (I6) (I7) as in Def. 5.1,

(I7a) $\mathsf{rf}_i = \mathsf{rf} \cap (E_i \times E_i)$, for $i \in \{1, 2\}$.

A PwT-MCA$_2$ need not satisfy requirement M7c, and thus we may have $d \xrightarrow{\mathsf{rf}} e$ and $e \le d$.

With the weakening of S6b, we must be careful not to allow spurious pairs to be added to the rf relation. Thus we add P7a, S7a, and I7a. For example, I7a ensure that $[\![$`if(`b`){`$r := x \parallel x := 1$`}else{`$r := x;\ x := 1$`}`$]\!]$ does not include $\boxed{\mathsf{R}x1} \rightleftarrows \boxed{\mathsf{W}x1}$, taking rf from the left and $\le$ from the right.

PwT-MCA$_2$ does not enforce M7c: $d \xrightarrow{\mathsf{rf}} e$ may not imply $d \le e$ when d and e come from different sides of a sequential composition. This means that rf must be verified during pomset construction, rather than post-hoc. In §7, 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.

LEMMA 5.3. *If $P \in [\![S]\!]_{\mathsf{mca2}}$ is complete, then for every $d \xrightarrow{\mathsf{rf}} e$ either*

- *external fulfillment: $d \le e$ 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 $(\nexists c')\ \kappa(c)$ is a tautology and $\lambda(c)$ blocks $\lambda(e)$ and $d' \xrightarrow{\mathsf{po}} c \xrightarrow{\mathsf{po}} e'$.*

These mimic the *external consistency* requirements of ARM-V8 [Alglave et al. 2021].

6 PwT-MCA RESULTS

PwP [Jagadeesan et al. 2020] is a novel memory model, intended to serve as a semantic basis for a Java-like language, where all access is safe. PwT-MCA generalizes PwP, making several small but significant changes. As a result, we have had to re-prove most of the theorems from PwP.

In §B, we show that PwT-MCA$_1$ supports the optimal lowering of relaxed accesses to ARM-V8 and that PwT-MCA$_2$ supports the optimal lowering of *all* accesses to ARM-V8. The proofs are based on two recent characterizations of ARM-V8 [Alglave et al. 2021]. For PwT-MCA$_1$, we use *External Global Consistency*. For PwT-MCA$_2$, we use *External Consistency*.

In §C, we prove sequential consistency for local-data-race-free programs. The proof uses *program order*, which we construct for C11 in §7. The same construction works for PwT-MCA. (This proof assumes there are no RMW operations.)

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

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

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

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

Notably, the semantics does *not* validate read introduction. When combined with case analysis (§9.4), read introduction can break temporal reasoning. This combination is allowed by speculative operational models. See §A.2 for a discussion.

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. It's proof, and all of the no-thin-air examples in [Jagadeesan et al. 2020, §6] hold equally for the models described here.

7 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 rules 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.

The chief difficulty is instrumenting our semantics to generate program order, for use in the various axioms of C11.

Definition 7.1. A PwT-PO is a PwT (Def. 4.4) equipped with relations π and po such that

(M8) $\pi : (E \longrightarrow E)$ 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) $\mathsf{po} \subseteq (S \times S)$ 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.

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

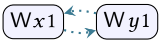
We use π to partition events E in two ways: we distinguish *real* events R from *phantom* events $\overline{R}$; we distinguish *simple* events S from *compound* events $\overline{S}$. From idempotency, it follows that all phantom events are simple ($\overline{R} \subseteq S$) and all compound events 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 7.2. *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, $\text{po} = \text{po}_1 \cup \text{po}_2 \cup E_1 \times E_2$. For the conditional, $\text{po} = \text{po}_1 \cup \text{po}_2$. By construction, po is a pre-order, which may include cycles due to coalescing. For example:

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



To find an acyclic po′, we replay the construction of P to get P'. When building P', we require disjoint union in s1 and i1: $E' = E'_1 \uplus E'_2$. If and event is unmerged in P (*i.e.* $e \in E_1 \uplus E_2$) then we choose the same event name for E' in P'. If an event is merged in P (*i.e.* $e \in E_1 \cap E_2$) 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 $\leq' = \text{po}'$.

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) $d \leq'' e$ exactly when $\pi(d) \leq \pi(e)$, (7) $\text{po}'' = \text{po}'$, and (8) π'' is the constructed merge function. □

Definition 7.3. For a PwT-po, let $\text{extract}(P)$ 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, $\text{extract}(P)$ 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 shows that if P is *complete*, then $\text{extract}(P)$ includes at least one simple event for every compound event in P.

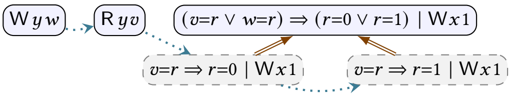
Lemma 7.4. *If P is a complete* PwT-po *with compound event e, then there is a phantom event $c \in \pi^{-1}(e)$ such that $\kappa(c)$ is a tautology.*

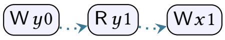
Proof. Immediate from m8b. □

A pomset in the image of extract is a *candidate execution.*

As an example, consider Java Causality Test Case 6. Taking $w = 0$ and $v = 1$, the PwT-po on the left below produces the candidate execution on the right. In diagrams, we visualize po using a dotted arrow ⋯➝, and π using a double arrow ⟹.

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

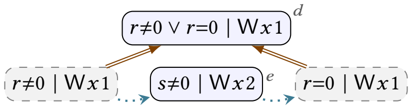

Wyw
Ryv
$(v{=}r \vee w{=}r) \Rightarrow (r{=}0 \vee r{=}1) \mid$ Wx1
$v{=}r \Rightarrow r{=}0 \mid$ Wx1
$v{=}r \Rightarrow r{=}1 \mid$ Wx1



We write $[\![\cdot]\!]^{\text{po}}$ for the semantic function defined by applying the construction of Lemma 7.2 to the base semantics of 1.

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

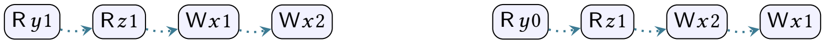
$$\texttt{if}(r)\{x\,{:=}\,1\};\ \texttt{if}(s)\{x\,{:=}\,2\};\ \texttt{if}(!r)\{x\,{:=}\,1\}$$


$r{\neq}0 \lor r{=}0 \mid \mathsf{W}x1$
d
$r{\neq}0 \mid \mathsf{W}x1$
$s{\neq}0 \mid \mathsf{W}x2$
e
$r{=}0 \mid \mathsf{W}x1$

(‡)

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

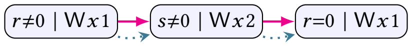
$$r\,{:=}\,y;\ s\,{:=}\,z;\ \texttt{if}(r)\{x\,{:=}\,1\};\ \texttt{if}(s)\{x\,{:=}\,2\};\ \texttt{if}(!r)\{x\,{:=}\,1\}$$



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

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

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


$r{\neq}0 \mid \mathsf{W}x1$
$s{\neq}0 \mid \mathsf{W}x2$
$r{=}0 \mid \mathsf{W}x1$

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

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

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

8 PwTer: AUTOMATIC LITMUS TEST EVALUATOR

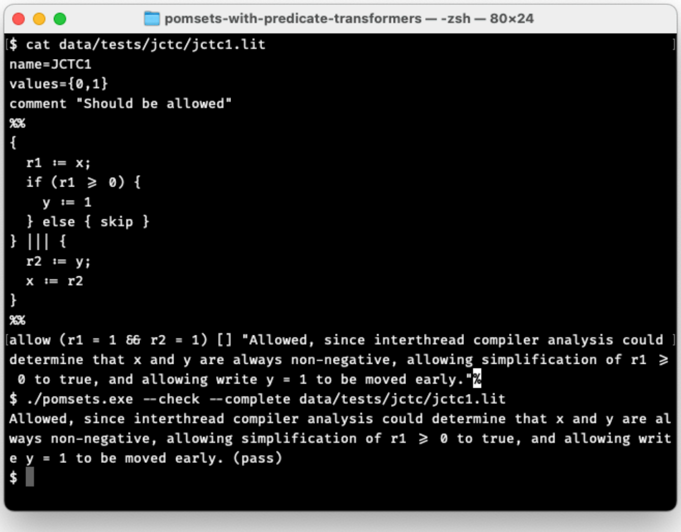
PwTER automatically and exhaustively calculates the allowed outcomes of litmus tests for the PwT, PwT-PO, and PwT-C11 models. It is built in OCaml, and uses Z3 [De Moura and Bjørner 2008] to judge the truth of predicates constructed by the models. PwTER obviates the need for error-prone hand evaluation.

PwTER allows several modes of evaluation: it can evaluate the rules of Fig. 1, implementing PwT; it can generate program order according to Section 7, implementing PwT-PO; and similar to MRD-C11 [Paviotti et al. 2020], it can construct C11-style pre-executions and filter them according to the rules of C11. Our choice of C11 rules is careful: we take the rules of RC11 [Lahav et al. 2017], but we replace the overly strong No-Thin-Air restriction acyclic($\mathsf{rf} \cup \mathsf{po}$) with acyclic($\mathsf{rf} \cup <$). This is a more precise categorisation of thin-air behavior, and it allows aggressive compiler optimizations which would be erroneously forbidden by RC11's original No-Thin-Air axiom. We show PwTER in action in Fig. 2. Finally, PwTER also allows us to toggle the complete check of 4.4, providing an interface for understanding how fragments of code might compose by exposing preconditions and termination conditions which are not yet tautologies.

9 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 the set $\mathcal{S}_{\mathcal{E}} = \{s_e \mid e \in \mathcal{E}\}$. By assumption (§4.1), these registers do not appear in programs: $S[N/s_e] = S$. The resulting semantics satisfies redundant read elimination.

