Predicate Transformers for Relaxed Memory: Sequential Composition for Concurrency Using Semantic Dependencies

ANONYMOUS AUTHOR(S)

Program logics and semantics tell us that when executing ((S1; S2), state0), we execute (S1, state0) to arrive at state1, then execute (S2, state1) to arrive at the final state2. This is, of course, and abstraction. Processors execute instructions out of order, due to pipelines and caches. Compilers reorder programs even more dramatically. All of this reordering is meant to be unobservable in single-threaded code. In multi-threaded code, however, all bets are off. A formal attempt to understand the resulting mess is known as a "relaxed memory model." The relaxed memory models that have been proposed to date either fail to address sequential composition, or overly restrict processors and compilers.

To support sequential composition, we propose adding families of predicate transformers to the existing model of "Pomsets with Preconditions," which already supports parallel composition. When composing (S1;S2), the predicate transformer used to validate the precondition of an event in S2 is chosen based on the semantic dependencies from S1 into this event. Our model retains the good properties of the prior work, including efficient implementation on Arm8, support for compiler optimizations, support for logics that prove the absence of thin-air behaviors, and a local data race freedom theorem.

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

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

ACM Reference Format:
Anonymous Author(s). 2021. Predicate Transformers for Relaxed Memory: Sequential Composition for Concurrency Using Semantic Dependencies. *Proc. ACM Program. Lang.* 0, OOPSLA, Article 0 (October 2021), 28 pages.

1 INTRODUCTION

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.

1.1 Sequential Composition

Introductory programmers are taught *sequential abstraction*: that the program $S_1\,;\,S_2$ executes S_1 before S_2. Since the late 60s, 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.

(D1) $wp_{\texttt{skip}}(\psi) = \psi$
(D2a) $wp_{x:=M}(\psi) = \psi[M/x]$
(D2b) $wp_{r:=M}(\psi) = \psi[M/r]$
(D2c) $wp_{r:=x}(\psi) = x{=}r \Rightarrow \psi$
(D3) $wp_{S_1;S_2}(\psi) = wp_{S_1}(wp_{S_2}(\psi))$
(D4) $wp_{\texttt{if}(M)\{S_1\}\texttt{else}\{S_2\}}(\psi) = ((M{\neq}0)\Rightarrow wp_{S_1}(\psi))\wedge((M{=}0)\Rightarrow wp_{S_2}(\psi))$

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2021 Copyright held by the owner/author(s).
2475-1421/2021/10-ART0
https://doi.org/

For this language, the Hoare triple $\{\phi\}\ S\ \{\psi\}$ holds exactly when $\phi \Rightarrow wp_S(\psi)$. We have split Dijkstra's rule for assignment (D2) into three cases. In our notation, r–s range over thread-local registers, which may be assigned at most once, x–z range over shared memory references, and M–N range over thread-local expressions, which do *not* include x–z.[1]

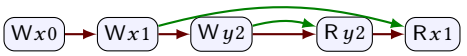
This is quite a pretty explanation of sequential computation in a sequential context. In a concurrent context, however, D2c is unsound! D2c assumes that a read from x must be fulfilled by a preceding write to x. In a concurrent context, writes may come from other threads.

Existing approaches to sequential composition in the concurrent context either assume exclusive access, as in concurrent separation logic [O'Hearn 2007], or abandon the logical approach altogether, as in the pomset model of Kavanagh and Brookes [2018]—this model uses syntactic dependencies and thus dramatically limits compiler optimization. This leaves open the question of how to apply logic to racy programs without overstraining the implementation. To understand the solution, one must first understand the constraints imposed by hardware and compilers.

1.2 Memory Models

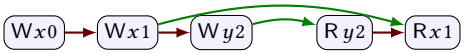
For single-threaded programs, memory can be thought of as you might expect: programs write to, and read from, memory references. This can be thought of as a total order of reads and writes, where each read has a matching *fulfilling* write, for example:

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



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

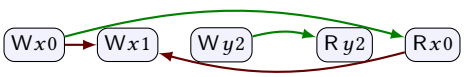
$$x := 0;\ x := 1;\ y := 2 \parallel r := y;\ s := x$$



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

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

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



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

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

[1] Under these assumptions, (D2c) is equivalent to $wp_{r:=x}(\psi) = \psi[x/r]$.

