The Leaky Semicolon

Compositional Semantic Dependencies for Relaxed-Memory Concurrency

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

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

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

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

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

ACM Reference Format:
Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The Leaky Semicolon: Compositional Semantic Dependencies for Relaxed-Memory Concurrency. *Proc. ACM Program. Lang.* 0, POPL, Article 0 (January 2022), 31 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].

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

2475-1421/2022/1-ART0
https://doi.org/

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2 OVERVIEW

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

2.1 Sequential Composition

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

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

Without loops, the Hoare triple $\{\phi\}\ S\ \{\psi\}$ holds exactly when $\phi \Rightarrow wp_S(\psi)$. This is an elegant explanation of sequential computation in a sequential context. Note that the assignment rule is sound because a read from a thread-local register must be fulfilled by a preceding write in the same thread. In a concurrent context, with shared variables (x–z), the obvious generalization of the assignment rule for reads, $wp_{r:=x}(\psi) = \psi[x/r]$, is unsound! In particular, a read from a shared memory location may be fulfilled by a write in another thread.

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

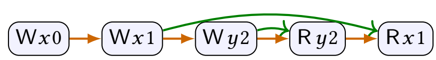
(1) it should not impose too much order, overconstraining the implementation,
(2) it should not impose too little order, allowing bogus executions, and
(3) it should be *compositional* and *direct*, as described in §1.

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

2.2 Memory Models

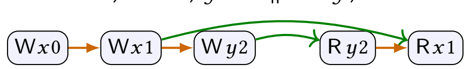
For single-threaded programs, memory can be thought of as you might expect: programs write to, and read from, memory references. This can be thought of as a total order over memory actions (→), where each read has a matching *fulfilling* write (→), for example:

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



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

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

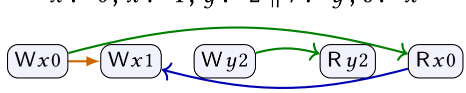


We can represent such an execution as a labelled partial order, or *pomset* [Gischer 1988; Pratt 1985]. A program may give rise to many executions, each reflecting a different interleaving of the threads.

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

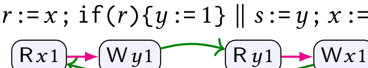
This has led to *relaxed memory models*, in which the requirement of sequential consistency is weakened to only apply *per-location*. This allows executions that are inconsistent with program order, such as the following, which contains an *antidependency* (→):

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



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 the following. (We routinely elide initializing writes when they are uninteresting.)

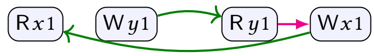
$$r := x\,;\; \mathtt{if}(r)\{y := 1\} \parallel s := y\,;\; x := s$$



This candidate execution forms a cycle in causal order, so is disallowed, but this depends crucially on the control dependency from (Rx1) to (Wy1), and the data dependency from (Ry1) to (Wx1).

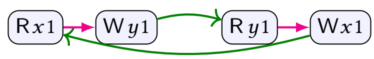
If either is missing, then this execution is acyclic and hence allowed. For example dropping the control dependency results in:

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



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

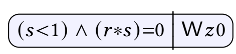
$$r := x\,;\ \texttt{if}(r)\{y := 1\}\,\texttt{else}\,\{y := 1\} \parallel s := y\,;\ x := s$$



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

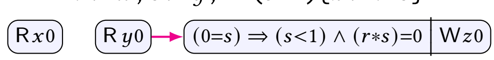
To address this, Jagadeesan et al. [2020] introduced *Pomsets with Preconditions* (PwP), where events are labeled with logical formulae. Nontrivial preconditions are introduced by store actions (modeling data dependencies) and conditionals (modeling control dependencies):

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

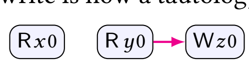


In this diagram, $(s{<}1)$ is a control dependency and $(r{*}s){=}0$ is a data dependency. Preconditions are updated as events are prepended (we assume the usual precedence for logical operators):

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



In this diagram there are two reads. As evidenced by the arrow, the read of y is ordered before the write, reflecting possible dependency; the read of x is not, reflecting independency. The dependent read of y allows the precondition of the write to weaken: now the old precondition need only be satisfied assuming the hypothesis $(0{=}s)$. The independent read of x allows no such weakening. Nonetheless, the precondition of the write is now a tautology, and so can be elided in the diagram:



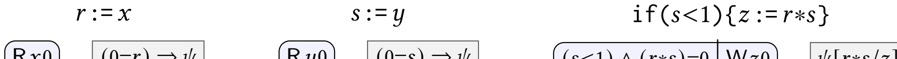
In order for a pomset to be *complete*, all preconditions must be tautologies. Thus, an execution of (†) that reads Ry2 must elide the write action to be considered complete. For this reason, we must allow read and write to generate empty pomsets, in addition to singletons.

2.3 Predicate Transformers For Relaxed Memory

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

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

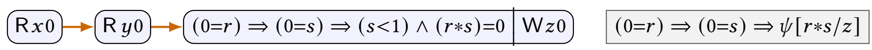
$r := x$ $\qquad$ $s := y$ $\qquad$ $\texttt{if}(s{<}1)\{z := r{*}s\}$



The predicate transformer for a write $z := M$ matches Dijkstra: taking ψ to $\psi[M/z]$. For a read $r := x$, however, Dijkstra would transform ψ to $\psi[x/r]$, which is equivalent to $(x{=}r) \Rightarrow \psi$ under the assumption that registers are assigned at most once. Instead, we use $(0{=}r) \Rightarrow \psi$, 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 subsequent preconditions, composing the predicate transformers.

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



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

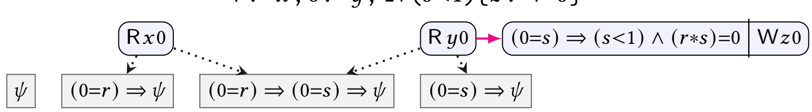
The key observation of this paper is that rather than working with one predicate transformer, we should work with a *family* of predicate transformers, indexed by sets of events. For example, for single-event pomsets, there are two predicate transformers, since there are two subsets of any one-element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

$$r := x \qquad\qquad s := y$$

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.