In this section we consider several mostly-orthogonal features: address calculation, if-closure, fences, and read-modify-write operations. Address calculation and if-closure do have some interaction, and we spell out the combined semantics in §A.1.



```
$ cat data/tests/jctc/jctc1.lit
name=JCTC1
values={0,1}
comment "Should be allowed"
%%
{
  r1 := x;
  if (r1 ⩾ 0) {
    y := 1
  } else { skip }
} ||| {
  r2 := y;
  x := r2
}
%%
allow (r1 = 1 && r2 = 1) [] "Allowed, since interthread compiler analysis could
determine that x and y are always non-negative, allowing simplification of r1 ⩾
 0 to true, and allowing write y = 1 to be moved early."%
$ ./pomsets.exe --check --complete data/tests/jctc/jctc1.lit
Allowed, since interthread compiler analysis could determine that x and y are al
ways non-negative, allowing simplification of r1 ⩾ 0 to true, and allowing writ
e y = 1 to be moved early. (pass)
$ 
```

Fig. 2. Example output of PwTer, validating TC1 [Pugh 2004].

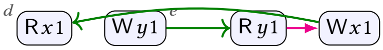
It is worth pointing out that address calculation and if-closure only affect the semantics of read and write. Fences introduce new trivial actions. RMWs require more infrastructure in order to ensure atomicity while compiling to ARM-V8.

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 §6 also apply equally, with the exception of RMWs: we have not proven DRF-SC or ARM-V8lowering for RMWs.

9.1 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."

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



This execution is not allowed by the semantics Fig. 1: the precondition of e in the independent case is

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

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

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

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 in the case that the events are destined to coalesce.

Definition 9.1. Let $[\![\cdot]\!]$ be defined as in Fig. 1, changing R4 of *READ*:

(R4a) if $e \in E$ and $e \in D$ then $\tau^D(\psi) \equiv v{=}s_e \Rightarrow \psi[s_e/r]$,
(R4b) if $e \in E$ and $e \notin D$ then $\tau^D(\psi) \equiv (v{=}s_e \vee x{=}s_e) \Rightarrow \psi[s_e/r]$,
(R4c) if $E = \emptyset$ then $(\forall s)\ \tau^D(\psi) \equiv \psi[s/r]$.

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 [Sevčík and Aspinall 2008, Fig. 5], referenced in [Paviotti et al. 2020, §6.4]. Consider the case where the reads are merged, both seeing 1:

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

In order to 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:

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

9.2 Address Calculation

Inevitably, address calculation complicates the definitions of *WRITE* and *READ*.

Definition 9.2. Let $[\![\cdot]\!]$ be defined as in Fig. 1, changing *WRITE* and *READ*:

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

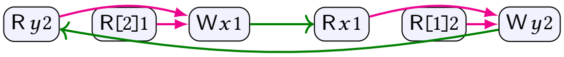
(W1) if $|E| \leqslant 1$,
(W2) $\lambda(e) = \mathsf{W}^\mu[\ell]v$,
(W3) $\kappa(e) \equiv L{=}\ell \wedge M{=}v$,
(W4a) if $E \neq \emptyset$ then $\tau^D(\psi) \equiv (L{=}\ell) \Rightarrow \psi[M/[\ell]][M{=}v/\mathsf{Q}_{[\ell]}]$,
(W4b) if $E = \emptyset$ then $(\forall k)\ \tau^D(\psi) \equiv (L{=}k) \Rightarrow \psi[M/[k]][\mathsf{ff}/\mathsf{Q}_{[k]}]$,
(W5a) if $E \neq \emptyset$ then $\checkmark \equiv L{=}\ell \wedge M{=}v$,
(W5b) if $E = \emptyset$ then $\checkmark \equiv \mathsf{ff}$.

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 $e \in E$ and $e \in D$ then $\tau^D(\psi) \equiv (L{=}\ell \Rightarrow v{=}s_e) \Rightarrow \psi[s_e/r]$,
(R4b) if $e \in E$ and $e \notin D$ then $\tau^D(\psi) \equiv ((L{=}\ell \Rightarrow v{=}s_e) \vee (L{=}\ell \Rightarrow [\ell]{=}s_e)) \Rightarrow \psi[s_e/r]$,
(R4c) if $E = \emptyset$ then $(\forall s)\ \tau^D(\psi) \equiv \psi[s/r]$,
(R5a) if $E \neq \emptyset$ or $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{tt}$.
(R5b) if $E = \emptyset$ and $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \mathsf{ff}$.

The combination of read-read independency (§4.7) 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:

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



This execution would become possible, however, if we were to replace $(L{=}\ell \Rightarrow v{=}s_e)$ by $(v{=}s_e)$ in R4a. In this case, $(\mathsf{R}\,y2)$ would not necessarily be dependency ordered before $(\mathsf{W}\,x1)$.

9.3 Fences and Read-Modify-Write Operations

The semantics of fences is straightforward. Let $[\![\mathsf{F}^{\mu}]\!] = \mathit{FENCE}(\mu)$, where if $P \in \mathit{FENCE}(\mu)$ then

(F1) $|E| \leqslant 1$, (F3) $\kappa(e) \equiv \mathsf{tt}$, (F5a) if $E \neq \emptyset$ then $\checkmark \equiv \mathsf{tt}$,
(F2) $\lambda(e) = \mathsf{F}^{\mu}$, (F4) $\tau^{D}(\psi) \equiv \psi$, (F5b) if $E = \emptyset$ then $\checkmark \equiv \mathsf{ff}$.

This semantics is identical to that of [Jagadeesan et al. 2020]; see there for examples.

RMW operations are more complex. To support RMWs, we add a relation $\xrightarrow{\mathsf{rmw}} \subseteq E \times E$ that relates the read of a successful RMW to the succeeding write.

Definition 9.3. Extend the definition of a pomset as follows.

(M10) $\mathsf{rmw} : E \rightarrow E$ 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 $d \leq e$,
  (M10c) if $\lambda(c)$ overlaps $\lambda(d)$ then
    (i) if $d \xrightarrow{\mathsf{rmw}} e$ then $c \leq e$ implies $c \leq d$,
    (ii) if $d \xrightarrow{\mathsf{rmw}} e$ then $d \leq c$ implies $e \leq c$.

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

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

To define specific operations, 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. The corresponding semantic functions are as follows.

Definition 9.4. Let *READ′* be defined as for *READ*, adding the constraint:

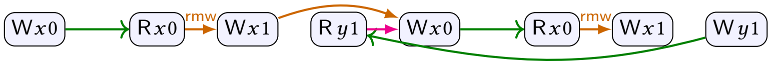
(R4d) if $(E \cap D) = \emptyset$ then $\tau^{D}(\psi) \vDash \psi$.

If $P \in \mathit{FADD}(r, L, M, \mu, \nu)$ then $(\exists P_1 \in \mathit{SEQ}(\mathit{READ}'(r,\ L,\ \mu),\ \mathit{WRITE}(L,\ r{+}M,\ \nu)))$
  (U1) if $\lambda_1(e)$ is a write then there is a read $\lambda_1(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.
If $P \in \mathit{EXCHG}(r, L, M, \mu, \nu)$ then $(\exists P_1 \in \mathit{SEQ}(\mathit{READ}'(r,\ L,\ \mu),\ \mathit{WRITE}(L,\ M,\ \nu)))$
  (U1) if $\lambda_1(e)$ is a write then there is a read $\lambda_1(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.
If $P \in \mathit{CAS}(r, L, M, N, \mu, \nu)$ then $(\exists P_1 \in \mathit{SEQ}(\mathit{READ}'(r,\ L,\ \mu),\ \mathit{IF}(r{=}M,\ \mathit{WRITE}(L,\ N,\ \nu),\ \mathit{SKIP})))$
  (U1) if $\lambda_1(e)$ is a write then there is a read $\lambda_1(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.

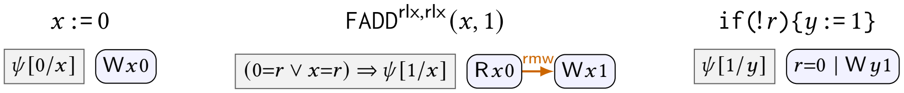
This definition ensures atomicity and supports lowering to Arm load/store exclusive operations. See [Jagadeesan et al. 2020] for examples.

One subtlety of the definition is that we use *READ′* rather than *READ*. Thus, for RMW operations, 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′*.

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



A write should only be visible to one FADD instruction, but here the write of 0 is visible to two. This is allowed because no order is required from (Rx0) to (Wy1) 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:

$$x := 0 \qquad\qquad \mathsf{FADD}^{\mathsf{rlx,rlx}}(x,1) \qquad\qquad \mathtt{if}(!r)\{y := 1\}$$
$$\boxed{\psi[0/x]}\ \boxed{\mathsf{W}x0} \qquad \boxed{\psi[1/x]}\ \boxed{\mathsf{R}x0}\xrightarrow{\mathsf{rmw}}\boxed{\mathsf{W}x1} \qquad \boxed{\psi[1/y]}\ \boxed{r{=}0 \mid \mathsf{W}y1}$$

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 $(\mathsf{W}y1)$.

9.4 If-Closure

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:

$$\mathtt{if}(M)\{x := 1\};\ S;\ \mathtt{if}(\neg M)\{x := 1\}$$
$$\boxed{\mathsf{W}x1}\rightarrow\boxed{\mathsf{W}x1}$$

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

$$\mathtt{if}(M)\{x := 1\} \qquad\qquad x := 1 \qquad\qquad \mathtt{if}(\neg M)\{x := 1\}$$
$$\boxed{M \mid \mathsf{W}x1} \qquad\qquad \boxed{\mathsf{W}x1} \qquad\qquad \boxed{\neg M \mid \mathsf{W}x1}$$

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.

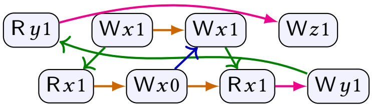
Def. 9.5 allows the execution, by splitting the middle command:

$$\mathtt{if}(M)\{x := 1\} \qquad\qquad x := 1 \qquad\qquad \mathtt{if}(\neg M)\{x := 1\}$$
$$^{d}\boxed{M \mid \mathsf{W}x1} \qquad ^{d}\boxed{\neg M \mid \mathsf{W}x1}\ \ ^{e}\boxed{M \mid \mathsf{W}x1} \qquad ^{e}\boxed{\neg M \mid \mathsf{W}x1}$$

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.