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

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

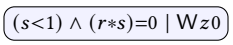
While syntactic dependency calculation suffices for hardware models, it is not preserved by common compiler optimizations. For example, if we calculate control dependencies syntactically, then there is a dependency from $(\mathsf{R}x1)$ to $(\mathsf{W}y1)$, and therefore a cycle in, the candidate execution:

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

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

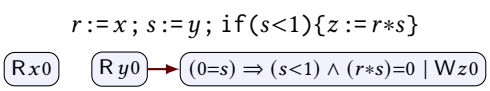
To address this, Jagadeesan et al. [2020] introduced *Pomsets with Preconditions*, where events are labeled with logical formulae. Nontrivial preconditions are introduced by store actions (modeling data dependencies) and conditionals (modeling control dependencies):

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

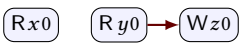


Preconditions are discharged by being ordered after a read:

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



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



1.3 Predicate Transformers For Relaxed Memory

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

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

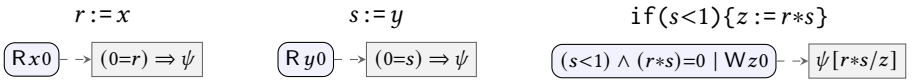
where S is the entire future computation! By formalizing sequential composition, we can show:

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

Then the equivalence holds in any (sequential) context.

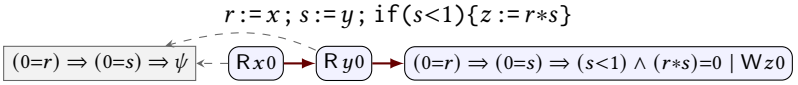
Predicate transformers are a good fit for logical models of dependency calculation, since both are concerned with preconditions and how they are transformed by sequential composition. Our first

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



The predicate transformer from the write matches Dijkstra. For the reads, however, D2C defines the transformer of $r := x$ to be $(x{=}r) \Rightarrow \psi$. Instead, we use $(0{=}r) \Rightarrow \psi$, reflecting the fact that 0 may come from a concurrent write. The obligation to find a matching write is moved from the sequential semantics of *substitution* and *implication* to the concurrent semantics of *fulfillment*.

For a sequentially consistent semantics, sequential composition is straightforward: we apply each predicate transformer to the preconditions of subsequent events, composing the predicate transformers. (In subsequent diagrams, we only show predicate transformers for reads.)

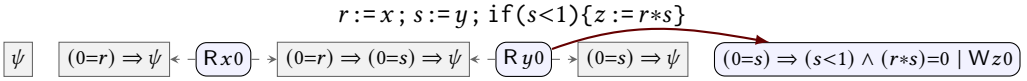


This model works for the sequentially consistent case, but needs to be weakened for the relaxed case. The key observation of this paper is that rather than working with one predicate transformer, we should work with a *family* of predicate transformers, indexed by sets of events.

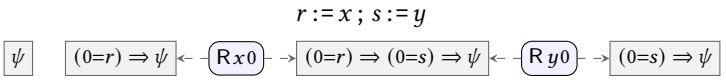
For example, for single-event pomsets, there are two predicate transformers, since there are two subsets of any one-element set. The *independent* transformer is indexed by the empty set, whereas the *dependent* transformer is indexed by the singleton. We visualize this by including more than one transformed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

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

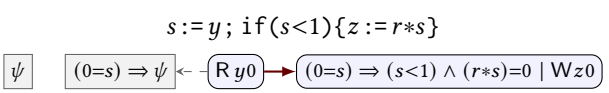
For example, we can recover the expected semantics for (†) by choosing the predicate transformer which is independent of $(\mathsf{R}x0)$ but dependent on $(\mathsf{R}y0)$, which is the transformer which maps ψ to $(0{=}s) \Rightarrow \psi$.