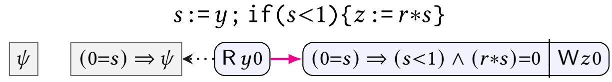
For example, we can recover the expected semantics for (†) by choosing the predicate transformer which is independent of (R x0) but dependent on (R y0), which is the transformer which maps ψ to $(0{=}s) \Rightarrow \psi$. (In subsequent diagrams, we only show predicate transformers for reads.)

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



In the diagram, the dotted lines indicate set inclusion into the index of the transformer-family. As a quick correctness test, we can see that sequential composition is associative in this case, since it does not matter whether we associate to the left, with the intermediate step eliding (W z0) in the diagram above, or to the right, with the intermediate step:

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



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

3 SEQUENTIAL SEMANTICS

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

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

3.1 Preliminaries

The syntax is built from

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

- a set of *expressions* $\mathcal{M}$, ranged over by M, N, L.

Memory references 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.

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

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

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], $\mathbin{+\!\!\!\!+\!\!\!\!\!\rightarrow}$ denotes parallel composition, preserving thread state on the right after a join. In examples and sublanguages without join, we use the symmetric $\parallel$ operator.

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

Throughout §1–7 we require that

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

In §8, we drop this restriction, requiring instead that

- there are registers that do not appear in programs.

The semantics is built from the following.

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

We require that

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

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

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

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

3.2 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 3.1. Actions (R) are *read* actions.
We say a *matches* b if $a = (\mathsf{W} x v)$ and $b = (\mathsf{R} x v)$.
We say a *blocks* b if $a = (\mathsf{W} x)$ and $b = (\mathsf{R} x)$, regardless of value.
We say a *overlaps* b if they access the same location, regardless of whether they read or write.
Let $\bowtie_{\mathsf{co}}$ capture write-write, read-write coherence: $\bowtie_{\mathsf{co}} = \{(\mathsf{W}x, \mathsf{W}x),\ (\mathsf{R}x, \mathsf{W}x),\ (\mathsf{W}x, \mathsf{R}x)\}$.
Let $\ltimes_{\mathsf{sync}}$ capture conflict due to synchronization:[2] $\ltimes_{\mathsf{sync}} = \{(a, \mathsf{W}^{\sqsupseteq\mathsf{rel}}),\ (a, \mathsf{F}^{\sqsupseteq\mathsf{rel}}),\ (\mathsf{R}, \mathsf{F}^{\sqsupseteq\mathsf{acq}}),\ (\mathsf{R}^{\sqsupseteq\mathsf{acq}}, b),\ (\mathsf{F}^{\sqsupseteq\mathsf{acq}}, b),\ (\mathsf{F}^{\sqsupseteq\mathsf{rel}}, \mathsf{W}),\ (\mathsf{W}^{\sqsupseteq\mathsf{rel}}x, \mathsf{W}x)\}$.
Let $\bowtie_{\mathsf{sc}}$ capture conflict due to sc access: $\bowtie_{\mathsf{sc}} = \{(\mathsf{W}^{\mathsf{sc}}, \mathsf{W}^{\mathsf{sc}}),\ (\mathsf{R}^{\mathsf{sc}}, \mathsf{W}^{\mathsf{sc}}),\ (\mathsf{W}^{\mathsf{sc}}, \mathsf{R}^{\mathsf{sc}}),\ (\mathsf{R}^{\mathsf{sc}}, \mathsf{R}^{\mathsf{sc}})\}$.
We say a *delays* b if $a \bowtie_{\mathsf{co}} b$ or $a \ltimes_{\mathsf{sync}} b$ or $a \bowtie_{\mathsf{sc}} b$.

3.3 PwT: Pomsets with Predicate Transformers

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

Definition 3.2. A *predicate transformer* is a function $\tau : \Phi \to \Phi$ 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 τ^C as the predicate transformer applied to event e where $d \in (C)$ if $d < e$.

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

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

In sequential composition, adding more order can only increase the size of C. Following Def. 3.3, the larger C is, the better, at least in terms of satisfying preconditions. Thus more order means weaker preconditions.

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

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

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

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

A PwT is *complete* if
(C3) $\kappa(e)$ is a tautology (for every $e \in E$), (C5) $\checkmark$ is a tautology.

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.

[Todo: I struggled with the concept of a "termination condition". Can you add a sentence or two of intuition here?]

[Todo: Fig 1. Please explain why (R4b) is defined in this way. In addition, why can the set E of events be empty for read and write operations?]

Before discussing the details of the model, we note that the semantics satisfies the expected monoid laws, as well as some laws concerning the conditional. We have verified Lemma 3.5 and Lemma 3.6e in Coq[4]. Similar laws apply to parallel composition; although note that $[\![S \mathbin{+\!\!\!\!\!\!+\!\!\!\!\!\!\!\;\rightarrow} \texttt{skip}]\!] \neq [\![S]\!]$, since this asymmetric operator throws away thread state from the left.

