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). 2022. The Leaky Semicolon: Compositional Semantic Dependencies for Relaxed-Memory Concurrency. *Proc. ACM Program. Lang.* 0, POPL, Article 0 (January 2022), 29 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;

2475-1421/2022/1-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 fail to validate temporal safety properties (see §??).

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, targeting the Intermediate Memory Model (IMM) [Podkopaev et al. 2019]. Jagadeesan et al. [2020] defined Pomsets with Preconditions (PwP), which use preconditions and logic to calculate dependencies for a Java-like language targeting multicopy-atomic (MCA) hardware, such as Arm8 [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, adding an event requires 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 and discussion of related work, we define sequential dependencies in §4. We then 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]; §6 summarizes the results for this model. In §7, we define PwT-C11, which models C11, adapting the approach of Paviotti et al. [2020]; §8 describes a tool for automatic evaluation of litmus tests. In §9, we extend the semantics to include additional features, such as address calculation and RMWs.

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) $wp_{\texttt{if(}M\texttt{)\{}S_1\texttt{\}else\{}S_2\texttt{\}}}(\psi) = \ ((M{\neq}0) \Rightarrow wp_{S_1}(\psi)) \wedge ((M{=}0) \Rightarrow wp_{S_2}(\psi))$

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

(D2a) $wp_{x:=M}(\psi) = \psi[M/x]$
(D2b) $wp_{r:=x}(\psi) = \psi[x/r]$

is unsound! In particular, a read from a shared memory location may be fulfilled by a write in another thread, invalidating D2b. (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:

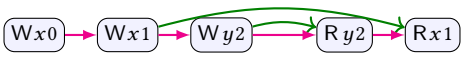
(1) it should not impose too much order, overconstraining the implementation,
(2) it should not impose too little order, allowing bogus executions, and
(3) it should be *compositional* and *direct*, as described in §1.

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

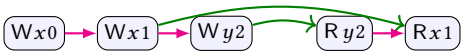
For single-threaded programs, memory can be thought of as you might expect: programs write to, and read from, memory references. This can be thought of as a total order of reads and writes (pink arrows →), where each read has a matching *fulfilling* write (green arrows ⟶), for example:

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



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 ||):

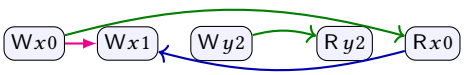
$$x := 0\,;\; x := 1\,;\; y := 2 \parallel r := y\,;\; s := x$$



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 $(\mathsf{R}x1)$ to $(\mathsf{W}y1)$, and the data dependency from $(\mathsf{R}y1)$ to $(\mathsf{W}x1)$. If either is missing, then this execution is acyclic and hence allowed. For example dropping the control dependency results in:

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

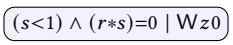
While syntactic dependency calculation suffices for hardware models, it is not preserved by common compiler optimizations. For example, if we calculate control dependencies syntactically, then there is a dependency from $(\mathsf{R}x1)$ to $(\mathsf{W}y1)$, and therefore a cycle in, the candidate execution:

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

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

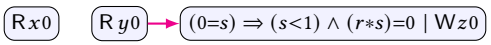
To address this, Jagadeesan et al. [2020] introduced *Pomsets with Preconditions (*PwP*)*, where events are labeled with logical formulae. Nontrivial preconditions are introduced by store actions (modeling data dependencies) and conditionals (modeling control dependencies):

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

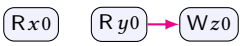


Preconditions are discharged by being ordered after a read (we assume the usual precedence for logical operators—$\neg$, $\wedge$, $\vee$, $\Rightarrow$):

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



Note that there is dependency order from $(\mathsf{R}y0)$ to $(\mathsf{W}z0)$ so the precondition for $(\mathsf{W}z0)$ only has to be satisfied assuming the hypothesis $(0{=}s)$. There is no matching order from $(\mathsf{R}x0)$ to $(\mathsf{W}z0)$ which is why we do not assume the hypothesis $(0{=}r)$. Nonetheless, the precondition on $(\mathsf{W}z0)$ 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,

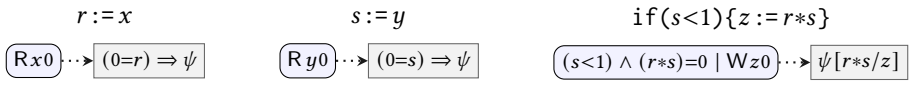
$$[\![x := M \,;\, y := N \,;\, S]\!] = [\![y := N \,;\, x := M \,;\, S]\!] \;\text{ if } x \neq y$$

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

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

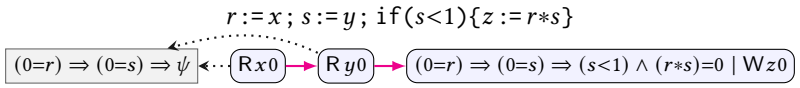
Then the equivalence holds in any context.

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 D2a. For the reads, however, D2b 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.)

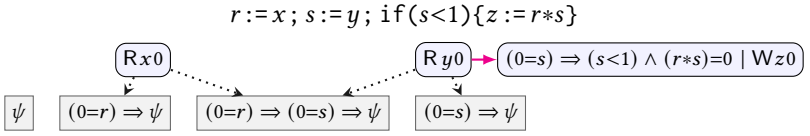


This model works for the sequentially consistent case, but needs to be weakened for the relaxed case. The key observation of this paper is that rather than working with one predicate transformer, we should work with a *family* of predicate transformers, indexed by sets of events.