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



Definition 9.5. Let $[\![\cdot]\!]$ be defined as in Fig. 1, changing *WRITE* and *READ*:

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

(W1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,

(W2) $\lambda(e) = \mathsf{W}^{\mu} x v_e$,

(W3) $\kappa(e) \equiv \theta_e \wedge M{=}v_e$,

(W4) $\tau^D(\psi) \equiv \bigwedge_{e \in E} \theta_e \Rightarrow \psi[M/x][M{=}v_e/\mathsf{Q}_x]$
$\wedge(\bigwedge_{e\in E} \neg\theta_e) \Rightarrow \psi[M/x][\mathsf{ff}/\mathsf{Q}_x]$

(W5) $\checkmark \equiv (\bigvee_{e\in E} \theta_e) \wedge (\bigwedge_{e\in E} \theta_e \Rightarrow M{=}v_e)$,

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

(R1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,

(R2) $\lambda(e) = \mathsf{R}^{\mu} x v_e$

(R3) $\kappa(e) \equiv \theta_e \wedge \mathsf{Q}_x$,

(R4) $(\forall s)\tau^D(\psi) \equiv \bigwedge_{e\in E\cap D} \theta_e \Rightarrow v_e{=}s_e \Rightarrow \psi[s_e/r]$
$\wedge \bigwedge_{e\in E\setminus D} \theta_e \Rightarrow (v_e{=}s_e \vee x{=}s_e) \Rightarrow \psi[s_e/r$
$\wedge(\bigwedge_{e\in E} \neg\theta_e) \Rightarrow \psi[s/r]$

(R5a) if $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{tt}$. (R5b) if $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \bigvee_{e \in E} \theta_e$.

We show how to combine address calculation and if-closure in §A.1.

10 CONCLUSIONS

This paper is the first to present a direct denotational semantics for sequential composition in a relaxed memory model which can be efficiently compiled to modern CPUs. There is, as usual, more research to be done.

We have not treated loops in this model, though 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; a similar approach could be applied here.

We've also not handled access elimination: store-forwarding and dead-write-removal are unsound. We expect that these can be validated by allowing events with different labels to merge in some instances.

PwT-MCA$_1$ is a simpler model than PwT-MCA$_2$, but requires fences on acquiring reads for ARM-v8. It would be illuminating to find out what the performance penalty is for these fences.

In a similar vein, the promising semantics (PS) validates read introduction whereas PwT-MCA does not. As a result PS admits behaviors that break reasoning about temporal safety properties (see §A.2). Nonetheless, read introduction is ubiquitous is some compilers. It would be interesting to know if there is a performance penalty for banning read introduction.

REFERENCES

Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *TOPLAS* (2021). To Appear.

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 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

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. *Proc. ACM Program. Lang.* 2, PLDI. To Appear.

Simon Cooksey, Sarah Harris, Mark Batty, Radu Grigore, and Mikoláš Janota. 2019. PrideMM: Second Order Model Checking for Memory Consistency Models. In *10th Workshop on Tools for Automatic Program Analysis.*

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

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

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

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

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

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). arXiv:1804.04214 http://arxiv.org/abs/1804.04214

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

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

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

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

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

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

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

William Pugh. 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/.

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*. 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 DISCUSSION

A.1 Combining Address Calculation and If-Closure

Def. 9.2 is naive with respect to merging events. Consider the following example:

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

$$\overset{c}{\boxed{r{=}1 \mid \mathsf{W}[1]0}}\;\; \overset{d}{\boxed{r{=}1 \mid \mathsf{W}[0]0}} \qquad\qquad \overset{d}{\boxed{r{=}0 \mid \mathsf{W}[0]0}} \rightarrow \overset{e}{\boxed{r{=}0 \mid \mathsf{W}[0]1}}$$

Merging, we have:

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

$$\overset{c}{\boxed{r{=}1 \mid \mathsf{W}[1]0}}\;\; \overset{d}{\boxed{r{=}0 \lor r{=}1 \mid \mathsf{W}[0]0}} \rightarrow \overset{e}{\boxed{r{=}0 \mid \mathsf{W}[0]1}}$$

The precondition of $\mathsf{W}[0]0$ is a tautology; however, this is not possible for $([r] := 0\,;\,[0] := !r)$ alone, using Def. 9.2.

Def. A.1, enables this execution using if-closure. Under this semantics, we have:

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

$$\overset{c}{\boxed{r{=}1 \mid \mathsf{W}[1]0}}\;\; \overset{d}{\boxed{r{=}0 \mid \mathsf{W}[0]0}} \qquad\qquad \overset{d}{\boxed{r{=}1 \mid \mathsf{W}[0]0}}\;\; \overset{e}{\boxed{r{=}0 \mid \mathsf{W}[0]1}}$$

Sequencing and merging:

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

$$\overset{c}{\boxed{r{=}1 \mid \mathsf{W}[1]0}}\;\; \overset{d}{\boxed{r{=}0 \lor r{=}1 \mid \mathsf{W}[0]0}} \rightarrow \overset{e}{\boxed{r{=}0 \mid \mathsf{W}[0]1}}$$

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

Definition A.1. Let $[\![\cdot]\!]$ be defined as in Fig. 1, changing *WRITE* and *READ*:

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

(w1) if $\theta_d \land \theta_e$ is satisfiable then $d = e$, (w5a) $\checkmark \equiv \theta_e \Rightarrow L{=}\ell_e \land M{=}v_e$,

(w2) $\lambda(e) = \mathsf{W}^{\mu}[\ell]v_e$, (w5b) $\checkmark \equiv \bigvee_{e \in E} \theta_e$.

(w3) $\kappa(e) \equiv \theta_e \land L{=}\ell_e \land M{=}v_e$,

(w4) $(\forall k)\tau^{D}(\psi) \equiv \bigwedge_{e \in E} \theta_e \Rightarrow (L{=}\ell) \Rightarrow \psi[M/x][M{=}v_e/\mathsf{Q}_x]$
$\land(\bigwedge_{e \in E} \neg\theta_e) \Rightarrow (L{=}k) \Rightarrow \psi[M/x][\mathsf{ff}/\mathsf{Q}_x]$

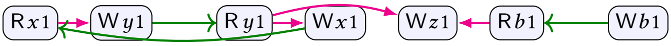
If $P \in \mathit{READ}(r,\, L,\, \mu)$ then $(\exists \ell : E \to \mathcal{V})\; (\exists v : E \to \mathcal{V})\; (\exists \theta : E \to \Phi)$

(R1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$, (R6) if $E = \emptyset$ and $\mu \neq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{ff}$.
(R2) $\lambda(e) = \mathsf{R}^{\mu}[\ell]v_e$
(R3) $\kappa(e) \equiv \theta_e \wedge L{=}\ell_e \wedge \mathsf{Q}_{[\ell]}$,
(R5) $(\forall s)\tau^D(\psi) \equiv \bigwedge_{e\in E\cap D} \theta_e \Rightarrow (L{=}\ell_e \Rightarrow v_e{=}s_e) \Rightarrow \psi[s_e/r]$
$\wedge \bigwedge_{e\in E\setminus D} \theta_e \Rightarrow ((L{=}\ell_e \Rightarrow v_e{=}s_e) \vee (L{=}\ell_e \Rightarrow [\ell]{=}s_e)) \Rightarrow \psi[s_e/r]$
$\wedge (\bigwedge_{e\in E} \neg\theta_e) \Rightarrow \psi[s/r]$,

A.2 Comparison to "Promising Semantics" [POPL 2017]

Jagadeesan et al. [2020] note that 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 their example OOTA4:

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



Under PS, this outcome is allowed by *baiting* with the `else` branch, then *switching* to the `then` branch, based on a coin flip (`if`(b)).[3]

Under all variants of PwT, this outcome is disallowed, due to a cycle in rf ∪ <. (Note that all reads cross threads in the example, so there is no difference between PwT-MCA$_1$ and PwT-MCA$_2$.)

Lochbihler [2013, Fig. 8] notes that such violations of temporal reasoning could result in violations of the Java security architecture.

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 (aka, if-closure) on the value read—effectively, this can turn one read into two, with different conditional branches taken for the two copies of the read. For detailed examples, see [Cho et al. 2021].

PwT invalidates read introduction. In return, PwT enjoys compositionality for temporal safety properties, as discussed in §6 and [Jagadeesan et al. 2020, §6].

This difference highlights the subtle tensions in 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.

A.3 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 $\tau^D(\psi) \equiv \psi[v/r][v/x]$,

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

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

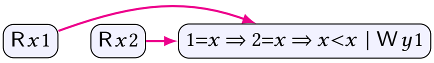
The use of conjunction in R4b′ causes disjunction closure to fail because the predicate transformer $\tau(\psi) = \psi' \wedge \psi''$ does not distribute through disjunction, even assuming that the prime operations do:[4] $\tau(\psi_1 \vee \psi_2) = (\psi_1' \vee \psi_2') \wedge (\psi_1'' \vee \psi_2'') \neq (\psi_1' \wedge \psi_1'') \vee (\psi_2' \wedge \psi_2'') = \tau(\psi_1) \vee \tau(\psi_2)$. See also §4.9.

The substitutions collapse x and r, allowing local invariant reasoning (LIR), as required by JMM causality test case 1, discussed at the end of §4.6. Without Skolemizing it is necessary to substitute $[x/r]$, since the reverse substitution $[r/x]$ is useless when r is bound—compare with §A.4. As discussed below (Downset closure), 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 §A.5 under the semantics of this paper, which elides the middle read event:

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

$$\boxed{\mathsf{W}x0} \qquad \boxed{r \geqslant 0 \mid \mathsf{W}y1}$$

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:

$$\boxed{\mathsf{W}x0} \qquad \boxed{0 \geqslant 0 \mid \mathsf{W}y1}$$

Augmentation of Preconditions. PwP allows augmentation of preconditions. As discussed in §A.6, this causes associativity to fail for delay, at least when attempting to validate Lemma 4.6(i)–(j) Thus, we use *weakest* preconditions, rather than general preconditions. As a result, we fail to validate the following refinement: $\mathcal{P}_1 \not\supseteq \texttt{if}(\phi)\{\mathcal{P}_1\}$.