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

The proof of (a) requires M5a for the termination condition in $(S\,;\,\texttt{skip})$. (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 3.6. *(c)* $[\![\texttt{if}(\phi)\{S_1\}\,\texttt{else}\,\{S_2\}]\!] \supseteq [\![S_1]\!]$ *if* ϕ *is a tautology.*
(d) $[\![\texttt{if}(\phi)\{S\}\,\texttt{else}\,\{S\}]\!] \supseteq [\![S]\!]$.
(e) $[\![\texttt{if}(\phi)\{S_1\,;\,S_3\}\,\texttt{else}\,\{S_2\,;\,S_3\}]\!] \supseteq [\![\texttt{if}(\phi)\{S_1\}\,\texttt{else}\,\{S_2\}\,;\,S_3]\!]$.
(f) $[\![\texttt{if}(\phi)\{S_1\,;\,S_2\}\,\texttt{else}\,\{S_1\,;\,S_3\}]\!] \supseteq [\![S_1\,;\,\texttt{if}(\phi)\{S_2\}\,\texttt{else}\,\{S_3\}]\!]$.
(g) $[\![\texttt{if}(\neg\phi)\{S_2\}\,;\,\texttt{if}(\phi)\{S_1\}]\!] \subseteq [\![\texttt{if}(\phi)\{S_1\}\,\texttt{else}\,\{S_2\}]\!] \supseteq [\![\texttt{if}(\phi)\{S_1\}\,;\,\texttt{if}(\neg\phi)\{S_2\}]\!]$.

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

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

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

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

[Todo: A compiler does not have to use the proposed semantics directly, i.e., for calculating dependencies. Instead, the semantics is meant to be a wrapper that validates some reasonable set of compiler optimizations. A compiler may make more conservative assumptions about dependencies than the semantics. This is explicitly allowed by Lemma 4.8 (augment closure).]

3.4 Pomsets and Complete Pomsets

[Todo: We include empty pomsets in the semantics of reads/writes/fences to allow them to appear on the false side of a conditional, for example, since complete pomsets require that all preconditions are tautologies.]

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

In complete pomsets, c5 requires that ✓ is a tautology, capturing termination. In *WRITE*, w5 ensures that all writes are included in complete pomsets—note that $\mathsf{K}(\emptyset) = \mathsf{ff}$. 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. [Todo: 494: This seems wrong: you're saying that [x:=1] is not a non-strict superset of [if(M)x:=1]. But they're surely equal if M is true?]

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 ✓ implies $1{=}v$. We might wish to require that all preconditions be satisfiable. However, unsatisfiable writes can 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:

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

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

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

In *READ*, ✓ depends on the mode. r5b ensures that all acquiring reads are included in complete pomsets. Instead r5a states that relaxed reads are optional: ✓ 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 ✓ to be $✓_1 \wedge ✓_2$. 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 $(\phi \wedge ✓_1) \vee (\neg\phi \wedge ✓_2)$: the program terminates as long as the taken 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\llbracket r := M \rrbracket = \mathit{ASSIGN}(r, M) \qquad \llbracket \mathsf{F}^\mu \rrbracket = \mathit{FENCE}(\mu) \qquad \llbracket S_1 \mathbin{+\!\!\!\!\!\!\;\!+} S_2 \rrbracket = \mathit{PAR}(\llbracket S_1 \rrbracket, \llbracket S_2 \rrbracket)$$

$$\llbracket x^\mu := M \rrbracket = \mathit{WRITE}(x, M, \mu) \qquad \llbracket \mathtt{skip} \rrbracket = \mathit{SKIP} \qquad \llbracket S_1 \,;\, S_2 \rrbracket = \mathit{SEQ}(\llbracket S_1 \rrbracket, \llbracket S_2 \rrbracket)$$

$$\llbracket r := x^\mu \rrbracket = \mathit{READ}(r, x, \mu) \qquad \llbracket \mathtt{if}(M)\{S_1\}\mathtt{else}\{S_2\} \rrbracket = \mathit{IF}(M{\neq}0, \llbracket S_1 \rrbracket, \llbracket S_2 \rrbracket)$$

Fig. 1. PwT Semantics

left and the empty set from the right. This is a general principle: for code that contributes no events at top-level, use the empty set.

3.5 Preconditions, Predicate Transformers, and Data Dependencies

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 §3.8. Here we assume we take $\mathsf{Q}_x = \mathrm{tt}$, for all x.

Note that $[\![S_1 \mathbin{+\!\!\!\!\!\parallel} S_2]\!]$ is asymmetric, taking the predicate transformer for S_2 in P4.

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 rule is S3, which uses the transformed predicate $\kappa_2'(e) = \tau_1^C(\kappa_2(e))$, where $C = \{c \mid c < e\}$ is the set of events that precede e in causal order. We call C the *dependent set* for e. Then $E \setminus (C)$ is the *independent set*.

Before looking at the details, it is useful to have a high-level view of how nontrivial preconditions and predicate transformers are introduced. (We discuss address dependencies in §8.4.)

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 §3.6. Unless otherwise noted, we assume pomsets are *complete* and *augment-minimal*.

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:

$$r := x \qquad\qquad y := r$$

$$\boxed{(v{=}r \lor x{=}r) \Rightarrow \psi}\ (\mathsf{R}xv)^{d} \dashrightarrow \boxed{v{=}r \Rightarrow \psi} \qquad \boxed{\psi[r/y]}\ (r{=}w \mid \mathsf{W}yw)^{e} \dashrightarrow \boxed{\psi[r/y]} \tag{††}$$

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

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

For the read, the dependent transformer $\tau_1^{\{d\}}$ is $1{=}r \Rightarrow \psi$; the independent transformer $\tau_1^{\emptyset}$ is $(1{=}r \lor x{=}r) \Rightarrow \psi$. 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:

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

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

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.

Note that both R4a and R4b degenerate to R4c when $\kappa(e) = \mathrm{ff}$.

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 $\phi \equiv ((1{=}r \lor x{=}r) \Rightarrow r{=}1)$. Under the assumption that r is bound (see footnote 3), this is logically equivalent to $(x{=}1)$.