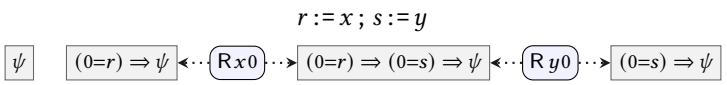
For example, for single-event pomsets, there are two predicate transformers, since there are two subsets of any one-element set. The *independent* transformer is indexed by the empty set, whereas the *dependent* transformer is indexed by the singleton. We visualize this by including more than one transformed 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.

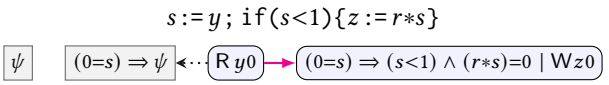
For example, we can recover the expected semantics for (†) by choosing the predicate transformer which is independent of $(\mathsf{R}x0)$ but dependent on $(\mathsf{R}y0)$, which is the transformer which maps ψ to $(0{=}s) \Rightarrow \psi$.



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 §??.

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

Other studies of relaxed memory can be categorized by their approach to dependency calculation. Hardware models use syntactic dependencies [Alglave et al. 2014]. Many software models do not bother with dependencies at all [Batty et al. 2011; Cox 2016; Watt et al. 2020, 2019]. Others have strong dependencies that disallow compiler optimizations and efficient implementation, typically requiring fences for every relaxed read on 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 §??.

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 (§4.4–4.9). 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 are finite[1] and 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.

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

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

Access modes, μ, are relaxed (rlx), release (rel), acquire (acq), and sequentially consistent (sc). Let expressions $(r := M)$ only affect thread-local state and thus do not have a mode. Reads $(r := [L]^{\mu})$ support rlx, acq, sc. Writes $([L]^{\mu} := r)$ support rlx, rel, sc. Fences (F^{μ}) support rel, acq, sc. In examples, the default mode for reads and writes is rlx—we systematically drop the annotation.

Commands, aka *statements*, S, include memory accesses at a given mode, as well as the usual structural constructs. Following [Ferreira et al. 1996], $\Rrightarrow$ denotes parallel composition, preserving thread state on the right after a join. In examples and sublanguages without join, we use the symmetric $\|$ 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 $y := \mathbb{M}\,;\, S$ is shorthand for $r := x\,;\, y := \mathbb{M}[r/x]\,;\, S$. Each occurrence of x in an extended expression corresponds to an separate read. We also write $\mathtt{if}(M)\{S\}$ as shorthand for $\mathtt{if}(M)\{S\}\,\mathtt{else}\,\{\mathtt{skip}\}$.

Throughout §1–8 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 $\mathcal{A} \times \mathcal{A}$: *delays* and *matches* $\subseteq$ *blocks* $\subseteq$ *overlaps*.

[1] We require finiteness for the semantics of address calculation (§9.5), which quantifies over all values. Using types, one could limit the finiteness assumption to the subset of values used for address calculation.

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:

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

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

Definition 4.1. Actions (R) are *read* actions.
We say a *matches* b if $a = (\mathsf{W} x v)$ and $b = (\mathsf{R} x v)$.
We say a *blocks* b if $a = (\mathsf{W} x)$ and $b = (\mathsf{R} x)$, regardless of value.
We say a *overlaps* b if they access the same location, regardless of whether they read or write.
Let $\bowtie_{\mathsf{co}}$ capture write-write, read-write coherence: $\bowtie_{\mathsf{co}} = \{(\mathsf{W}x, \mathsf{W}x),\ (\mathsf{R}x, \mathsf{W}x),\ (\mathsf{W}x, \mathsf{R}x)\}$.
Let $\ltimes_{\mathsf{sync}}$ capture conflict due to synchronization:[2] $\ltimes_{\mathsf{sync}} = \{(a, \mathsf{W}^{\sqsupseteq \mathsf{rel}}),\ (a, \mathsf{F}^{\sqsupseteq \mathsf{rel}}),\ (\mathsf{R}, \mathsf{F}^{\sqsupseteq \mathsf{acq}}),\ (\mathsf{R}^{\sqsupseteq \mathsf{acq}}, b),\ (\mathsf{F}^{\sqsupseteq \mathsf{acq}}, b),\ (\mathsf{F}^{\sqsupseteq \mathsf{rel}}, \mathsf{W}),\ (\mathsf{W}^{\sqsupseteq \mathsf{rel}} x, \mathsf{W}x)\}$.
Let $\bowtie_{\mathsf{sc}}$ capture conflict due to sc access: $\bowtie_{\mathsf{sc}} = \{(\mathsf{W}^{\mathsf{sc}}, \mathsf{W}^{\mathsf{sc}}),\ (\mathsf{R}^{\mathsf{sc}}, \mathsf{W}^{\mathsf{sc}}),\ (\mathsf{W}^{\mathsf{sc}}, \mathsf{R}^{\mathsf{sc}}),\ (\mathsf{R}^{\mathsf{sc}}, \mathsf{R}^{\mathsf{sc}})\}$.
We say a *delays* b if $a \bowtie_{\mathsf{co}} b$ or $a \ltimes_{\mathsf{sync}} b$ or $a \bowtie_{\mathsf{sc}} b$.