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 §9.4.

$$\begin{array}{cccc} \texttt{if}(M)\{x := 1\} & \texttt{if}(!M)\{x := 1\} & \texttt{if}(M)\{y := 1\} & \texttt{if}(!M)\{y := 1\} \\ \boxed{M \mid \mathsf{W}x1} & \boxed{\neg M \mid \mathsf{W}x1} & \boxed{M \mid \mathsf{W}y1} & \boxed{\neg M \mid \mathsf{W}y1} \end{array}$$

Associating left and right, we have:

$$\begin{array}{cc} \texttt{if}(M)\{x := 1\};\ \texttt{if}(!M)\{x := 1\} & \texttt{if}(M)\{y := 1\};\ \texttt{if}(!M)\{y := 1\} \\ \boxed{\mathsf{W}x1} & \boxed{\mathsf{W}y1} \end{array}$$

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

Associating into the middle, instead, we require:

$$\begin{array}{ccc} \texttt{if}(M)\{x:=1\} & \texttt{if}(!M)\{x:=1\};\ \texttt{if}(M)\{y:=1\} & \texttt{if}(!M)\{y:=1\} \\ \boxed{M \mid \mathsf{W}x1} & \boxed{\neg M \mid \mathsf{W}x1}\quad \boxed{M \mid \mathsf{W}y1} & \boxed{\neg M \mid \mathsf{W}y1} \end{array}$$

Joining left and right, we have:

$$\begin{array}{c} \texttt{if}(M)\{x:=1\};\ \texttt{if}(!M)\{x:=1\};\ \texttt{if}(M)\{y:=1\};\ \texttt{if}(!M)\{y:=1\} \\ \boxed{\mathsf{W}x1}\quad \boxed{\mathsf{W}y1} \end{array}$$

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

$$\begin{array}{ccc} \texttt{if}(M)\{r:=x\} & y:=r & \texttt{if}(!M)\{s:=x\} \\ \boxed{M \mid \mathsf{R}x1} & \boxed{r{=}1 \mid \mathsf{W}y1} & \boxed{\neg M \mid \mathsf{R}x1} \end{array}$$

Associating left, with causal strengthening:

$$\begin{array}{cc} \texttt{if}(M)\{r:=x\};\ y:=r & \texttt{if}(!M)\{s:=x\} \\ \boxed{M \mid \mathsf{R}x1} \rightarrow \boxed{M \mid \mathsf{W}y1} & \boxed{\neg M \mid \mathsf{R}x1} \end{array}$$

Finally, merging:

$$\begin{array}{c} \texttt{if}(M)\{r:=x\};\ y:=r;\ \texttt{if}(!M)\{s:=x\} \\ \boxed{\mathsf{R}x1} \rightarrow \boxed{M \mid \mathsf{W}y1} \end{array}$$

Instead, associating right:

$$\begin{array}{cc} \texttt{if}(M)\{r:=x\} & y:=r;\ \texttt{if}(!M)\{s:=x\} \\ \boxed{M \mid \mathsf{R}x1} & \boxed{r{=}1 \mid \mathsf{W}y1}\quad \boxed{\neg M \mid \mathsf{R}x1} \end{array}$$

Merging:

$$\begin{array}{c} \texttt{if}(M)\{r:=x\};\ y:=r;\ \texttt{if}(!M)\{s:=x\} \\ \boxed{\mathsf{R}x1} \rightarrow \boxed{\mathsf{W}y1} \end{array}$$

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 the example in §9.2 is:

$$\begin{array}{c} s:=[r];\ x:=s \\ \boxed{r{=}2 \mid \mathsf{R}[2]1} \rightarrow \boxed{(r{=}2 \Rightarrow 1{=}s) \Rightarrow s{=}1 \mid \mathsf{W}x1} \end{array}$$

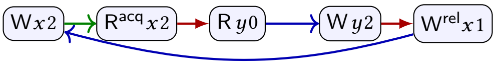
In PwP, we have, instead:

$$\begin{array}{c} s:=[r];\ x:=s \\ \boxed{r{=}2 \mid \mathsf{R}[2]1} \rightarrow \boxed{r{=}2 \land [2]{=}1 \mid \mathsf{W}x1} \end{array}$$

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 ARM-V8 and TSO.

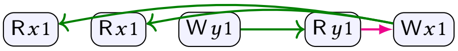
$$x := 2;\ r := x^{\mathsf{acq}};\ s := y \parallel y := 2;\ x^{\mathsf{rel}} := 1$$



We discussed two approaches to this problem in §B.

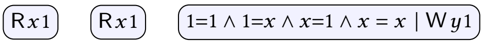
Redundant Read Elimination. Contrary to the claim, redundant read elimination fails for PwP. We discussed redundant read elimination in §9.1. Consider JMM Causality Test Case 2, which we discussed there.

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



Under the semantics of PwP, we have

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


$\mathsf{R}x1$
$\mathsf{R}x1$
$1{=}1 \wedge 1{=}x \wedge x{=}1 \wedge x = x \mid \mathsf{W}y1$

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

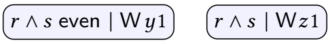
A.4 Substitutions

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 $\tau^D(\psi) \equiv \psi[r/x]$,

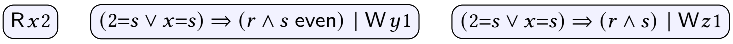
Perhaps surprisingly, this semantics is incomparable with that of Fig. 1. Consider the following:

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


$r \wedge s$ even $\mid \mathsf{W}y1$
$r \wedge s \mid \mathsf{W}z1$

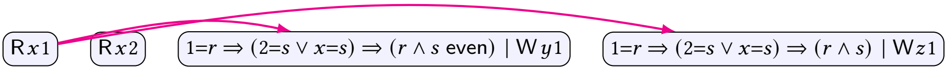
Prepending $(s := x)$, we get the same result regardless of whether we substitute $[s/x]$, since x does not occur in either precondition. Here we show the independent case:

$$s := x;\ \mathtt{if}(r \wedge s\ \mathsf{even})\{y := 1\};\ \mathtt{if}(r \wedge s)\{z := 1\}$$


$\mathsf{R}x2$
$(2{=}s \vee x{=}s) \Rightarrow (r \wedge s$ even$) \mid \mathsf{W}y1$
$(2{=}s \vee x{=}s) \Rightarrow (r \wedge s) \mid \mathsf{W}z1$

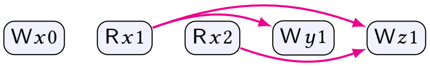
Since the preconditions mention x, prepending $(r := x)$, we now get different results depending on whether we perform the substitution. Without any substitution, we have:

$$r := x;\ s := x;\ \mathtt{if}(r \wedge s\ \mathsf{even})\{y := 1\};\ \mathtt{if}(r \wedge s)\{z := 1\}$$


$\mathsf{R}x1$
$\mathsf{R}x2$
$1{=}r \Rightarrow (2{=}s \vee x{=}s) \Rightarrow (r \wedge s$ even$) \mid \mathsf{W}y1$
$1{=}r \Rightarrow (2{=}s \vee x{=}s) \Rightarrow (r \wedge s) \mid \mathsf{W}z1$

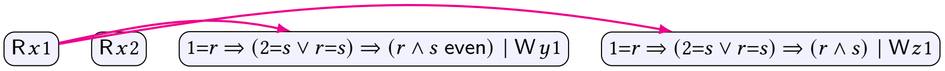
Prepending $(x := 0)$, which substitutes $[0/x]$, the precondition of $(\mathsf{W}y1)$ becomes $(1{=}r \Rightarrow (2{=}s \vee 0{=}s) \Rightarrow (r \wedge s\ \mathsf{even}))$, which is a tautology, whereas the precondition of $\mathsf{W}z1$ becomes $(1{=}r \Rightarrow$

$(2{=}s \lor 0{=}s) \Rightarrow (r \land s))$, 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 $(1{=}r \Rightarrow 2{=}s \Rightarrow (r \land s))$, which is a tautology.

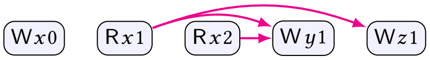


The situation reverses with the substitution $[r/x]$:

$$r := x;\ s := x;\ \mathtt{if}(r \land s \text{ even})\{y := 1\};\ \mathtt{if}(r \land s)\{z := 1\}$$


Rx1
Rx2
1=r ⇒ (2=s ∨ r=s) ⇒ (r ∧ s even) | Wy1
1=r ⇒ (2=s ∨ r=s) ⇒ (r ∧ s) | Wz1

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 we will see is not the case for Arm. By leaving out the substitution, we also ensure that downgraded reads are fulfilled by preceding writes, not reads.

A.5 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 A.2. 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) $(\forall e \in E_2)\ \tau^D_2(e) \equiv \tau^D_1(e)$,
(5) $\checkmark_2 \vDash \checkmark_1$,
(6a) $(\forall d \in E_2)\ (\forall e \in E_2)\ d \leq_2 e$ iff $d \leq_1 e$,
(6b) $(\forall d \in E_1)\ (\forall e \in E_2)$ if $d \leq_1 e$ 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 §4.7. Consider

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

$$\boxed{\mathsf{R}x0} \qquad \boxed{\mathsf{R}y0}$$

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:

$$r := x \qquad\qquad \mathtt{if}(!r)\{s := y\}$$

$$\boxed{\mathsf{R}x0} \qquad\qquad \emptyset$$

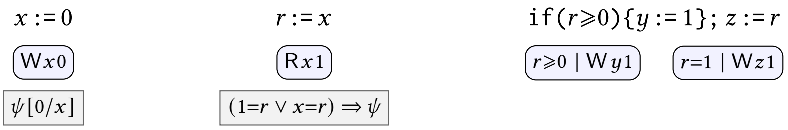
Attempting to get $(\mathsf{R}y0)$, we instead get:

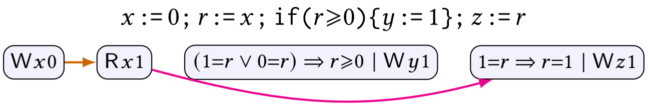
$$r := x \qquad\qquad \mathtt{if}(!r)\{s := y\}$$

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

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

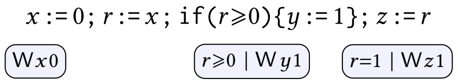
Second, the semantics is not downset closed because the independency reasoning of R4b is only applicable for pomsets where the ignored read is present! Revisiting JMM causality test case 1 from the end of §4.6:



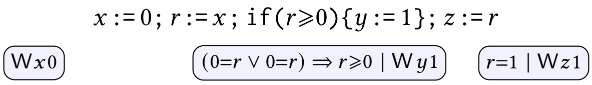


The precondition of $(\mathsf{W}y1)$ is a tautology.