Eliding transformers, the two outcomes are:

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

$$(\mathsf{R}x1)^{d} \rightarrow (\mathsf{W}y1)^{e} \qquad\qquad (\mathsf{R}x1)^{d} \quad (x{=}1 \mid \mathsf{W}y1)^{e}$$

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:

$$x := 1 \qquad\qquad x := 1\,;\, r := x\,;\, y := r$$

$$\boxed{\psi[1/x]}\ \ (1{=}1 \mid \mathsf{W}x1)^c \dashrightarrow \boxed{\psi[1/x]} \qquad\qquad (1{=}1 \mid \mathsf{W}x1)^c\ \ (\mathsf{R}x1)^d\ \ (1{=}1 \mid \mathsf{W}y1)^e$$

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 $w{=}1$:

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

$$(\mathsf{R}x0)^d \rightarrow (0{=}r \Rightarrow r{=}1 \mid \mathsf{W}y1)^e \qquad\qquad (\mathsf{R}x0)^d\ \ ((0{=}r \lor x{=}r) \Rightarrow r{=}1 \mid \mathsf{W}y1)^e$$

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:

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

$$\boxed{(0{=}r \lor x{=}r) \Rightarrow \psi[r/y]}\ \ (\mathsf{R}x0)^d \dashrightarrow \boxed{0{=}r \Rightarrow \psi[r/y]} \qquad ((0{=}r \lor x{=}r) \Rightarrow 1{=}1 \mid \mathsf{W}y1)^e$$

3.6 Control Dependencies

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

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

For events from E_1, I3 requires $\phi \land \kappa_1(e)$. For events from E_2, I3 requires $\neg\phi \land \kappa_2(e)$. For coalescing events in $E_1 \cap E_2$, I3 requires $(\phi \land \kappa_1(e)) \lor (\neg\phi \land \kappa_2(e))$. This semantics allows common code to be lifted out of a conditional, validating the transformation $[\![\mathtt{if}(M)\{S\}\mathtt{else}\{S\}]\!] \supseteq [\![S]\!]$.

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

$$y := r \qquad\qquad y := 1 \qquad\qquad \mathtt{if}(r{=}1)\{y := r\}\mathtt{else}\{y := 1\}$$

$$(r{=}1 \mid \mathsf{W}y1)^e \qquad\qquad (1{=}1 \mid \mathsf{W}y1)^e \qquad\qquad ((r{=}1 \Rightarrow r{=}1) \land (r{\neq}1 \Rightarrow 1{=}1) \mid \mathsf{W}y1)^e$$

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

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

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

$$\boxed{(v{=}r \lor x{=}r) \Rightarrow \psi}\ \ (\mathsf{R}xv)^d \dashrightarrow \boxed{v{=}r \Rightarrow \psi} \qquad\qquad \boxed{\tau_2^{\emptyset}(\psi)}\ \ (r{=}1 \mid \mathsf{W}yw)^e \dashrightarrow \boxed{\tau_2^{\{e\}}(\psi)}$$

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

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

$$(\mathsf{R}x1)^d \rightarrow (1{=}r \Rightarrow r{=}1 \mid \mathsf{W}y1)^e \qquad\qquad (\mathsf{R}x1)^d\ \ ((1{=}r \lor x{=}r) \Rightarrow r{=}1 \mid \mathsf{W}y1)^e$$

[Todo: Add example showing empty set on untaken branch.]

3.7 A Refinement: No Dependencies into Reads

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

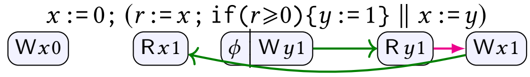
$$\kappa_2'(e) = \begin{cases} \tau_1^{E_1}(\kappa_2(e)) & \text{if } \lambda(e) \text{ is a read} \\ \tau_1^{C}(\kappa_2(e)) & \text{otherwise, where } C = \{c \mid c < e\} \end{cases}$$

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

3.8 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]:

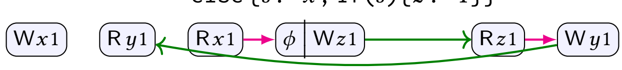
$$x := 0;\ (r := x;\ \mathtt{if}(r \geqslant 0)\{y := 1\} \parallel x := y) \qquad (\text{TC1})$$



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. 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]:

$$\begin{array}{l} x := 1;\ r := y;\ \mathtt{if}(r{=}0)\{x := 0;\ s := x;\ \mathtt{if}(s)\{z := 1\}\} \ \parallel\ \mathtt{if}(z)\{y := 1\} \\ \qquad\qquad\qquad\quad \mathtt{else}\{s := x;\ \mathtt{if}(s)\{z := 1\}\} \end{array}$$



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

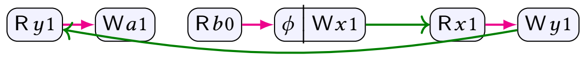
[Todo: Make this understandable.] Note that the two branches of the conditional are the same except for the leading $(x := 0)$. Without Q_x, the precondition ϕ is tt, which is a tautology, and the execution is allowed, resulting in a violation of DRF-SC. To construct this pomset, we have chosen the empty pomset for $[\![x := 0]\!]$. The constraints on complete pomsets do not filter out this pomset, since $x := 0$ is in the untaken branch of the conditional. The problem here is that we have forgotten the local state of x in the untaken branch of the execution. Nonetheless, we are using the subsequent read.

With Q_x, the precondition of ϕ is 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:

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



Building the precondition ϕ from right to left:

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

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

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

Dependent case:

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

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

Independent case:

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

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

3.9 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:

$$r := y \qquad\qquad x := !r \qquad\qquad x := !!r$$

$$\boxed{\psi[1/r] \wedge \psi[y/r]}\ \boxed{\mathsf{R}\,y1} \dashrightarrow \boxed{\psi[1/r]} \qquad \boxed{r{=}0 \mid \mathsf{W}\,x1} \qquad \boxed{r{\neq}0 \mid \mathsf{W}\,x1}$$

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

$$r := y \qquad\qquad x := !r\,;\ x := !!r \qquad\qquad r := y\,;\ (x := !r\,;\ x := !!r)$$

$$\boxed{\mathsf{R}\,y1} \qquad\qquad \boxed{\mathsf{W}\,x1} \qquad\qquad \boxed{\mathsf{R}\,y1}\ \boxed{\mathsf{W}\,x1}$$

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

$$r := y\,;\ x := !r \qquad\qquad x := !!r \qquad\qquad (r := y\,;\ x := !r)\,;\ x := !!r$$

$$\boxed{\mathsf{R}\,y1}\ \boxed{1{=}0 \wedge y{=}0 \mid \mathsf{W}\,x1} \qquad \boxed{r{\neq}0 \mid \mathsf{W}\,x1} \qquad \boxed{\mathsf{R}\,y1}\ \boxed{\phi \mid \mathsf{W}\,x1}$$

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.