As a sanity check, we can see that sequential composition is associative in this case, since it does not matter whether we associate to the left, with intermediate step:



or to the right, with intermediate step:



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

1.4 Contributions

We show how predicate transformers [Dijkstra 1975] can be added to pomsets with preconditions [Jagadeesan et al. 2020] to create a compositional semantics for sequential composition.

- §3 presents the basic model, which satisfies many desiderata, but not all.
- §4 shows two approaches for efficient implementation on Arm. The first uses a suboptimal lowering for acquiring reads. The second uses an optimal lowering, but requires changes to the definitions of parallel and sequential composition.
- §?? generalizes the basic semantics of read and write to validate compiler optimizations.

Because it is closely related, we expect that the memory-model results of [Jagadeesan et al. 2020] apply to our model, including compositional reasoning for temporal safety properties and local DRF-SC as in [Cho et al. 2021; Dolan et al. 2018; Dongol et al. 2019].

2 RELATED WORK

Marino et al. [2015] argue that the "silently shifting semicolon" sufficiently problematic for programmers that concurrent languages should guarantee sequential abstraction, despite the performance penalties. In this paper, we have 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 prior studies of relaxed memory that include sequential composition and/or precise calculation of semantic dependencies. 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. They use step-indexing to account for loops; we expect that the same approach could be applied here. 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 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]. In their introduction, Jagadeesan et al. [2020] note that these models fail to validate compositional reasoning of temporal properties—see their examples OOTA4 and OOTA5 (from [Lochbihler 2013]). The difference with our model can be understood in terms of the valid program transformations. The speculative models allow reads to be introduced, with subsequent case analysis on the value read—effectively, this can turn one read into two, with different conditional branches taken for the two copies of the read. Our model invalidates this transformation. In return, our model enjoys compositionality for temporal safety properties.

3 THE BASIC MODEL

After some preliminaries, we define the basic model (§3.3 and Fig 1). We explain the model using examples (§3.4–3.6), establish some basic properties (§3.7), and discuss program transformations (§3.8–3.9). We encourage readers to skip to the examples, coming back as needed.

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 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.

$$\mu \ ::= \ \mathsf{rlx} \mid \mathsf{ra} \mid \mathsf{sc} \qquad\qquad \nu \ ::= \ \mathsf{acq} \mid \mathsf{rel} \mid \mathsf{ar}$$

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

Memory modes, μ, are relaxed (rlx), release-acquire (ra), and sequentially consistent (sc). Relaxed mode is the default; we regularly elide it from examples. ra/sc accesses are collectively known as *synchronized accesses*.

Fence modes, ν, are acquire (acq), release (rel), and acquire-release (ar).

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

Throughout §1–4 we require that

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

In §5, we drop this restriction, requiring instead that

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

The semantics is built from the following.

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

We require that:

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

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 $M{\neq}0$. As usual, implication associates to the right; thus $r{=}v \Rightarrow s{>}w \Rightarrow \psi$ is read $(r{=}v) \Rightarrow ((s{>}w) \Rightarrow \psi)$.

We say ϕ is a *tautology* if $\mathsf{tt} \vDash \phi$. We say ϕ is *unsatisfiable* if $\phi \vDash \mathsf{ff}$.

3.2 Actions in This Paper

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

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

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 least order over access and fence modes such that $\mathsf{rlx} \sqsubseteq \mathsf{ra} \sqsubseteq \mathsf{sc}$ and $\mathsf{rel} \sqsubseteq \mathsf{ar}$ and $\mathsf{acq} \sqsubseteq \mathsf{ar}$. We write $(\mathsf{W}^{\sqsupseteq \mathsf{ra}})$ to stand for either $(\mathsf{W}^{\mathsf{ra}})$ or $(\mathsf{W}^{\mathsf{sc}})$, and similarly for the other actions and modes.