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 $\tau : \Phi \to \Phi$ such that
(x1) $\tau(\psi_1 \wedge \psi_2) \equiv \tau(\psi_1) \wedge \tau(\psi_2)$,
(x2) $\tau(\psi_1 \vee \psi_2) \equiv \tau(\psi_1) \vee \tau(\psi_2)$,
(x3) if $\phi \vDash \psi$, then $\tau(\phi) \vDash \tau(\psi)$.

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

As discussed in §1, predicate transformers suffice for sequentially consistent models, but not relaxed models, where dependency calculation is crucial. For dependency calculation, we use a *family* of predicate transformers, indexed by sets of events. 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.

[2] Symmetry would suggest that we include $(\mathsf{R}x, \mathsf{R}^{\sqsupseteq \mathsf{acq}} x)$, but this is not sound for Arm8.

Definition 4.4. A *pomset with predicate transformers* (PwT) is a tuple $(E, \lambda, \kappa, \tau, \checkmark, <)$ where

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

A PwT is *complete* if

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

Let P range over pomsets, and $\mathcal{P}$ over sets of pomsets. We give the semantics of programs $[\![\cdot]\!]$ in Fig. 1. 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). As usual, we write $d \leq e$ to mean $d < e$ or $d = e$. 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 (X1, for the termination condition) and disjunction closure (X2, for the predicate transformers themselves). (b) also requires that S6 enforce projection as well as inclusion (see the definition of *respects*). □

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

Proof. Straightforward calculation. □

In §9.4, we refine the semantics to validate the reverse inclusions for (e), (f), and (g). In §9.2, we refine the semantics to validate the reverse inclusion for (h). For Fig. 1, (i) and (j) can be strengthened to equations, rather than inclusions. However, the reverse direction does not hold for PwT-MCA (§5.1) nor for PwT-PO (§7). For further discussion, see §**??**.

Definition 4.7. P_2 is an *augment* of P_1 if all fields are equal except, perhaps, the order, where we require $<_2 \supseteq <_1$.

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 $[\![\cdot]\!]$. □

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

Within a pomset P, let $\kappa = \bigvee_{e \in E} \kappa(e)$.

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

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) $<$ respects $<_1$ and $<_2$.

If $P \in \mathit{IF}(\phi, \mathcal{P}_1, \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$
(i1) $E = (E_1 \cup E_2)$,
(i2) $\lambda = (\lambda_1 \cup \lambda_2)$,
(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) $<$ respects $<_1$ and $<_2$.

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

If $P \in \mathit{WRITE}(x, M, \mu)$ then $(\exists v \in \mathcal{V})$
(w1) $|E| \leqslant 1$,
(w2) $\lambda(e) = \mathsf{W}^\mu x v$,
(w3) $\kappa(e) \equiv M{=}v$,
(w4) $\tau^D(\psi) \equiv \psi[M/x][\kappa/\mathsf{Q}_x]$,
(w5) $\checkmark \equiv \kappa$,

If $P \in \mathit{READ}(r, x, \mu)$ then $(\exists v \in \mathcal{V})$
(r1) $|E| \leqslant 1$,
(r2) $\lambda(e) = \mathsf{R}^\mu x v$,
(r3) $\kappa(e) \equiv \mathsf{Q}_x$,
(r4a) if $e \in E \cap D$ then $\tau^D(\psi) \equiv (\kappa(e) \Rightarrow v{=}r) \Rightarrow \psi$,
(r4b) if $e \in E \setminus D$ then $\tau^D(\psi) \equiv (\kappa(e) \Rightarrow (v{=}r \vee x{=}r)) \Rightarrow \psi$,
(r4c) if $E = \emptyset$ then $\tau^D(\psi) \equiv \psi$,
(r5a) if $E \neq \emptyset$ or $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{tt}$,
(r5b) if $E = \emptyset$ and $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \mathsf{ff}$.

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

Fig. 1. PwT Semantics

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 (Wx1). From this it is easy to see that $[\![x := 1\,;\, x := 1]\!] \supseteq [\![x := 1]\!]$ is a valid refinement. It is equally obvious that

$[\![x := 1]\!] \not\supseteq [\![x := 1\,;\, x := 1]\!]$ is not a valid refinement, since the latter includes a two-element pomset, but the former does not.[3]

[Todo: Remind that ff $\equiv \bigvee_\emptyset \phi$.] In complete pomsets, C5 requires that $\checkmark$ is a tautology, capturing termination. In *WRITE*, W5 ensures that all writes are included in complete pomsets. This also ensures $[\![x := 1]\!] \not\supseteq [\![\texttt{if}(M)\{x := 1\}]\!]$, since $[\![\texttt{if}(M)\{x := 1\}]\!]$ includes the empty set with termination condition $\neg M$, but $[\![x := 1]\!]$ can only include the empty set with termination condition ff.

In addition, W5 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, however, W5 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:

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

By merging, the semantics allows the following:

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

$$(\mathsf{W}x1) \quad (M \mid \mathsf{W}x3)$$

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 $\checkmark_1 \wedge \checkmark_2$. 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 $(\phi \wedge \checkmark_1) \vee (\neg\phi \wedge \checkmark_2)$: 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 events 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 complete pomset with event $(\mathsf{W}x3)$, since $2{=}3$ is not a tautology.

We defer discussion of Q_x to §4.8. Here we assume we take $\mathsf{Q}_x = \mathsf{tt}$, for all x.

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 §9.5.)

Preconditions are introduced in:

- (I3) for control dependencies,
- (W3) for data dependencies on writes.