4 PwT-MCA: POMSETS WITH PREDICATE TRANSFORMERS FOR MCA

In concurrent hardware, each processor has two views of each memory location: local and global. In a *multicopy-atomic* (MCA) architecture, there is only one global view, shared by all processors. These two views are neatly reflected in PwT: the local view corresponds to the preconditions. The global view corresponds to the partial order.

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

A variant, PwT-MCA$_2$, supports optimal lowering for all access modes on Arm8. To achieve this, PwT-MCA$_2$ drops the global requirement that *reads-from* implies pomset order (M7c). The models are the same, except for *internal reads*, where a thread reads its own write. We show an example at the beginning of §4.2.

The lowering proofs can be found in [Jeffrey et al. 2022]. The proofs use recent alternative characterizations of Arm8 [Alglave et al. 2021].

4.1 PwT-MCA1

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

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

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

The definition of completeness extends Def. 3.4 as follows:

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

The semantic function extends Fig. 1 as follows:

(P7) (S7) (I7) rf respects rf_1 and rf_2,

(I6a) if $\lambda_1(d)$ delays $\lambda_2(e)$ then $d \leq e$.

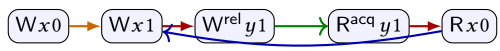
We write $[\![\cdot]\!]_{\mathsf{mca1}}$ for the semantic function when it is unclear from context.

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 3.5) extend to parallel composition, with `skip` as right unit only due to the asymmetry of P4.

Only I6a requires explanation. From Def. 3.1, recall that a *delays* b if $a \bowtie_{\mathsf{co}} b$ or $a \ltimes_{\mathsf{sync}} b$ or $a \bowtie_{\mathsf{sc}} b$. I6a 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:[5]

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



The execution is disallowed due to the cycle. All of the order shown is required at top-level: The intra-thread order comes from I6a: $(\mathsf{W}x0) \to (\mathsf{W}x1)$ 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* $(\mathsf{R}x0) \longrightarrow (\mathsf{W}x1)$ is required by M7b. (Alternatively, we could have $(\mathsf{W}x1) \longrightarrow (\mathsf{W}x0)$, again resulting in a cycle.)

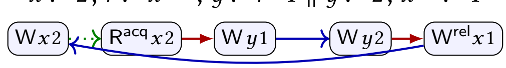
The semantics gives the expected results for store buffering and load buffering, as well as litmus tests involving fences and SC access. The model of coherence is weaker than C11, in order to support common subexpression elimination, and stronger than Java, in order to support local reasoning about data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

Lemmas 3.5 and 3.6 hold for PwT-MCA$_1$. For further discussion of item (g) see [Jeffrey et al. 2022].

4.2 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 I6a, 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 4.2. The definition of PwT-MCA$_2$ is derived from that of PwT-MCA$_1$ by removing M7c and adding the following:

(P6a) if $d \in E_1$, $e \in E_2$ and $d \xrightarrow{\mathsf{rf}} e$ then $d < e$,
(P6b) if $d \in E_1$, $e \in E_2$ and $e \xrightarrow{\mathsf{rf}} d$ then $e < d$,
(S6a) if $d \in E_1$, $e \in E_2$ and $\lambda_1(d)$ delays $\lambda_2(e)$ then either $d \xrightarrow{\mathsf{rf}} e$ or $d \le e$,
(S6b) if $d \in E_1$, $e \in E_2$ and $\lambda_1(d)$ blocks $\lambda_2(e)$ then $c \xrightarrow{\mathsf{rf}} e$ implies $d \le c$.

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. Perhaps move Lemma 4.3 to appendix.]

[5] We use different colors for arrows representing order:
- $d \to e$ arises from $\bowtie_{\mathsf{co}}$ (I6a),
- $d \to e$ arises from $\ltimes_{\mathsf{sync}}$ or $\bowtie_{\mathsf{sc}}$ (I6a),
- $d \to e$ arises from control/data/address *dependency* (S3, definition of $\kappa_2'(d)$),
- $d \longrightarrow e$ arises from *reads-from* (M7a),
- $d \longrightarrow e$ arises from *blocking* (M7b).

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

With the weakening of I6a, we must be careful not to allow spurious pairs to be added to the rf relation. For example, $[\![\texttt{if}(b)\{r := x \parallel x := 1\}\texttt{else}\{r := x\,;\, x := 1\}]\!]$ should not include $(\mathsf{R}x1) \xrightarrow{\mathsf{rf}} (\mathsf{W}x1)$, taking rf from the left and $<$ from the right. The use of respects ensures this.

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

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

- *external fulfillment: $d < e$ and if $\lambda(c)$ blocks $\lambda(e)$ then either $c \leq d$ or $e \leq c$, 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].

5 PwT-MCA RESULTS

PwP 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. The proofs can be found in [Jeffrey et al. 2022].

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*.

We prove sequential consistency for local-data-race-free programs. The proof uses *program order*, which we construct for C11 in §6. The same construction works for PwT-MCA. (This proof assumes there are no RMW operations.)