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



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



We leave the elaboration of this idea to future work.

A.6 Logical Encoding of Delay for PwT-MCA

PwT-MCA satisfies one direction of Lemma 4.6(i)–(j)

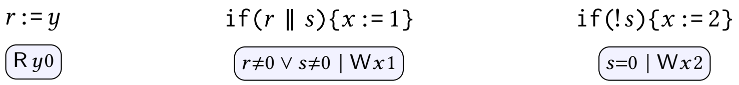
(i) $\mathtt{if}(\phi)\{\mathcal{P}_1\}\mathtt{else}\{\mathcal{P}_2\} \supseteq \mathtt{if}(\phi)\{\mathcal{P}_1\}; \mathtt{if}(\neg\phi)\{\mathcal{P}_2\}$.

(j) $\mathtt{if}(\phi)\{\mathcal{P}_1\}\mathtt{else}\{\mathcal{P}_2\} \supseteq \mathtt{if}(\neg\phi)\{\mathcal{P}_2\}; \mathtt{if}(\phi)\{\mathcal{P}_1\}$.

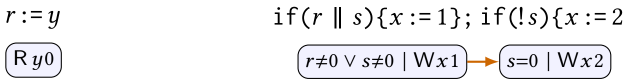
In order to validate the reverse inclusions, we could require that s6a not impose order when $\kappa_1(d) \wedge \kappa_2(e)$ is unsatisfiable. Thus, following on §4.11, we would also like this:

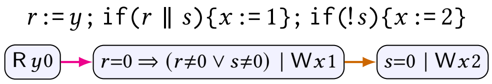
(s6b′) if $\lambda_1(d)$ delays $\lambda_2(e)$ and $\kappa_1(d) \wedge \kappa'_2(e)$ is λ-consistent then $d \leq e$.

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

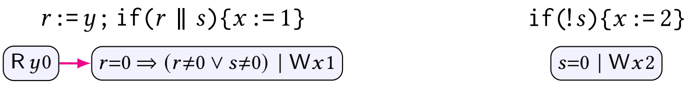


Associating right, order is required since $((r{\neq}0 \vee s{\neq}0) \wedge s{=}0)$ is satisfiable (take $r{=}1$ and $s{=}0$):

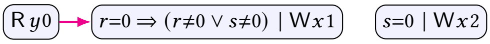




Associating left, order is not required between the writes since $(s{\neq}0 \wedge s{=}0)$ is unsatisfiable:



$$r\mathbin{:=}y;\ \texttt{if}(r \parallel s)\{x\mathbin{:=}1\};\ \texttt{if}(!s)\{x\mathbin{:=}2\}$$



This motivates the logic-based presentation of delay.

In the data model, we require additional symbols: Q_{sc}, $\mathsf{Q}^{x}_{\mathsf{ro}}$, and $\mathsf{Q}^{x}_{\mathsf{wo}}$. We refer to these collectively as *quiescence symbols*.

We update the Def. 4.4 of complete pomset to substitute true for every quiescence symbol:

Definition A.3. A PwT is *complete* if

(c3) $\kappa(e)[\mathsf{tt}/\mathsf{Q}]$ is a tautology, (c5) $\checkmark[\mathsf{tt}/\mathsf{Q}]$ is a tautology.

We define some helper notation:

Definition A.4. Let $\mathsf{Q}^{*}_{\mathsf{ro}} = \bigwedge_{y} \mathsf{Q}^{y}_{\mathsf{ro}}$, and similarly for $\mathsf{Q}^{*}_{\mathsf{wo}}$.
Let formulae $\mathsf{Q}^{\mathsf{S}x}_{\mu}$, $\mathsf{Q}^{\mathsf{L}x}_{\mu}$, and $\mathsf{Q}^{\mathsf{F}}_{\mu}$ be defined:

$$\begin{array}{lll}
\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}} = \mathsf{Q}^{x}_{\mathsf{ro}} \wedge \mathsf{Q}^{x}_{\mathsf{wo}} & \mathsf{Q}^{\mathsf{L}x}_{\mathsf{rlx}} = \mathsf{Q}^{x}_{\mathsf{wo}} & \mathsf{Q}^{\mathsf{F}}_{\mathsf{rel}} = \mathsf{Q}^{*}_{\mathsf{ro}} \wedge \mathsf{Q}^{*}_{\mathsf{wo}} \\
\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rel}} = \mathsf{Q}^{*}_{\mathsf{ro}} \wedge \mathsf{Q}^{*}_{\mathsf{wo}} & \mathsf{Q}^{\mathsf{L}x}_{\mathsf{acq}} = \mathsf{Q}^{x}_{\mathsf{wo}} & \mathsf{Q}^{\mathsf{F}}_{\mathsf{acq}} = \mathsf{Q}^{*}_{\mathsf{ro}} \\
\mathsf{Q}^{\mathsf{S}x}_{\mathsf{sc}} = \mathsf{Q}^{*}_{\mathsf{ro}} \wedge \mathsf{Q}^{*}_{\mathsf{wo}} \wedge \mathsf{Q}_{\mathsf{sc}} & \mathsf{Q}^{\mathsf{L}x}_{\mathsf{sc}} = \mathsf{Q}^{x}_{\mathsf{wo}} \wedge \mathsf{Q}_{\mathsf{sc}} & \mathsf{Q}^{\mathsf{F}}_{\mathsf{sc}} = \mathsf{Q}^{*}_{\mathsf{ro}} \wedge \mathsf{Q}^{*}_{\mathsf{wo}} \wedge \mathsf{Q}_{\mathsf{sc}}
\end{array}$$

Let $[\phi/\mathsf{Q}^{*}_{\mathsf{ro}}]$ substitute ϕ for every $\mathsf{Q}^{y}_{\mathsf{ro}}$, and similarly for $\mathsf{Q}^{*}_{\mathsf{wo}}$.
Let substitutions $[\phi/\mathsf{Q}^{\mathsf{S}x}_{\mu}]$, $[\phi/\mathsf{Q}^{\mathsf{L}x}_{\mu}]$, and $[\phi/\mathsf{Q}^{\mathsf{F}}_{\mu}]$ be defined:

$$\begin{array}{lll}
[\phi/\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}}] = [\phi/\mathsf{Q}^{x}_{\mathsf{wo}}] & [\phi/\mathsf{Q}^{\mathsf{L}x}_{\mathsf{rlx}}] = [\phi/\mathsf{Q}^{x}_{\mathsf{ro}}] & [\phi/\mathsf{Q}^{\mathsf{F}}_{\mathsf{rel}}] = [\phi/\mathsf{Q}^{*}_{\mathsf{wo}}] \\
[\phi/\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rel}}] = [\phi/\mathsf{Q}^{x}_{\mathsf{wo}}] & [\phi/\mathsf{Q}^{\mathsf{L}x}_{\mathsf{acq}}] = [\phi/\mathsf{Q}^{*}_{\mathsf{ro}}, \phi/\mathsf{Q}^{*}_{\mathsf{wo}}] & [\phi/\mathsf{Q}^{\mathsf{F}}_{\mathsf{acq}}] = [\phi/\mathsf{Q}^{*}_{\mathsf{ro}}, \phi/\mathsf{Q}^{*}_{\mathsf{wo}}] \\
[\phi/\mathsf{Q}^{\mathsf{S}x}_{\mathsf{sc}}] = [\phi/\mathsf{Q}^{x}_{\mathsf{wo}}, \phi/\mathsf{Q}_{\mathsf{sc}}] & [\phi/\mathsf{Q}^{\mathsf{L}x}_{\mathsf{sc}}] = [\phi/\mathsf{Q}^{*}_{\mathsf{ro}}, \phi/\mathsf{Q}^{*}_{\mathsf{wo}}, \phi/\mathsf{Q}_{\mathsf{sc}}] & [\phi/\mathsf{Q}^{\mathsf{F}}_{\mathsf{sc}}] = [\phi/\mathsf{Q}^{*}_{\mathsf{ro}}, \phi/\mathsf{Q}^{*}_{\mathsf{wo}}, \phi/\mathsf{Q}_{\mathsf{sc}}]
\end{array}$$

Update the following rules from Fig. 1. (The change is similar for address calculation and if-closure.)

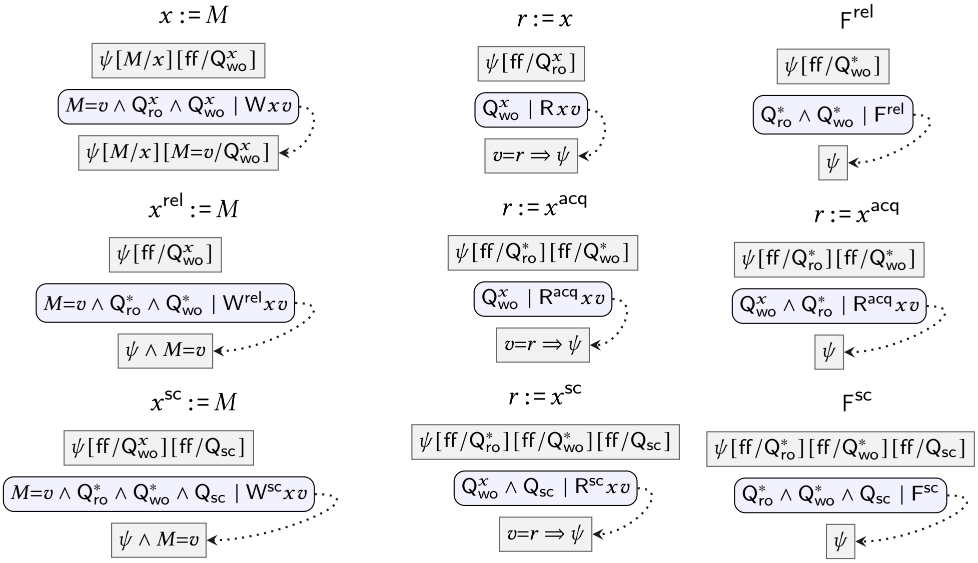
(w3) $\kappa(e) \equiv M{=}v \wedge \mathsf{Q}^{\mathsf{S}x}_{\mu}$,
(w4a) if $E \neq \emptyset$ then $\tau^{D}(\psi) \equiv \psi[M/x][M{=}v/\mathsf{Q}^{\mathsf{S}x}_{\mu}]$,
(w4b) if $E = \emptyset$ then $\tau^{D}(\psi) \equiv \psi[M/x][\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_{\mu}]$,
(R3) $\kappa(e) \equiv \mathsf{Q}^{\mathsf{L}x}_{\mu}$,
(R4a) if $E \neq \emptyset$ and $(E \cap D) \neq \emptyset$ then $\tau^{D}(\psi) \equiv v{=}r \Rightarrow \psi$,
(R4b) if $E \neq \emptyset$ and $(E \cap D) = \emptyset$ then $\tau^{D}(\psi) \equiv (v{=}r \vee x{=}r) \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_{\mu}]$,
(R4c) if $E = \emptyset$ then $\tau^{D}(\psi) \equiv \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_{\mu}]$,
(F3) $\kappa(e) \equiv \mathsf{Q}^{\mathsf{F}x}_{\mu}$,
(F4a) if $E \neq \emptyset$ then $\tau^{D}(\psi) \equiv \psi$,
(F4b) if $E = \emptyset$ then $\tau^{D}(\psi) \equiv \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{F}x}_{\mu}]$.