Predicate transformers are introduced in:

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

The rules track dependencies. We discuss data dependencies (W3) here and control dependencies (I3) in §4.6. Unless otherwise noted, we assume pomsets are *complete* and *augment-minimal.*

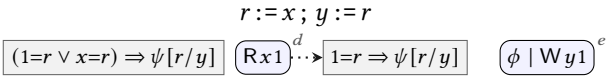
[3] These are distinguished by the context: $[-] \parallel r := x\,;\, x := 2\,;\, s := x\,;\, \texttt{if}(r{=}s)\{z := 1\}$.

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:

(††)

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

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



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 $d < e$ then we apply the dependent transformer and $\phi \equiv (1{=}r \Rightarrow r{=}1)$, a tautology.
- If $d \not< e$ then we apply the independent transformer and $\phi \equiv ((1{=}r \vee x{=}r) \Rightarrow r{=}1)$. Under the assumption that r is bound, this is logically equivalent to $(x{=}1)$. (We make this more precise in §9.2.)

Eliding transformers, the two outcomes are:

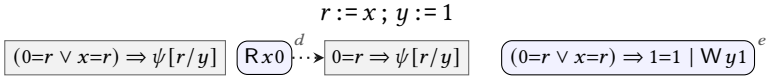
The independent case on the right can only participate in a top-level pomset if the precondition $(x{=}1)$ is discharged. To do so, we must prepend a pomset P_0 that writes 1 to x:

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. Taking $v{=}0$ and $w{=}1$:

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

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



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:

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:

where $\tau_2^{\emptyset}(\psi) \equiv \tau_2^{\{e\}}(\psi) \equiv (r{=}1 \wedge \psi[1/y]) \vee (r{\neq}1 \wedge \psi)$. As for (††), there are two possibilities:

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.

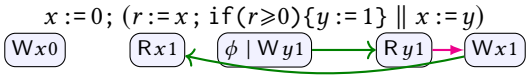
$$\kappa_2'(e) = \begin{cases} \tau_1(\kappa_2(e)) & \text{if } \lambda(e) \text{ is a read} \\ \tau_1^{\downarrow e}(\kappa_2(e)) & \text{otherwise, where } \downarrow e = \{c \mid c < e\} \end{cases}$$

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 §?? for a discussion.)

4.8 Subtleties: Local Invariant Reasoning and Local State

R4b introduces locations into formula, in order to track the local state of memory. This is necessary to support local invariant reasoning as in JMM Causality Test Case 1 (TC1) [Pugh 2004]:



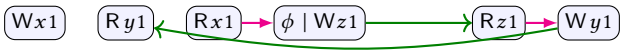
(TC1)

In order to allow this execution, the precondition ϕ must be a tautology. Using R4b and W4, 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]. Local invariant reasoning requires that we track the state of variables in the logic, not just registers. This is one reason we use predicate transformers rather than simple postconditions.

[Todo: Put tc12′ first. Fix the narrative.] Q_x ensures that the local state of x is up-to-date when x is read. R3 and R4 add these "quiescence" constraints, which are simplified by W4. Consider the following example [Paviotti et al. 2020, §6.3]:

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



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

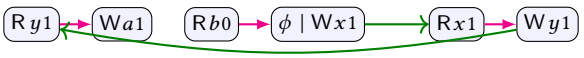
Note that the two branches of the conditional are the same except for the leading $(x := 0)$. Without Q_x, the precondition ϕ is $\mathtt{tt}$, which is a tautology, and the execution is allowed, resulting in a violation of DRF-SC. To construct this pomset, we have chosen the empty pomset for $[\![x := 0]\!]$. 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 of ϕ is $\mathtt{ff}$. Intuitively, Q_x requires that the most recent prior write to x must be in the pomset in order to read x.

We include Q_x in R3 to reduce the number of useless pomsets—when Q_x is false for $(x := r)$, the read is useless and can be eliminated by taking $E = \emptyset$. By including Q_x in R3, we also guarantee initialization in complete pomsets: (C3) requires tautologies, which means that all variables must be initialized sequentially in order to get rid of Q_x.

Control variant of TC12 with all initial values 0:

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



Building the precondition ϕ from right to left:

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

$$\phi_2 \equiv (\mathsf{Q}_b \Rightarrow 0{=}s) \Rightarrow s{=}0 \qquad \text{(Prepending } s := b\text{)}$$

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

Dependent case:

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

$$\phi_5 \equiv 1{=}r \Rightarrow (r{\neq}0 \wedge (0{=}s \Rightarrow s{=}0)) \vee (r{=}0 \wedge s{=}0) \qquad \text{(Prepending Initializers)}$$

Independent case:

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

$$\phi'_5 \equiv (1{=}r \vee 0{=}r) \Rightarrow (r{\neq}0 \wedge (0{=}s \Rightarrow s{=}0)) \vee (r{=}0 \wedge s{=}0) \qquad \text{(Prepending Initializers)}$$

4.9 Associativity and Skolemization

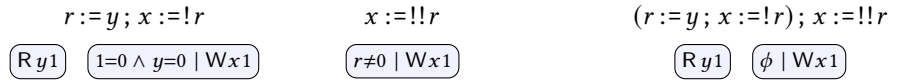
The predicate transformers we have chosen for R4a and R4b are different from the ones used traditionally, which are written using substitution. Attempting to write R4a and R4b in this style we would have (as in [Jagadeesan et al. 2020]):

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

Sadly, R4b′ fails X2, and therefore is not a predicate transformer. This is not merely a theoretical inconvenience: adopting R4b′ would also break associativity. Consider the following example, where we elide transformers for the writes and "!" represents logical negation:

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

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



The precondition $\phi \equiv (1{=}0 \wedge y{=}0) \vee (1{\neq}0 \wedge y{\neq}0)$ is equivalent to $y{\neq}0$, which is not a tautology. Our solution is to Skolemize, replacing substitution by implication, with uniquely chosen registers. Using Fig. 1, we compute $\phi \equiv ((1{=}r \vee y{=}r) \Rightarrow r{=}0) \vee ((1{=}r \vee y{=}r) \Rightarrow r{\neq}0)$, which is a tautology.