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

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

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

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

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

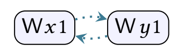
Prop. 6.1 of [Jagadeesan et al. 2020] establishes a compositional principle for proving that programs validate formula in past-time temporal logic. The principal is based entirely on the pomset order relation. Its proof, and all of the no-thin-air examples in [Jagadeesan et al. 2020, §6] hold equally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

6 PwT-C11: POMSETS WITH PREDICATE TRANSFORMERS FOR C11

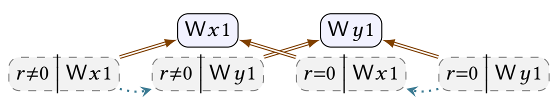
PwT can be used to generate semantic dependencies to prohibit thin-air executions of C11, while preserving optimal lowering for relaxed access. We follow the approach of Paviotti et al. [2020], using our semantics to generate C11 candidate executions with a dependency relation, then applying the axioms of RC11 [Lahav et al. 2017]. The No-Thin-Air axiom of RC11 is overly restrictive, requiring that $\mathsf{rf} \cup \mathsf{po}$ be acyclic. Instead, we require that $\mathsf{rf} \cup <$ is acyclic. This is a more precise 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. Using the obvious construction (described in the proof of Lemma 6.2), po is a pre-order, which may include cycles due to coalescing. For example:

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



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



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

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

(m8) $\pi : (E \to E)$ is an idempotent function capturing *merging*, such that
let $R = \{e \mid \pi(e){=}e\}$ be *real* events, let $\overline{R} = (E \setminus R)$ be *phantom* events,
let $S = \{e \mid \forall d.\ \pi(d){=}e \Rightarrow d{=}e\}$ be *simple* events, let $\overline{S} = (E \setminus S)$ be *compound* events,
(m8a) $\lambda(e) = \lambda(\pi(e))$, (m8b) if $e \in \overline{S}$ then $\kappa(e) \models \bigvee_{\{c \in \overline{R} \mid \pi(c)=e\}} \kappa(c)$.

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

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

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

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

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

We extend the semantics with po as follows. For pomsets with at most one event, po is the identity. For sequential composition, $\mathsf{po} = \mathsf{po}_1 \cup \mathsf{po}_2 \cup E_1 \times E_2$. For parallel composition and the conditional, $\text{po} = \text{po}_1 \cup \text{po}_2$. As noted at the beginning of this section, po may contain cycles. To find an acyclic po′, we replay the construction of P to get P'. When building P', we require disjoint union in s1 and i1: $E' = E'_1 \uplus E'_2$. If and event is unmerged in P (*i.e.* $e \in E_1 \uplus E_2$) then we choose the same event name for E' in P'. If an event is merged in P (*i.e.* $e \in E_1 \cap E_2$) then we choose fresh event names—e'_1 and e'_2—and extend π accordingly: $\pi(e'_1) = \pi(e'_2) = e$. In P', we take $\leq' = \text{po}'$.

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

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

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

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

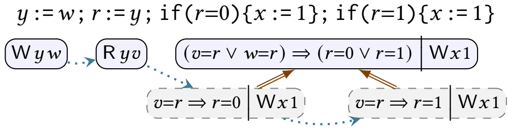
Lemma 6.4. *If P is a complete* PwT-po *with compound event e, then there is a phantom event $c \in \pi^{-1}(e)$ such that $\kappa(c)$ is a tautology.*

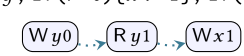
Proof. Immediate from m8b. □

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.

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

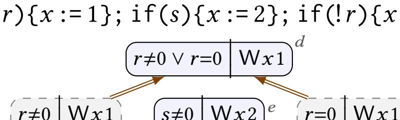




We write $[\![\cdot]\!]^{\text{po}}$ for the semantic function defined by applying the construction of Lemma 6.2 to the base semantics of 1.

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

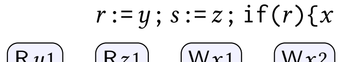
$\texttt{if}(r)\{x := 1\};\ \texttt{if}(s)\{x := 2\};\ \texttt{if}(!r)\{x := 1\}$

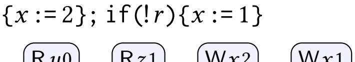


(‡)

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

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

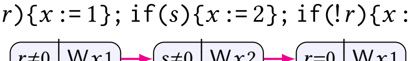




It is somewhat surprising that the writes are independent of both reads!

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:

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

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.

7 PwTer: AUTOMATIC LITMUS TEST EVALUATOR

PwTer automatically and exhaustively calculates the allowed outcomes of litmus tests for the PwT, PwT-po, and PwT-C11 models. 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 §6, implementing PwT-po; and similar to mrd [Paviotti et al. 2020], it can construct C11-style pre-executions and filter them according to the rules of RC11 as described in §6, implementing PwT-C11. Finally, PwTer also allows us to toggle the complete check of 3.4, providing an interface for understanding how fragments of code might compose by exposing preconditions and termination conditions that are not yet tautologies. We have run PwTer over the Java Causality Tests [Pugh 2004] supported in the input syntax, and tabulated the results in Figure 2.

The execution times give a good indication the poor scaling of the tool with program size: for larger test cases, the tool takes exponentially longer to compute, and for the largest tests the memory footprint is too large for even a well-equipped computer. The compositional nature of the semantics makes tool building practical, but it is not enough to make it scalable for large tests. The definitions of $\mathit{SEQ}(\mathcal{P}_1, \mathcal{P}_2)$ and $\mathit{IF}(\phi, \mathcal{P}_1, \mathcal{P}_2)$ – in combination with the rules for reads and writes has exponential complexity. This is compounded by the hidden complexity of calculating the possible merges between pomsets through union in rules s1 and i1. Significant effort has been put into throwing away spurious merges early in PwTer, so that executing the tool remains manageable for small examples. Some further optimisations may be possible within the tool to improve the situation further, such as killing 'dead-end' pomsets at each sequence operator, or by doing a directed search for particular execution outcomes. Unsurprisingly the execution time is dominated by the calculation of the denotation, with the additional axiomatic filtering step of PwT-C11 being within margin of error difference of just calculating the PwT semantics. PwTer is available online at https://github.com/graymalkin/pomsets-with-predicate-transformers.

[Todo: Remove link? — perhaps we put in the appendices.]