The quiescence formulae indicate what must precede an event. For example, all preceding accesses must be ordered before a releasing write, whereas only writes on x must be ordered before a releasing read on x.

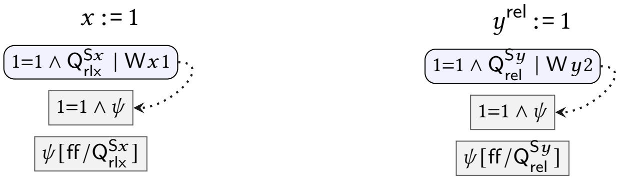
The quiescence substitutions update quiescence symbols in subsequent code. For subsequent independent code, w3 and R3 substitute false. In complete pomsets, we substitute true for . For example, we substitute ff for $\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rel}}$ in the independent case for a releasing write; this ensures that subsequent writes to x follow the releasing write in top-level pomsets. Similarly, we substitute ff for $\mathsf{Q}^{\mathsf{L}x}_{\mathsf{acq}}$ in the independent case for an acquiring write; this ensures that all subsequent accesses follow the acquiring read in top-level pomsets.

[Todo: Fix these examples]

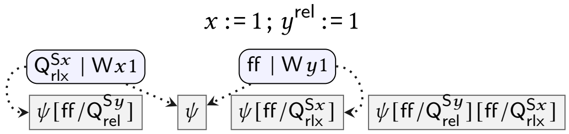
Example A.5. The following pomsets show the effect of quiescence for each access mode.


$x := M$
$\psi[M/x][\mathsf{ff}/\mathsf{Q}^x_{\mathsf{wo}}]$
$M{=}v \land \mathsf{Q}^x_{\mathsf{ro}} \land \mathsf{Q}^x_{\mathsf{wo}} \mid \mathsf{W}\,x\,v$
$\psi[M/x][M{=}v/\mathsf{Q}^x_{\mathsf{wo}}]$
$r := x$
$\psi[\mathsf{ff}/\mathsf{Q}^x_{\mathsf{ro}}]$
$\mathsf{Q}^x_{\mathsf{wo}} \mid \mathsf{R}\,x\,v$
$v{=}r \Rightarrow \psi$
$\mathsf{F}^{\mathsf{rel}}$
$\psi[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}]$
$\mathsf{Q}^*_{\mathsf{ro}} \land \mathsf{Q}^*_{\mathsf{wo}} \mid \mathsf{F}^{\mathsf{rel}}$
ψ
$x^{\mathsf{rel}} := M$
$\psi[\mathsf{ff}/\mathsf{Q}^x_{\mathsf{wo}}]$
$M{=}v \land \mathsf{Q}^*_{\mathsf{ro}} \land \mathsf{Q}^*_{\mathsf{wo}} \mid \mathsf{W}^{\mathsf{rel}}x\,v$
$\psi \land M{=}v$
$r := x^{\mathsf{acq}}$
$\psi[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{ro}}][\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}]$
$\mathsf{Q}^x_{\mathsf{wo}} \mid \mathsf{R}^{\mathsf{acq}}x\,v$
$v{=}r \Rightarrow \psi$
$r := x^{\mathsf{acq}}$
$\psi[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{ro}}][\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}]$
$\mathsf{Q}^x_{\mathsf{wo}} \land \mathsf{Q}^*_{\mathsf{ro}} \mid \mathsf{R}^{\mathsf{acq}}x\,v$
ψ
$x^{\mathsf{sc}} := M$
$\psi[\mathsf{ff}/\mathsf{Q}^x_{\mathsf{wo}}][\mathsf{ff}/\mathsf{Q}_{\mathsf{sc}}]$
$M{=}v \land \mathsf{Q}^*_{\mathsf{ro}} \land \mathsf{Q}^*_{\mathsf{wo}} \land \mathsf{Q}_{\mathsf{sc}} \mid \mathsf{W}^{\mathsf{sc}}x\,v$
$\psi \land M{=}v$
$r := x^{\mathsf{sc}}$
$\psi[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{ro}}][\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}][\mathsf{ff}/\mathsf{Q}_{\mathsf{sc}}]$
$\mathsf{Q}^x_{\mathsf{wo}} \land \mathsf{Q}_{\mathsf{sc}} \mid \mathsf{R}^{\mathsf{sc}}x\,v$
$v{=}r \Rightarrow \psi$
F^{sc}
$\psi[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{ro}}][\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}][\mathsf{ff}/\mathsf{Q}_{\mathsf{sc}}]$
$\mathsf{Q}^*_{\mathsf{ro}} \land \mathsf{Q}^*_{\mathsf{wo}} \land \mathsf{Q}_{\mathsf{sc}} \mid \mathsf{F}^{\mathsf{sc}}$
ψ

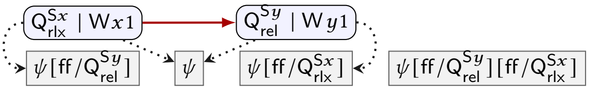
Example A.6. The definition enforces publication. Consider:


$x := 1$
$1{=}1 \land \mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}} \mid \mathsf{W}\,x\,1$
$1{=}1 \land \psi$
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}}]$
$y^{\mathsf{rel}} := 1$
$1{=}1 \land \mathsf{Q}^{\mathsf{S}y}_{\mathsf{rel}} \mid \mathsf{W}\,y\,2$
$1{=}1 \land \psi$
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}y}_{\mathsf{rel}}]$

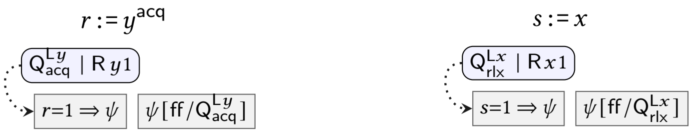
Since $\mathsf{Q}^{\mathsf{S}y}_{\mathsf{rel}}[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}}]$ is ff, composing these without order simplifies to:


$x := 1;\ y^{\mathsf{rel}} := 1$
$\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}} \mid \mathsf{W}\,x\,1$
$\mathsf{ff} \mid \mathsf{W}\,y\,1$
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}y}_{\mathsf{rel}}]$
ψ
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}}]$
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}y}_{\mathsf{rel}}][\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}}]$

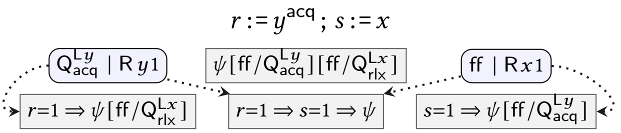
In order to get a satisfiable precondition for $(\mathsf{W}\,y\,1)$, we must introduce order:


$\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}} \mid \mathsf{W}\,x\,1$
$\mathsf{Q}^{\mathsf{S}y}_{\mathsf{rel}} \mid \mathsf{W}\,y\,1$
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}y}_{\mathsf{rel}}]$
ψ
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}}]$
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}y}_{\mathsf{rel}}][\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}}]$

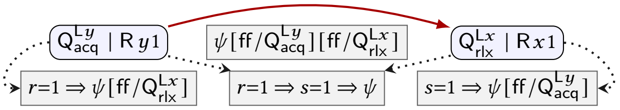
Example A.7. The definition enforces subscription. Consider:


$r := y^{\mathsf{acq}}$
$\mathsf{Q}^{\mathsf{L}y}_{\mathsf{acq}} \mid \mathsf{R}\,y\,1$
$r{=}1 \Rightarrow \psi$
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}y}_{\mathsf{acq}}]$
$s := x$
$\mathsf{Q}^{\mathsf{L}x}_{\mathsf{rlx}} \mid \mathsf{R}\,x\,1$
$s{=}1 \Rightarrow \psi$
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_{\mathsf{rlx}}]$

Since $\mathsf{Q}^{\mathsf{L}x}_{\mathsf{rlx}}[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}y}_{\mathsf{acq}}]$ is ff, composing these without order simplifies to:


$r := y^{\mathsf{acq}};\ s := x$
$\mathsf{Q}^{\mathsf{L}y}_{\mathsf{acq}} \mid \mathsf{R}\,y\,1$
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}y}_{\mathsf{acq}}][\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_{\mathsf{rlx}}]$
$\mathsf{ff} \mid \mathsf{R}\,x\,1$
$r{=}1 \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_{\mathsf{rlx}}]$
$r{=}1 \Rightarrow s{=}1 \Rightarrow \psi$
$s{=}1 \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}y}_{\mathsf{acq}}]$

In order to get a satisfiable precondition for $(\mathsf{R}x1)$, we must introduce order:


$\mathsf{Q}^{\mathsf{L}y}_{\mathsf{acq}} \mid \mathsf{R}\,y1$
$\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}y}_{\mathsf{acq}}][\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_{\mathsf{rlx}}]$
$\mathsf{Q}^{\mathsf{L}x}_{\mathsf{rlx}} \mid \mathsf{R}\,x1$
$r{=}1 \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_{\mathsf{rlx}}]$
$r{=}1 \Rightarrow s{=}1 \Rightarrow \psi$
$s{=}1 \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}y}_{\mathsf{acq}}]$

Example A.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.

$$\begin{array}{cccc} \texttt{if}(r)\{x:=2\} & x:=1 & x:=2 & \texttt{if}(!r)\{x:=1\} \\ \boxed{r{\neq}0 \mid \mathsf{W}x2} & \boxed{\mathsf{W}x1} & \boxed{\mathsf{W}x2} & \boxed{r{=}0 \mid \mathsf{W}x1} \end{array}$$

Augmenting the middle preconditions and then using sequential composition, we have:

$$\begin{array}{ccc} \texttt{if}(r)\{x:=2\} & x:=1;\ x:=2 & \texttt{if}(!r)\{x:=1\} \\ \boxed{r{\neq}0 \mid \mathsf{W}x2} & \boxed{r{\neq}0 \mid \mathsf{W}x1}\quad\boxed{r{=}0 \mid \mathsf{W}x2} & \boxed{r{=}0 \mid \mathsf{W}x1} \end{array}$$

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

$$\begin{array}{c} \texttt{if}(r)\{x:=2\};\ x:=1;\ x:=2;\ \texttt{if}(!r)\{x:=1\} \\ \boxed{\mathsf{W}x2} \rightarrow \boxed{\mathsf{W}x1} \end{array}$$

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

$$\begin{array}{c} y:=1;\ r:=y;\ \texttt{if}(r)\{x:=2\};\ x:=1;\ x:=2;\ \texttt{if}(!r)\{x:=1\} \\ \boxed{\mathsf{W}\,y1} \rightarrow \boxed{\mathsf{R}\,y1}\quad \boxed{\mathsf{W}x2} \rightarrow \boxed{\mathsf{W}x1} \end{array}$$

It is for this reason that we use *weakest* preconditions, rather than preconditions.

B LOWERING PwT-MCA TO ARM

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

B.1 Arm executions

Our description of ARM-V8 follows Alglave et al. [2021], adapting the notation to our setting.

Definition B.1. An ARM-V8 *execution graph*, G, is tuple $(E, \lambda, \mathsf{poloc}, \mathsf{lob})$ such that

(A1) $E \subseteq \mathcal{E}$ is a set of events,
(A2) $\lambda : E \longrightarrow \mathcal{A}$ 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 $\ltimes_{\mathsf{sync}}$, 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 B.2. Execution G is (co, rf, gcb)*-valid*, under *External Global Consistency* (EGC) if

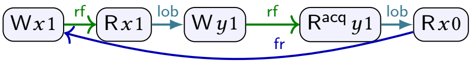
(A5) co $\subseteq E \times E$, is a per-location total order on writes, capturing *coherence*,
(A6) rf $\subseteq E \times E$, is a relation, capturing *reads-from*, such that
  (A6a) rf is surjective and injective relation on $\{e \in E \mid \lambda(e)$ is a read$\}$,
  (A6b) if $d \xrightarrow{\mathsf{rf}} e$ then $\lambda(d)$ matches $\lambda(e)$,
  (A6c) poloc ∪ co ∪ rf ∪ fr is acyclic, where $e \xrightarrow{\mathsf{fr}} c$ if $e \xleftarrow{\mathsf{rf}} d \xrightarrow{\mathsf{co}} c$, for some d,
(A7) gcb ⊇ (co ∪ 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 (co, rf, cb)*-valid* under *External Consistency* (EC) if

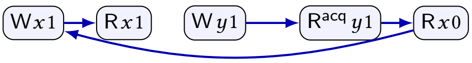
(A5) and (A6), as for EGC,
(A8) cb ⊇ (co ∪ lob) is a linear order such that if $d \xrightarrow{\mathsf{rf}} e$ then either
  (A8a) $d \xrightarrow{\mathsf{cb}} e$ and if $\lambda(c)$ blocks $\lambda(e)$ then either $c \xrightarrow{\mathsf{cb}} d$ or $e \xrightarrow{\mathsf{cb}} c$, or
  (A8b) $d \xleftarrow{\mathsf{cb}} e$ 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 ARM-V8. They explain EGC and EC using the following example, which is allowed by ARM-V8.[5]

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

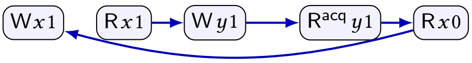


EGC drops lob-order in the first thread using A7b, since (Wx1) is not lob-ordered before (Wy1).



(gcb)

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



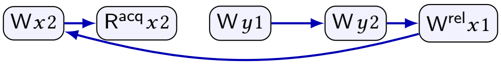
(cb)

B.2 Lowering PwT-MCA1 to Arm

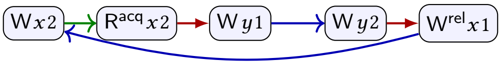
The optimal lowering for ARM-V8 is unsound for PwT-MCA$_1$. The optimal lowering maps relaxed access to `ldr`/`str` and non-relaxed access to `ldar`/`stlr` [Podkopaev et al. 2019]. In this section, we consider a suboptimal strategy, which lowers non-relaxed reads to (`dmb.sy`; `ldar`). Significantly, 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)



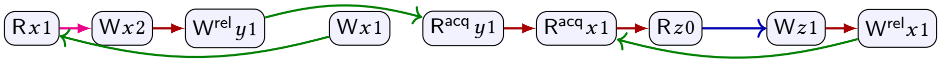
(≤)

This attempted execution is allowed by ARM-V8, but disallowed by our semantics.

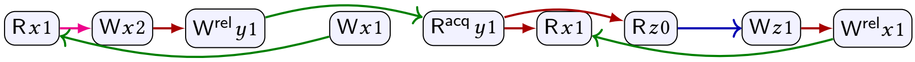
[5] We have changed an address dependency in the first thread to a data dependency.

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 B.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 ARM-V8.

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

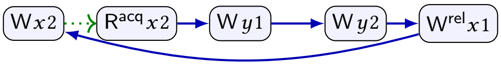
PROOF. By way of contradiction, assume $e \xrightarrow{\mathsf{gcb}} c$. If $c \xrightarrow{\mathsf{rf}} e$ then by A7 we must also have $c \xrightarrow{\mathsf{gcb}} e$, contradicting the assumption that gcb is a total order. Otherwise that there is some $d \neq c$ such that $d \xrightarrow{\mathsf{rf}} e$, 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 ARM-V8. 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 W5, any dependence on the read can by satisfied without introducing order in S3. □

B.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 $d \leq e$. 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 B.6. *Suppose G is EC-valid via $(\mathsf{co}, \mathsf{rf}, \mathsf{cb})$. Then there a permutation cb' of cb such that G is EC-valid via $(\mathsf{co}, \mathsf{rf}, \mathsf{cb}')$ and $\mathsf{cb}' \supseteq \mathsf{fr}$, 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.

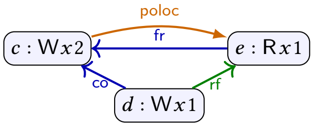
If A8a applies to $d \xrightarrow{\mathsf{rf}} e$, then $e \xrightarrow{\mathsf{cb}} c$, since it cannot be that $c \xrightarrow{\mathsf{co}} d$.

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 B.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 ARM-V8. M7b holds since cb_1 is consistent with co_1 and fr_1. s6b follows from A8b. As noted above, lob includes the order required by s3 and s6a. □

C LDRF-SC FOR PwT-MCA

In this appendix, we establish a 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 §A.3).
- PwP allows preconditions that are stronger than the weakest precondition.
- PwP imposes M7c (rf implies $\leq$) 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 7.2 to correct this deficiency.

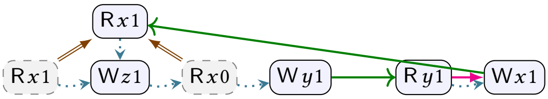
The first two items require us to define gen differently, below.

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.

Program Order. Let $[\![\cdot]\!]^{\mathsf{po}}_{\mathsf{mca2}}$ be defined by applying the construction of Lemma 7.2 to $[\![\cdot]\!]_{\mathsf{mca2}}$. We consider only *complete* pomsets. For these, we derive program order on compound events as follows. By Lemma 7.4, 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 C.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:

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



Data Race. Data races are defined using *program* order (po), not *pomset* order ($\le$).

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. 5.1) and P' is a *downset* of P (Def. A.2).

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 $\mathrm{gen}[\![S]\!] = \{P \in \nabla[\![S]\!]^{\mathsf{po}}_{\mathsf{mca2}} \mid P \text{ is augmentation-minimal in } \nabla[\![S]\!]^{\mathsf{po}}_{\mathsf{mca2}}\}$.

Extensions. We say that P' *S-extends* P if $P \neq P' \in \mathrm{gen}[\![S]\!]$ 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) $P \in \mathrm{gen}[\![S]\!]$, (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.

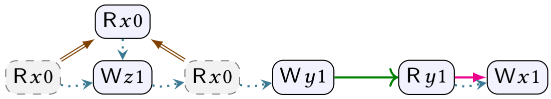
Sequentiality. Let $\lessdot_L = <_L \cup \mathsf{po}$, where $<_L$ is the restriction of $<$ to events that access locations in L. We say that P' is *L-sequential after* P if (1) P' is po-convex, (2) $\lessdot_L$ is acyclic in $E' \setminus E$.

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. 7.1).

LEMMA C.1. *Suppose $P' \in \mathrm{gen}[\![S]\!]$ 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 $P'' \in \mathrm{gen}[\![S]\!]$.*

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:

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



Theorem C.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 C.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 $\leq_*$ (the transitive reduction $\leq$) can be decomposed into three disjoint relations. Let $\mathsf{ppo} = (\leq_* \cap \mathsf{po})$ denote *preserved* program order, as required by sequential composition and conditional. The other two relations are cross-thread subsets of $(\leq_* \setminus \mathsf{po})$: rfe (reads-from-external) orders writes before reads, satisfying P6; cae (coherence-after-external) orders read and write accesses before writes, satisfying M7b. (Within a thread, S6 induces order that is included in ppo.)