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 Arm8, but requires extra synchronization for acquiring reads.

A variant, PwT-MCA$_2$, supports optimal lowering for all access modes on Arm8. To achieve this, PwT-MCA$_2$ drops the global requirement that *reads-from* implies pomset order (M7c). The models are the same, except for *internal reads*, where a thread reads its own write.

The lowering proofs can be found in §**??**. The proofs use recent alternative characterizations of Arm8 [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 $c \leq d$ or $e \leq c$,
  (M7c) if $d \xrightarrow{\mathsf{rf}} e$ then $d < e$.

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) $\tau^D(\psi) \equiv \tau_2^D(\psi)$,
(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) $<$ respects $<_1$ and $<_2$,
(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 $P \in SEQ(\mathcal{P}_1, \mathcal{P}_2)$ 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 $P \in IF(\phi, \mathcal{P}_1, \mathcal{P}_2)$ 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) $\mathsf{rf} \supseteq (\mathsf{rf}_1 \cup \mathsf{rf}_2)$.

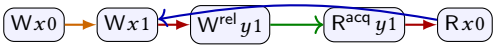
Let $[\![S_1 \Vdash S_2]\!] = PAR([\![S_1]\!], [\![S_2]\!])$. We write $[\![\cdot]\!]_{\mathsf{mca1}}$ for the semantic function when it is unclear from context.

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:[4]

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



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

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

Lemmas 4.5 and 4.6 hold for PwT-MCA_1. For further discussion of items (i) and (j), see §??.

[4] We use different colors for arrows representing order:

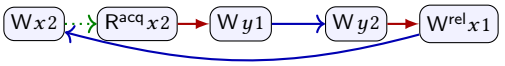
- $d \to e$ arises from $\bowtie_{\mathsf{co}}$ (S6a),
- $d \to e$ arises from $\ltimes_{\mathsf{sync}}$ or $\bowtie_{\mathsf{sc}}$ (S6a),
- $d \to e$ arises from control/data/address *dependency* (S3, definition of $\kappa_2'(d)$),
- $d \to e$ arises from *reads-from* (M7a),
- $d \to e$ arises from *blocking* (M7b).

In PwT-MCA_2, it is possible for rf to contradict $<$. In this case, we use a dotted arrow for rf: $d \dashrightarrow e$ indicates that $e < d$.

5.2 PwT-MCA2

Lowering PwT-MCA1 to Arm8 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 Arm8, but disallowed by PwT-MCA$_1$, due to the cycle.

Arm8 allows the execution because the read of x is internal to the thread. This aspect of Arm8 semantics is difficult to model locally. To capture this, we found it necessary to drop M7c and relax S6a, adding local constraints on rf to *PAR*, *SEQ* and *IF*. 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 PwT-MCA$_2$ is *complete* if it satisfies C3, C5, and C7—this is the same as for PwT-MCA$_1$.

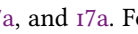
If $P \in PAR(\mathcal{P}_1, \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$
(P1) (P2) (P3) (P4) (P5) (P6) (P7) as in Def. 5.1,
(P6a) if $d \in E_1$, $e \in E_2$ and $d \xrightarrow{\mathsf{rf}} e$ then $d < e$,
(P6b) if $d \in E_1$, $e \in E_2$ and $e \xrightarrow{\mathsf{rf}} d$ then $e < d$,
(P7a) $\mathsf{rf}_i = \mathsf{rf} \cap (E_i \times E_i)$, for $i \in \{1, 2\}$.

If $P \in SEQ(\mathcal{P}_1, \mathcal{P}_2)$ 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 $P \in IF(\phi, \mathcal{P}_1, \mathcal{P}_2)$ 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 $e < d$.

[Todo: Example using S6a and S6b. To make space, Lemma 5.3 could move to appendix.]

With the weakening of S6a, 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 ensures that $[\![\texttt{if}(b)\{r := x \parallel x := 1\}\ \texttt{else}\ \{r := x;\ x := 1\}]\!]$ does not include 

, taking rf from the left and $<$ from the right.

As a consequence of dropping M7c, sequential rf must be validated 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: $d < e$ 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 Arm8 [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 §??, we show that PwT-MCA$_1$ supports the optimal lowering of relaxed accesses to Arm8 and that PwT-MCA$_2$ supports the optimal lowering of *all* accesses to Arm8. The proofs are based on two recent characterizations of Arm8 [Alglave et al. 2021]. For PwT-MCA$_1$, we use *External Global Consistency*. For PwT-MCA$_2$, we use *External Consistency*.

In §??, 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:

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

Here $\mathsf{id}(S)$ 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:

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

Notably, the semantics does *not* validate read introduction. When combined with case analysis (§9.4), read introduction can break temporal reasoning. This combination is allowed by speculative operational models. See §?? 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. This is a more precise categorisation of thin-air behavior, and it allows aggressive compiler optimizations that would be erroneously forbidden by RC11's original No-Thin-Air axiom.