[Todo: Verify that referee comments sufficiently addressed regarding "scalability challenges". Perhaps mention that it is easier for hardware models?]

[Todo: Shrink by using two columns and dump PwT?]

Causality Test	PwT-C11		PwT
	Result	Execution Time (s)	Execution Time (s)
jctc1	pass	2.397	2.608
jctc2	pass	25.780	25.754
jctc3	pass	196.935	205.120
jctc4	pass	2.269	2.110
jctc5	pass	63.714	69.441
jctc6	pass	11.245	12.489
jctc7	pass	88.250	96.099
jctc8	pass	2.482	2.473
jctc9	pass	13.592	15.384
jctc10	pass	494.133	513.234
jctc11	⊥	–	–
jctc12	⊥	–	–
jctc13	pass	2.101	2.247
jctc17	pass	178.304	186.228
jctc18	pass	177.292	2.247

Fig. 2. Tool results for supported Java Causality Test Cases [Pugh 2004]. ⊥ indicates the tool failed to run for this test due to a memory overflow. Tests run on an Intel i9-9980HK with 64 GB of memory, execution times are the mean of 3 runs.

8 REFINEMENTS AND ADDITIONAL FEATURES

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

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

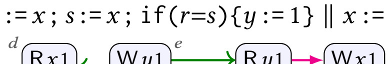
It is worth pointing out that address calculation and if-introduction only affect the semantics of read and write. RMWs introduce new infrastructure in order to ensure atomicity while compiling to Arm8 using load-exclusive and store-exclusive.

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 §5 also apply equally, with the exception of RMWs: we have not proven DRF-SC or Arm8 lowering for RMWs.

8.1 Register Recycling and Redundant Read Elimination

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

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



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

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

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

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

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

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

Definition 8.1. Let $[\![\cdot]\!]$ be defined as in Fig. 1, changing R4 of *READ*:

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

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

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

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

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

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:

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

8.2 Read-Modify-Write Operations

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 8.2. Extend the definition of a pomset as follows.

(M10) $\mathsf{rmw} : E \rightarrow E$ is a partial function capturing read-modify-write *atomicity*, such that
  (M10a) if $d \xrightarrow{\mathsf{rmw}} e$ then $\lambda(e)$ blocks $\lambda(d)$,
  (M10b) if $d \xrightarrow{\mathsf{rmw}} e$ then $d < e$,
  (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 8.3. 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

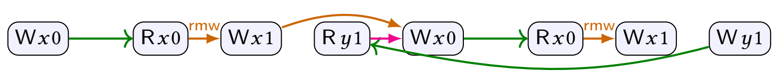
(U10) if $\lambda(e)$ is a write then there is a read $\lambda(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\text{rmw}} e$.

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

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

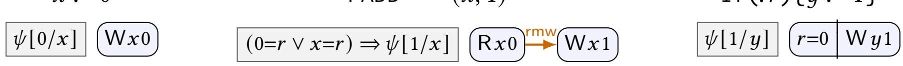
One subtlety of the definition is that we use READ' rather than READ. Thus, for RMW operations, the independent case for a read is the same as the empty case. To see why this should be, consider the relaxed variant of the CDRF example from Lee et al. [2020], using READ rather than READ'.

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



A write should only be visible to one FADD instruction, but here the write of 0 is visible to two. This is allowed because no order is required from (Rx0) to (Wy1) in the last thread. To see why, consider the independent transformers of the last thread and initializer:

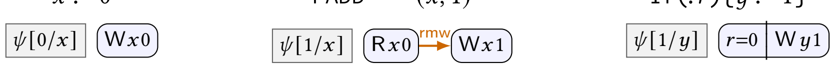
$x := 0$ $\qquad$ $\mathsf{FADD}^{\mathsf{rlx,rlx}}(x, 1)$ $\qquad$ $\mathtt{if}(!r)\{y := 1\}$



After sequencing, the precondition of (Wy1) is a tautology: $(0{=}r \lor 0{=}r) \Rightarrow r{=}0$.

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

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

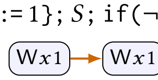


After sequencing, the precondition of (Wy1) is $r{=}0$, which is *not* a tautology. This forces any top-level pomset to include dependency order from (Rx0) to (Wy1).

8.3 If-Introduction (aka Case Analysis)

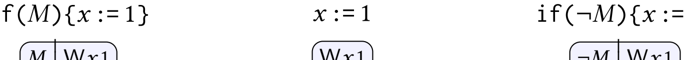
In order to model sequential composition, we must allow inconsistent predicates in a single pomset, unlike PwP. For example, if $S = (x := 1)$, then the semantics Fig. 1 does *not* allow:

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



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:

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



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

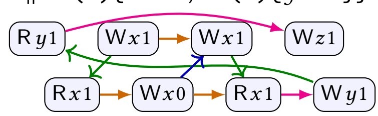
Def. 8.4 allows the execution, by splitting the middle command:

$$\begin{array}{ccc} \texttt{if}(M)\{x := 1\} & x := 1 & \texttt{if}(\neg M)\{x := 1\} \\ {}^{d}\boxed{M \mid \mathsf{W}x1} & {}^{d}\boxed{\neg M \mid \mathsf{W}x1}\ \ {}^{e}\boxed{M \mid \mathsf{W}x1} & {}^{e}\boxed{\neg M \mid \mathsf{W}x1} \end{array}$$

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.

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



We show the rules for write and read.[6] The rule for fences requires similar treatment.

Definition 8.4. If $P \in \mathit{WRITE}(x, M, \mu)$ then $(\exists v : E \rightarrow \mathcal{V})\ (\exists \phi : E \rightarrow \Phi)$

- (w1) if $\kappa(d) \wedge \kappa(e)$ is satisfiable then $d = e$,
- (w2) $\lambda(e) = \mathsf{W}^{\mu} x v_e$,
- (w3) $\kappa(e) \equiv \phi_e \wedge M{=}v_e$,
- (w4) $\tau^D(\psi) \equiv \psi[M/x][\mathsf{K}(E)/\mathsf{Q}_x]$,
- (w5) $\checkmark \equiv \mathsf{K}(E)$,
- (w6) $\phi_e[N/s_d] = \phi_e$.