Definition 3.1. Actions (R) are *read* actions. Actions $(\mathsf{W}^{\sqsupseteq \mathsf{ra}})$ and $(\mathsf{F}^{\sqsupseteq \mathsf{rel}})$ are *release* 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 *delays* b if $a \bowtie_{\mathsf{co}} b$ or $a \ltimes_{\mathsf{sync}} b$ or $a \bowtie_{\mathsf{sc}} b$.
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 order due to synchronization: $\ltimes_{\mathsf{sync}} = \{(a, \mathsf{W}^{\sqsupseteq \mathsf{ra}}),\ (a, \mathsf{F}^{\sqsupseteq \mathsf{rel}}),\ (\mathsf{R}, \mathsf{F}^{\sqsupseteq \mathsf{acq}}),\ (\mathsf{R} x, \mathsf{R}^{\sqsupseteq \mathsf{ra}} x),\ (\mathsf{R}^{\sqsupseteq \mathsf{ra}}, a),\ (\mathsf{F}^{\sqsupseteq \mathsf{acq}}, a),\ (\mathsf{F}^{\sqsupseteq \mathsf{rel}}, \mathsf{W}),\ (\mathsf{W}^{\sqsupseteq \mathsf{ra}} x, \mathsf{W} x)\}$.
Let $\bowtie_{\mathsf{sc}}$ capture order due to sc access: $\bowtie_{\mathsf{sc}} = \{(\mathsf{W}^{\mathsf{sc}}, \mathsf{W}^{\mathsf{sc}}),\ (\mathsf{R}^{\mathsf{sc}}, \mathsf{W}^{\mathsf{sc}}),\ (\mathsf{W}^{\mathsf{sc}}, \mathsf{R}^{\mathsf{sc}}),\ (\mathsf{R}^{\mathsf{sc}}, \mathsf{R}^{\mathsf{sc}})\}$.

3.3 Pomsets with Predicate Transformers

Predicate transformers are functions on formulae which preserve logical structure, providing a natural model of sequential composition.

Definition 3.2. A *predicate transformer* is a function $\tau : \Phi \to \Phi$ such that
(x1) $\tau(\mathsf{ff})$ is ff,
(x2) $\tau(\psi_1 \wedge \psi_2)$ is $\tau(\psi_1) \wedge \tau(\psi_2)$,
(x3) $\tau(\psi_1 \vee \psi_2)$ is $\tau(\psi_1) \vee \tau(\psi_2)$,
(x4) if $\phi \vDash \psi$, then $\tau(\phi) \vDash \tau(\psi)$.

The definition follows Dijkstra [1975]. Note that substitutions ($\tau(\psi) = \psi[M/r]$ and $\tau(\psi) = \psi[M/x]$) and implications on the right ($\tau(\psi) = \phi \Rightarrow \psi$) 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. We use τ^D as the predicate transformer applied to any event e where if $d \in D$ then $d < e$.

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

Definition 3.4. A *pomset with predicate transformers* over $\mathcal{A}$ is a tuple $(E, \lambda, \kappa, \tau, \checkmark, \mathsf{rf}, \leq)$ where
(M1) $E \subseteq \mathcal{E}$ is a set of *events*,
(M2) $\lambda : E \to \mathcal{A}$ defines a *label* for each event,
(M3) $\kappa : E \to \Phi$ defines a *precondition* for each event,
(M4) $\tau : 2^{\mathcal{E}} \to \Phi \to \Phi$ is a *family of predicate transformers* over E,
(M5) $\checkmark : \Phi$ defines a *termination condition*,
(M6) $\mathsf{rf} : E \to E$ is an injective relation capturing *reads-from* such that
  (M6a) if $d \xrightarrow{\mathsf{rf}} e$ then $\lambda(d)$ matches $\lambda(e)$,
(M7) $\leq : E \times E$, is a partial order capturing *causality*, such that
  (M7a) if $d \xrightarrow{\mathsf{rf}} e$ and $\lambda(c)$ blocks $\lambda(e)$ then either $c \leq d$ or $e \leq c$.