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

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

(M8) $\pi : (E \to E)$ is an idempotent function capturing *merging*, such that
let $R = \{e \mid \pi(e){=}e\}$ be *real* events, let $\overline{R} = (E \setminus R)$ be *phantom* events,
let $S = \{e \mid \forall d.\ \pi(d){=}e \Rightarrow d{=}e\}$ be *simple* events, let $\overline{S} = (E \setminus S)$ be *compound* events,
(M8a) $\lambda(e) = \lambda(\pi(e))$, (M8b) if $e \in \overline{S}$ then $\kappa(e) \vdash \bigvee_{\{c \in \overline{R} \mid \pi(c) = e\}} \kappa(c)$.

(M9) $\mathsf{po} \subseteq (S \times S)$ 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 $\pi(e) \in R$.

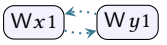
We use π to partition events E in two ways: we distinguish *real* events R from *phantom* events $\overline{R}$; we distinguish *simple* events S from *compound* events $\overline{S}$. From idempotency, it follows that all phantom events are simple ($\overline{R} \subseteq S$) and all compound events 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, $\mathsf{po} = \mathsf{po}_1 \cup \mathsf{po}_2 \cup E_1 \times E_2$. For the conditional, $\mathsf{po} = \mathsf{po}_1 \cup \mathsf{po}_2$. 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 $\leq' = \mathsf{po}'$.

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

Definition 7.3. For a PwT-po, let $\mathsf{extract}(P)$ be the projection of P onto the set $\{e \in E_1 \mid e \text{ is simple and } \kappa_1(e) \text{ is a tautology}\}$.

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

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

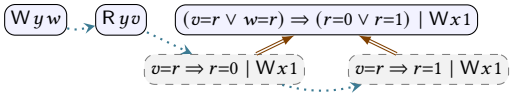
Lemma 7.4. *If P is a complete* PwT-po *with compound event e, then there is a phantom event $c \in \pi^{-1}(e)$ such that $\kappa(c)$ is a tautology.*

Proof. Immediate from M8b. □

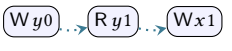
A pomset in the image of extract is a *candidate execution*.

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 $\cdots\!\!\rightarrow$, and π using a double arrow $\Longrightarrow$.

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



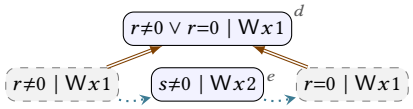
$y := 0;\ r := y;\ \texttt{if}(r{=}0)\{x := 1\};\ \texttt{if}(r{=}1)\{x := 1\}$



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

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



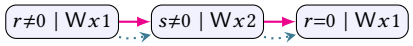
Adding a pair of reads to complete the pomset, we can extract the following candidate execution.

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

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

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

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



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:

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

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

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

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 §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 RC11 as described in §7. 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 that are not yet tautologies. We show PwTer in action in Fig. 2. PwTer will be made open source upon publication.

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

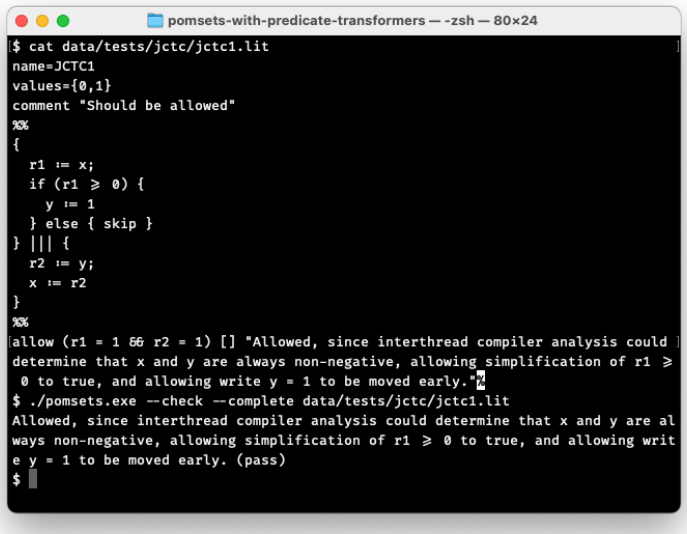


Fig. 2. Example output of PwTer, validating TC1 [Pugh 2004].

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.

Our approach to register recycling allows us to define a criterion for eliminating certain types of useless pomsets (§9.2).

In the remainder of 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 §9.6.

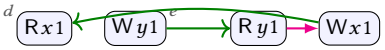
It is worth pointing out that address calculation and if-closure only affect the semantics of read and write. Fences introduce new trivial actions. RMWs require more infrastructure in order to ensure atomicity while compiling to Arm8 using load-exclusive and store-exclusive.

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 Arm8 lowering 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."

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



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

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

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

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

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

which is a tautology. In (**), the first read resolves the nondeterminism in both the first and the second read. Given the choice of event names, the outcome of the second read is predetermined! In (*), the second read remains nondeterministic, even 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 $e \in E \cap D$ then $\tau^D(\psi) \equiv v{=}s_e \Rightarrow \psi[s_e/r]$,
(R4b) if $e \in E \setminus D$ then $\tau^D(\psi) \equiv (v{=}s_e \vee x{=}s_e) \Rightarrow \psi[s_e/r]$,
(R4c) if $E = \emptyset$ then $\tau^D(\psi) \equiv (\forall s)\ \psi[s/r]$.