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

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

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. w6 and R6 require that the predicates do not mention registers in $\mathcal{S}_{\mathcal{E}}$.

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

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

8.4 Address Calculation

[Todo: Check definitions and examples in this subsection.]

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

Definition 8.5. Within a pomset P, let $\mathsf{K}(x) = \bigvee \{\kappa(e) \mid e \in E \wedge \lambda(e) = \mathsf{W}x\}$.

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

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

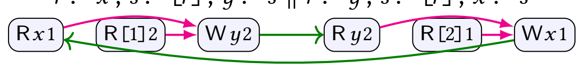
If $P \in \mathit{READ}(r, L, \mu)$ then $(\exists \ell \in \mathcal{V})\ (\exists v \in \mathcal{V})$

[6] The Coq development uses $\vDash$ rather than $\equiv$ in w3 and R3. Given the quantification over ϕ, these are equivalent.

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

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

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

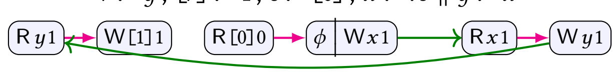


(ADDR1)

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 ($\mathsf{W}x1$).

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:

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

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

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

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:

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

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

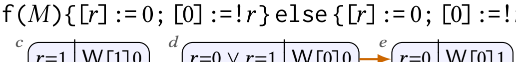
8.5 Combining Address Calculation and If-Introduction

Def. 8.5 is naive with respect to merging events. Consider the following example:

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

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. 8.5.

Def. 8.6, enables this execution using if-introduction. Under this semantics, we have:

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

Sequencing and merging:

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

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

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

Definition 8.6. If $P \in \mathit{WRITE}(L, M, \mu)$ then $(\exists \ell : E \to \mathcal{V})\ (\exists v : E \to \mathcal{V})\ (\exists \phi : E \to \Phi)$

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

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

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

9 RELATED WORK

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

There are 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.

[Todo: Text about PwP.]

[Todo: By the way, I note that PwP doesn't use these termination conditions, and there's no explanation in the paper of why this part of the model has changed, even in the appendix – could you add it?]

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].

[Todo: We agree that a new memory model needs to be positioned against existing models. The usual result here is a compilation correctness to hardware memory models. For PwT-MCA, we address this by showing compilation result for Armv8 model (§5). Comparing software models, however, is unsatisfying: they are all incomparable, i.e., there are examples which are allowed by one/disallowed by the other and vice versa. Morally, our model sits between the strong models (exemplified by RC11 [Lahav-al:PLDI17]) and the speculative models (exemplified by the promising semantics [Kang-al:POPL17]). As we argue in §1: - The strong models require too much synchronization. - The speculative models allow thin air behaviors. Looking at the details, however, PwT-MCA is incomparable to both RC11 and promising semantics. RC11 allows non-MCA behaviors that PwT-MCA disallows. PwT-MCA has a weaker notion of coherence than the promising semantics. Some differences reflect an attempt to fix a bug. For example, Weakestmo [Chakraborty-Vafeiadis:POPL19] purposefully disallows thin-air-like behaviors of the promising semantics. Other differences reflect a different balance between allowed optimizations and reasoning principles. There are fundamental conflicts, for example: - between Common Subexpression Elimination (CSE) and read-read coherence (§D.1) - between if-introduction (aka, case analysis, if-closure) and java-style final-field semantics. (If-introduction requires that address dependencies and control dependencies are the same. Final-fields require that they be different.)] [Todo: Text about promising.]

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 described a tool, PwTer, for automatically evaluating PwT in §7.

10 CONCLUSIONS

This paper is the first to present a direct denotational semantics for sequential composition in a relaxed memory model that can be efficiently compiled to modern CPUs. We extract from this model a semantic dependency relation and use it to build PwT-C11, a solution to the Out-of-Thin-Air problem in C11, and PwT-mca, a model for Java-like languages.

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

PwT-mca does not validate access elimination: store-forwarding and dead-write-removal are unsound. We expect that these can be validated by allowing events with different actions to merge. Nor does PwT-mca validate the reverse inclusions for Lemma 3.6(g). The culprit is delay, which introduces order regardless of whether preconditions are disjoint. As an example, using augmentation, $[\![\mathtt{if}(r)\{x := 1\}\,\mathtt{else}\,\{x := 2\}]\!]$ has an execution with $(r{=}0 \mid \mathsf{W}x2) \rightarrow (r{\neq}0 \mid \mathsf{W}x1)$, whereas $[\![\mathtt{if}(r)\{x := 1\};\ \mathtt{if}(!r)\{x := 2\}]\!]$ has no such execution. We expect that this can be remedied by encoding delay in the logic.

PwT-mca_1 is a simpler model than PwT-mca_2, but requires fences on acquiring reads for Arm8. It would be illuminating to find out what the performance penalty is for these fences.

PwT does not validate read introduction, whereas speculative operational semantics do. Recent work shows a tension between read introduction and compositional reasoning for temporal safety properties (see §9). Nonetheless, read introduction is ubiquitous in some compilers. It would be interesting to know if there is a performance penalty for banning read introduction.

Acknowledgements

This paper has been greatly improved by the comments of the anonymous reviewers.

Riely was supported by the National Science Foundation under grant No. CCR-1617175.

[Todo: Add support for Mark&Simon.]

[Todo: Add support for Anton&Ilya.]

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

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

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

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

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

Alan Jeffrey and James Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*,

M. Grohe, E. Koskinen, and N. Shankar (Eds.). ACM, 759–767. https://doi.org/10.1145/2933575.2934536

Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. Online Appendices for The Leaky Semicolon. (To appear).

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

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

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

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

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

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

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

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

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

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

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

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

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

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