A pomset is *top-level* if $\checkmark$ is a tautology and for every $e \in E$,
(T1) $\kappa(e)$ is a tautology,
(T2) if $\lambda(e)$ is a read then there is some $d \xrightarrow{\mathsf{rf}} e$.

We give the semantics of programs in Fig 1.

Suppose $R_1 : E_1 \times E_1$ and $R_2 : E_2 \times E_2$.
We say R *extends* R_1 *and* R_2 if $R \supseteq (R_1 \cup R_2)$ and $R \cap (E_1 \times E_1) = R_1$ and $R \cap (E_2 \times E_2) = R_2$.

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

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

(P1) $E = (E_1 \uplus E_2)$,
(P2) $\lambda = (\lambda_1 \cup \lambda_2)$,
(P3a) if $e \in E_1$ then $\kappa(e) \vDash \kappa_1(e)$,
(P3b) if $e \in E_2$ then $\kappa(e) \vDash \kappa_2(e)$,
(P4) $\tau^D(\psi) \vDash \tau_1^D(\psi)$,
(P5) $\checkmark \vDash \checkmark_1 \wedge \checkmark_2$,
(P6) rf extends rf_1 and rf_2,
(P7a) $\leq$ extends $\leq_1$ and $\leq_2$,
(P7b) if $d \in E_1$, $e \in E_2$ and $d \xrightarrow{\mathsf{rf}} e$ then $d \leq e$.

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

(C1) $E = (E_1 \cup E_2)$,
(C2) $\lambda = (\lambda_1 \cup \lambda_2)$,
(C3a) if $e \in E_1 \setminus E_2$ then $\kappa(e) \vDash \phi \wedge \kappa_1(e)$,
(C3b) if $e \in E_2 \setminus E_1$ then $\kappa(e) \vDash \neg\phi \wedge \kappa_2(e)$,
(C3c) if $e \in E_1 \cap E_2$ then $\kappa(e) \vDash (\phi \wedge \kappa_1(e)) \vee (\neg\phi \wedge \kappa_2(e))$,
(C4) $\tau^D(\psi) \vDash (\phi \wedge \tau_1^D(\psi)) \vee (\neg\phi \wedge \tau_2^D(\psi))$,
(C5) $\checkmark \vDash (\phi \wedge \checkmark_1) \vee (\neg\phi \wedge \checkmark_2)$.
(C6a) rf extends rf_1 and rf_2,
(C6b) $\mathsf{rf} \subseteq (\mathsf{rf}_1 \cup \mathsf{rf}_2)$,
(C7a) $\leq$ extends $\leq_1$ and $\leq_2$,
(C7b) $\leq \subseteq (\leq_1 \cup \leq_2)$.

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

(S1) $E = (E_1 \cup E_2)$,
(S2) $\lambda = (\lambda_1 \cup \lambda_2)$,
(S3a) if $e \in E_1 \setminus E_2$ then $\kappa(e) \vDash \kappa_1(e)$,
(S3b) if $e \in E_2 \setminus E_1$ then $\kappa(e) \vDash \kappa_2'(e)$,
(S3c) if $e \in E_1 \cap E_2$ then $\kappa(e) \vDash \kappa_1(e) \vee \kappa_2'(e)$,
(S3d) if $\lambda_2(e)$ is a release then $\kappa(e) \vDash \checkmark_1$,
(S4) $\tau^D(\psi) \vDash \tau_1^D(\tau_2^D(\psi))$,
(S5) $\checkmark \vDash \checkmark_1 \wedge \tau_1(\checkmark_2)$,
(S6) rf extends rf_1 and rf_2,
(S7a) $\leq$ extends $\leq_1$ and $\leq_2$,
(S7b) if $d \in E_1$, $e \in E_2$ and $d \xrightarrow{\mathsf{rf}} e$ then $d \leq e$,
(S7c) if $\lambda_1(d)$ delays $\lambda_2(e)$ then $d \leq e$.