Using this decomposition, we can show the following.

Lemma C.3. *Suppose $P'' \in gen[\![S]\!]$ has an external read $d \xrightarrow{\mathsf{rf}''} e$ that is maximal in $(\mathsf{ppo} \cup \mathsf{rfe})$. 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.

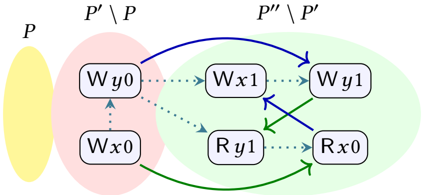
We can prove a similar result for $(\mathsf{po} \cup \mathsf{rfe})$-maximal read and write accesses.

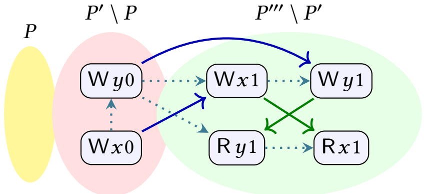
Turning to the proof of the theorem, if P'' is L-sequential after P, then the result follows from (1). Otherwise, there must be a $<_L$ cycle in P'' involving all of the actions in $(E'' \setminus E')$: If there were no such cycle, then P'' would be L-sequential; if there were elements outside the cycle,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

If there is a $(\mathsf{po} \cup \mathsf{rfe})$-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 $(\mathsf{W}x1)$.

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



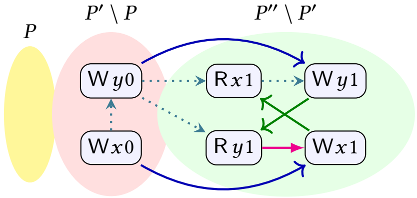


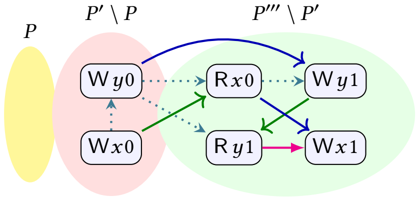
It is important that e be (po ∪ rfe)-maximal, not just (ppo ∪ rfe)-maximal. The latter criterion would allow us to choose e to be the read of y, but then there would be no e-similar pomset: if an execution reads 0 for y then there is no read of x, due to the conditional.

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. □

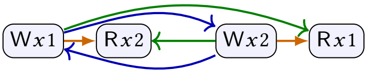
D PwT-MCA: ADDITIONAL EXAMPLES

This appendix includes additional examples. They all apply equally to PwT-MCA$_1$ and PwT-MCA$_2$. Several of these are taken directly from [Jagadeesan et al. 2020].

D.1 Coherence

The following execution is disallowed by fulfillment.

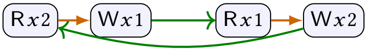
$$x := 1;\ r := x \parallel x := 2;\ s := x$$



(COH)

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

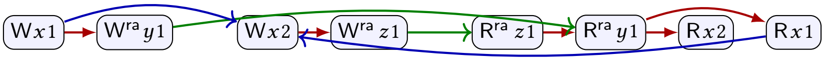
$$r := x;\ x := 1 \parallel s := x;\ x := 2$$



(TC16)

We also forbid the following, which Java allows:

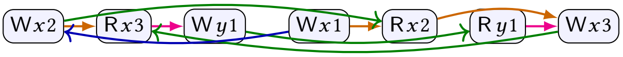
$$x := 1;\ y^{\mathsf{ra}} := 1 \parallel x := 2;\ z^{\mathsf{ra}} := 1 \parallel r := z^{\mathsf{ra}};\ r := y^{\mathsf{ra}};\ r := x;\ r := x$$



(CO3)

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

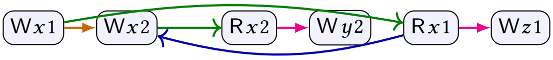
$$x := 2;\ \mathtt{if}(x{\neq}2)\{y := 1\} \parallel x := 1;\ r := x;\ \mathtt{if}(y)\{x := 3\}$$



(COH-CYC)

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

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



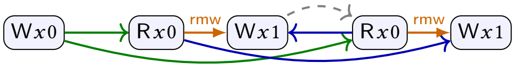
(CO2)

Here, the reader sees 2 then 1, although they are written in the reverse order. This behavior is allowed by Java in order to validate CSE without requiring aliasing analysis.

D.2 RMWs

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

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

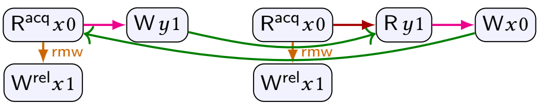


(RMW0)

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:

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

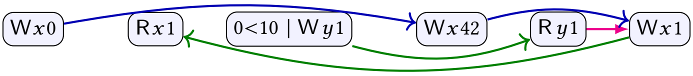


(CDRF)

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:

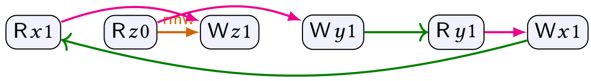
$$x := 0;\ \left(r := \mathsf{CAS}^{\mathsf{rlx,rlx}}(x, 0, 1);\ \mathtt{if}(r{<}10)\{y := 1\} \parallel x := 42;\ x := y\right)$$



(GA+E)

PS also does not support register promotion, as modeled by RP below. Our semantics permits RP:

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

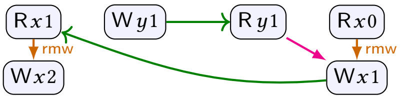


(RP)

These following examples are from [Cho et al. 2021].

CDRF shows that PwT semantics is not too permissive for ra-RMWs. But what about rlx-RMWs. The following execution is allowed by ARM-V8, and PS2.0, but disallowed by PS2.1.

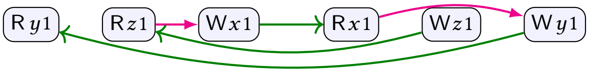
$$r := \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x, 1)\,;\ y := 1 \parallel r := y\,;\ s := \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x, r)$$



(RMW-W)

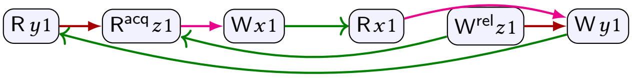
If this $\{z\}$-DRF-RA?

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



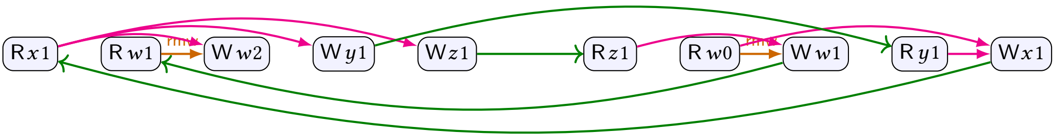
(NAIVE-LDRF-RA-FAIL)

Interpreting $\{z\}$ as ra:



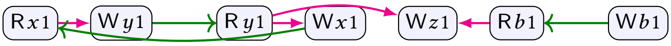
PwT disallows LDRF-FAIL-PS, which is similar to OOTA4.

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



(LDRF-FAIL-PS)

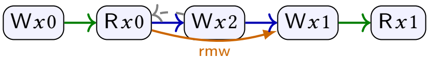
$$y := x \parallel r := y\,;\ \mathtt{if}(b)\{x := r\,;\ z := r\}\,\mathtt{else}\,\{x := 1\} \parallel b := 1$$



(OOTA4)

Example D.1. This definition ensures atomicity, disallowing executions such as [Podkopaev et al. 2019, Ex. 3.2]:

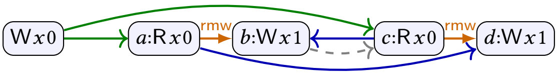
$$x := 0\,;\ \mathsf{INC}^{\mathsf{rlx},\mathsf{rlx}}(x) \parallel x := 2\,;\ r := x$$



By M10c(i), since (Wx2) $\longrightarrow$ (Wx1), it must be that (Wx2) $\longrightarrow$ (Rx0), creating a cycle.

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

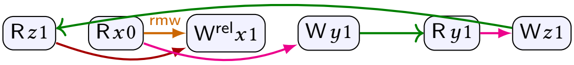
$$x := 0;\ (\texttt{INC}^{\mathsf{rlx,rlx}}(x) \parallel \texttt{INC}^{\mathsf{rlx,rlx}}(x))$$



The order from read-to-write is required by fulfillment. Apply M10c(i) of the second RMW to $a \longrightarrow d$, we have that $a \longrightarrow c$. Subsequently applying M10c(ii) of the first RMW, we have $b \longrightarrow c$, creating a cycle.

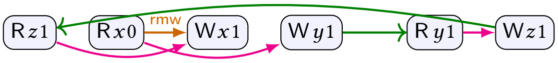
Example D.3. By using two actions rather than one, the definition allows examples such as the following, which is allowed by ARM-v8 [Podkopaev et al. 2019, Ex. 3.10]:

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



A similar example, also allowed by ARM-v8 [Chakraborty and Vafeiadis 2019, Fig. 6]:

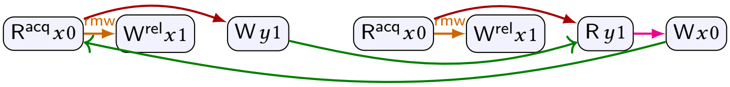
$$r := z;\ s := \texttt{FADD}^{\mathsf{rlx,rlx}}(x, r);\ y := s{+}1 \parallel r := y;\ z := r$$



This is allowed by WEAKESTMO, but not PS.

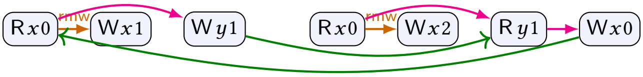
Example D.4. Consider the CDRF example from [Lee et al. 2020]:

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



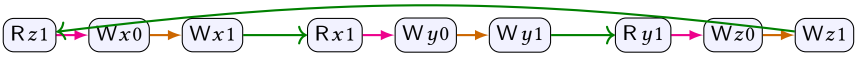
Example D.5. Consider this example from [Lee et al. 2020, §C]:

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

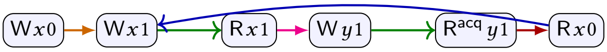


D.3 MCA

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



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



These candidate executions are invalid, due to cycles.