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

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

As a further example, consider [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:

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

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 Register Consistency

[Todo: Drop M3a. Move this into a discussion of the proof of compilation correctness.]

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

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

Definition 9.3. A *λ-predicate transformer* is a function $\tau : \Phi \to \Phi$ such that

(x3) (x1) (x2) 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 and termination conditions.

(M3a) $\kappa(e)$ is λ-consistent, (M5b) $\checkmark$ is λ-consistent.

With this modification, dead-code elimination (Lemma 4.6h) can be changed from an inclusion to an equation:

$$\texttt{if}(\phi)\{\mathcal{P}_1\}\texttt{else}\{\mathcal{P}_2\} = \mathcal{P}_1 \text{ if } \phi \text{ is a tautology.}$$

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) $\kappa(e) \equiv \mathsf{tt}$, (F5) $\checkmark \equiv \kappa$.

(F2) $\lambda(e) = \mathsf{F}^\mu$, (F4) $\tau^D(\psi) \equiv \psi$,

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 9.4. Extend the definition of a pomset as follows.

(M10) $\mathsf{rmw} : E \to E$ is a partial function capturing read-modify-write *atomicity*, such that

(M10a) if $d \xrightarrow{\mathsf{rmw}} e$ then $\lambda(e)$ blocks $\lambda(d)$,

(M10b) if $d \xrightarrow{\mathsf{rmw}} e$ then $d < e$,

(M10c) if $\lambda(c)$ overlaps $\lambda(d)$ and $d \xrightarrow{\mathsf{rmw}} e$ then $c < e$ implies $c \leq d$ and $d < c$ implies $e \leq c$.

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

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

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

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

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

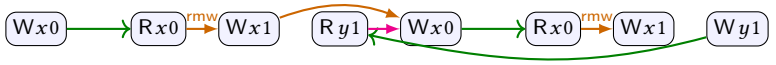
(U10) if $\lambda(e)$ is a write then there is a read $\lambda(d)$ such that $\kappa(e) \vdash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.

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

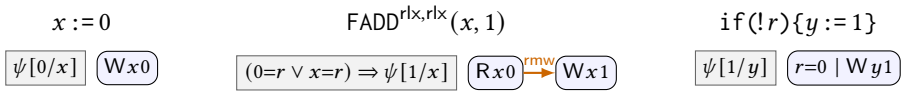
This definition ensures atomicity and supports lowering to Arm load/store exclusive operations. See [Jagadeesan et al. 2020] for examples.

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′*.

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

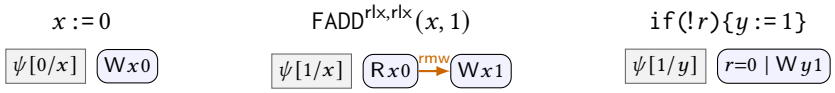


A write should only be visible to one FADD instruction, but here the write of 0 is visible to two. This is allowed because no order is required from $(\mathsf{R}x0)$ to $(\mathsf{W}y1)$ in the last thread. To see why, consider the independent transformers of the last thread and initializer:



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

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

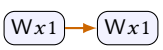


After sequencing, the precondition of $(\mathsf{W}y1)$ is $r{=}0$, which is *not* a tautology. This forces any top-level pomset to include dependency order from $(\mathsf{R}x0)$ to $(\mathsf{W}y1)$.

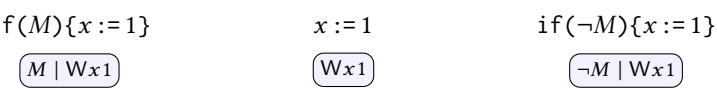
9.4 If-Closure

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:

$$\mathtt{if}(M)\{x := 1\};\ S;\ \mathtt{if}(\neg M)\{x := 1\}$$

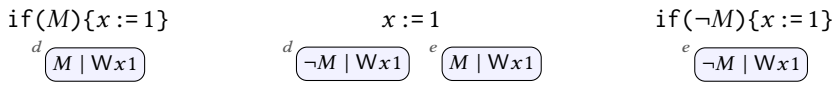


However, if $S = (\mathtt{if}(\neg M)\{x := 1\};\ \mathtt{if}(M)\{x := 1\})$, then it *does* allow the execution. Looking at the initial program:



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.

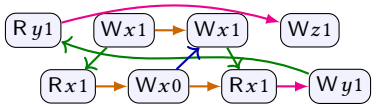
Def. 9.6 allows the execution, by splitting the middle command:



Coalescing events gives the desired result.

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

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



Definition 9.6. If $P \in \mathit{WRITE}(x, M, \mu)$ then $(\exists v : E \to \mathcal{V})$

(w2) $\lambda(e) = \mathsf{W}^{\mu} x v_e$, (w4) $\tau^D(\psi) \equiv \psi[M/x][\kappa/\mathsf{Q}_x]$,
(w3) $\kappa(e) \vDash M{=}v_e$, (w5) $\checkmark \equiv \kappa$,

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

(R2) $\lambda(e) = \mathsf{R}^{\mu} x v_e$ (R5a) if $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{tt}$,
(R3) $\kappa(e) \vDash \mathsf{Q}_x$, (R5b) if $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \kappa$.
(R4) $\tau^D(\psi) \equiv \bigwedge_{e \in E \cap D}(\kappa(e) \Rightarrow v_e{=}s_e) \Rightarrow \psi[s_e/r]$
$\wedge \bigwedge_{e \in E \setminus D}(\kappa(e) \Rightarrow (v_e{=}s_e \vee x{=}s_e)) \Rightarrow \psi[s_e/r]$
$\wedge \neg\kappa \Rightarrow (\forall s)\ \psi[s/r]$

The definition allows multiple events to represent a single action, each with a disjoint precondition. The predicate transformers are derived from those defined for the conditional.