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

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

(R1) if $d, e \in E$ then $d = e$,
(R2) $\lambda(e) = \mathsf{R}^\mu x v$,
(R4a) if $(E \cap D) \neq \emptyset$ then $\tau^D(\psi) \vDash v{=}r \Rightarrow \psi$,
(R4b) if $E \neq \emptyset$ and $(E \cap D) = \emptyset$ then $\tau^D(\psi) \vDash (v{=}r \vee x{=}r) \Rightarrow \psi$.
(R4c) if $E = \emptyset$ then $\tau^D(\psi) \vDash \psi$.

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

(W1) if $d, e \in E$ then $d = e$,
(W2) $\lambda(e) = \mathsf{W}^\mu x v$,
(W3) $\kappa(e) \vDash M{=}v$,
(W4) $\tau^D(\psi) \vDash \psi[M/x]$,
(W5a) if $E \neq \emptyset$ then $\checkmark \vDash M{=}v$,
(W5b) if $E = \emptyset$ then $\checkmark \vDash \mathsf{ff}$.

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

(F1) if $d, e \in E$ then $d = e$,
(F2) $\lambda(e) = \mathsf{F}^\mu$,
(F4) $\tau^D(\psi) \vDash \psi$,
(F5) if $E = \emptyset$ then $\checkmark \vDash \mathsf{ff}$.

$$[\![r := M]\!]_1 = \mathit{LET}(r, M) \qquad [\![\mathtt{skip}]\!]_1 = \mathit{SKIP}$$
$$[\![r := x^\mu]\!]_1 = \mathit{READ}(r, x, \mu) \qquad [\![S_1 \parallel S_2]\!]_1 = [\![S_1]\!]_1 \parallel [\![S_2]\!]_1$$
$$[\![x^\mu := M]\!]_1 = \mathit{WRITE}(x, M, \mu) \qquad [\![S_1 ; S_2]\!]_1 = [\![S_1]\!]_1 ; [\![S_2]\!]_1$$
$$[\![\mathsf{F}^\nu]\!]_1 = \mathit{FENCE}(\nu) \qquad [\![\mathtt{if}(M)\{S_1\}\mathtt{else}\{S_2\}]\!]_1 = \mathit{IF}(M{\neq}0, [\![S_1]\!]_1, [\![S_2]\!]_1)$$

Fig. 1. Semantics of programs

3.4 Examples: Pomsets

3.5 Examples: Pomsets with Preconditions

3.6 Examples: Pomsets with Predicate Transformers

3.7 Basic Properties

Lemma 3.5. *For any P in the range of $[\![\cdot]\!]_1$, $d \xrightarrow{\mathsf{rf}} e$ implies $d \leq e$.*
Proof. Induction on the definition of $[\![\cdot]\!]_1$. □

The semantics to be closed with respect to *augmentation* Augments include more order and stronger formulae; in examples, we typically consider pomsets that are augment-minimal. One intuitive reading of augment closure is that adding order can only cause preconditions to weaken.

Definition 3.6. P_2 is an *augment* of P_1 if
(1) $E_2 = E_1$, (3) $\kappa_2(e) \vDash \kappa_1(e)$, (5) $\checkmark_2 \vDash \checkmark_1$, (7) $\leq_2 \supseteq \leq_1$.
(2) $\lambda_2(e) = \lambda_1(e)$, (4) $\tau_2^D(e) \vDash \tau_1^D(e)$, (6) $\mathsf{rf}_2 = \mathsf{rf}_1$,

Lemma 3.7. *If $P_1 \in [\![S]\!]_1$ and P_2 augments P_1 then $P_2 \in [\![S]\!]_1$.*
Proof. Induction on the definition of $[\![\cdot]\!]_1$. □