This modification validates Lemma 4.6e, f, and g as equations.

We show how to combine address calculation and if-closure in §9.6.

9.5 Address Calculation

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

Definition 9.7. Within a pomset P, let $\kappa|_x = \bigvee \{\kappa(e) \mid e \in E \wedge \lambda(e) = \mathsf{W}x\}$.
If $P \in \mathit{WRITE}(L, M, \mu)$ then $(\exists \ell \in \mathcal{V})\ (\exists v \in \mathcal{V})$

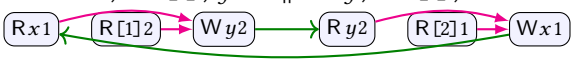
(w1) if $|E| \leqslant 1$, (w4) $\tau^D(\psi) \equiv \bigwedge_{k \in \mathcal{V}} L{=}k \ \Rightarrow \psi[M/[k]][\kappa|_{[k]}/\mathsf{Q}_{[k]}]$,
(w2) $\lambda(e) = \mathsf{W}^{\mu}[\ell]v$, (w5) $\checkmark \equiv \bigvee_{e \in E}(L{=}\ell \wedge M{=}v)$.
(w3) $\kappa(e) \equiv L{=}\ell \wedge M{=}v$,

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

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

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:

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

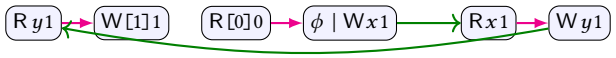


(ADDR1)

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

[Todo: Check above example. Narrate TC12.] TC12 with all initial values 0:

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



(TC12)

Building the precondition ϕ from right to left:

$$\phi_1 \equiv s{=}0 \qquad (x := s)$$
$$\phi_2 \equiv (\mathsf{Q}_{[0]} \Rightarrow 0{=}s) \Rightarrow s{=}0 \qquad (\text{Prepending } s := [0])$$
$$\phi_3 \equiv (r{=}1 \Rightarrow \phi_2[1/[1]][\mathtt{tt}/\mathsf{Q}_{[1]}]) \wedge (r{=}0 \Rightarrow \phi_2[1/[0]][\mathtt{ff}/\mathsf{Q}_{[0]}]) \qquad (\text{Prepending } \mathtt{if})$$
$$\equiv (r{=}1 \Rightarrow (\mathsf{Q}_{[0]} \Rightarrow 0{=}s) \Rightarrow s{=}0) \wedge (r{=}0 \Rightarrow s{=}0)$$

Dependent case:

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

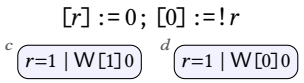
Independent case:

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

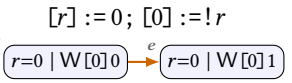
9.6 Combining Address Calculation and If-Closure

Def. 9.7 is naive with respect to merging events. Consider the following example:

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

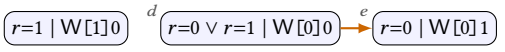


$$[r] := 0\,;\ [0] := !r$$



Merging, we have:

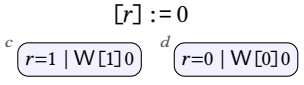
$$\mathtt{if}(M)\{[r] := 0\,;\ [0] := !r\}\,\mathtt{else}\,\{[r] := 0\,;\ [0] := !r\}$$



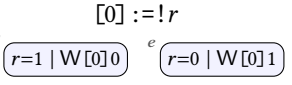
The precondition of $\mathsf{W}[0]0$ is a tautology; however, this is not possible for $([r] := 0\,;\ [0] := !r)$ alone, using Def. 9.7.

Def. 9.8, enables this execution using if-closure. Under this semantics, we have:

$$[r] := 0$$

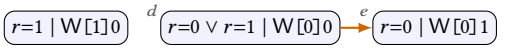


$$[0] := !r$$



Sequencing and merging:

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



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

Definition 9.8. If $P \in \mathit{WRITE}(L, M, \mu)$ then $(\exists \ell : E \to \mathcal{V})$ $(\exists v : E \to \mathcal{V})$

(w2) $\lambda(e) = \mathsf{W}^{\mu}[\ell]v_e$,
(w3) $\kappa(e) \vDash L{=}\ell_e \wedge M{=}v_e$,
(w4) $\tau^D(\psi) \equiv \bigwedge_{k \in \mathcal{V}} L{=}k \ \Rightarrow \psi[M/k][\kappa|_{[k]}/\mathsf{Q}_{[k]}]$,
(w5) $\checkmark \equiv \kappa$.

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

(R2) $\lambda(e) = \mathsf{R}^{\mu}\,[\ell]\,v_e$

(R3) $\kappa(e) \equiv \phi_e \wedge L{=}\ell_e \wedge \mathsf{Q}_{[\ell]}$,

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

(R5a) if $\mu \sqsubseteq \mathsf{rl}\times$ then $\checkmark \equiv \mathsf{tt}$,

(R5b) if $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \kappa$.

REFERENCES

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

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

Mark Batty, 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 *Formal Methods. FM 2019 International Workshops*. Springer International Publishing, 507–525.

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

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

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

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

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

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