Lemma 3.8. *$(\mathcal{P}_1 ; \mathcal{P}_2) ; \mathcal{P}_3 = \mathcal{P}_1 ; (\mathcal{P}_2 ; \mathcal{P}_3)$ and $\mathcal{P} ; \mathtt{skip} = \mathcal{P} = \mathtt{skip} ; \mathcal{P}$.*
$(\mathcal{P}_1 \parallel \mathcal{P}_2) \parallel \mathcal{P}_3 = \mathcal{P}_1 \parallel (\mathcal{P}_2 \parallel \mathcal{P}_3)$ and $\mathcal{P} \parallel \mathtt{skip} = \mathcal{P}$.
Proof. Straightforward calculation. Associativity of ; requires disjunction closure (x3). □

Note that E_1 and E_2 are not necessarily disjoint. In *IF*, the definition of *extends* stops coalescing the rf in

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

We have given the semantics of *IF* using disjunctive normal form. Dijkstra [1975] used conjunctive normal form. Note that $(\phi \wedge \theta_1) \vee (\neg\phi \wedge \theta_2)$ is logically equivalent to $(\phi \Rightarrow \theta_1) \wedge (\neg\phi \Rightarrow \theta_2)$.

3.8 Valid Transformations

3.9 Invalid Transformations

4 ARM

For simplicity, we restrict to top level parallel composition and ignore fences[2].

4.1 Arm executions

Definition 4.1. An *Arm8 execution graph*, G, is tuple $(E, \lambda, \mathsf{poloc}, \mathsf{lob})$ such that
(A1) $E \subseteq \mathcal{E}$ is a set of events,
(A2) $\lambda : E \to \mathcal{A}$ defines a label for each event,
(A3) $\mathsf{poloc} : E \times E$, is a per-thread, per-location total order, capturing *per-location program order*,
(A4) $\mathsf{lob} : E \times E$, is a per-thread partial order capturing *locally-ordered-before*, such that
  (A4a) $\mathsf{poloc} \cup \mathsf{lob}$ is acyclic.

The definition of lob is complex. Comparing with our definition of sequential composition, it is sufficient to note that lob includes
(L1) read-write dependencies, required by s3,
(L2) synchronization delay of $\ltimes_{\mathsf{sync}}$, required by s7c,
(L3) sc access delay of $\bowtie_{\mathsf{sc}}$, required by s7c,
(L4) write-write and read-to-write coherence delay of $\ltimes_{\mathsf{co}}$, required by s7c,

[2]Fences are not actions in Arm8, which complicates the theorem statements.

and that lob does *not* include

(L5) read-read control dependencies, required by s3,
(L6) write-to-read order of rf, required by s7b,
(L7) write-to-read coherence delay of $\ltimes_{\mathsf{co}}$, required by s7c.

Definition 4.2. Execution G is (co, rf, gcb)*-valid*, under *External Global Consistency* (EGC) if

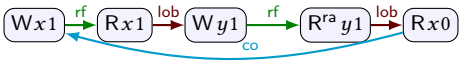
(A5) $\mathsf{co} : E \times E$, is a per-location total order on writes, capturing *coherence*,
(A6) $\mathsf{rf} : E \times E$, is a surjective and injective relation on reads, capturing *reads-from*, such that
  (A6a) if $d \xrightarrow{\mathsf{rf}} e$ then $\lambda(d)$ matches $\lambda(e)$,
  (A6b)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

Execution G is (co, rf, cb)*-valid* under *External Consistency* (EC) if

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

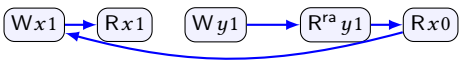
[Alglave et al. 2021] explain EGC and EC using the following example.[3]

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



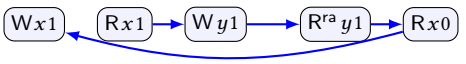
(✓Arm8)

EGC drops lob-order in the first thread using 4.4, since $(\mathsf{W}x1)$ is not lob-ordered before $(\mathsf{W}y1)$.



(gcb)

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



(cb)

4.2 Arm Compilation 1

Podkopaev et al. [2019] lowers to Arm8 as follows: Relaxed access is implemented using `ldr`/`str`, non-relaxed access using `ldar`/`stlr`. In this section, we consider a suboptimal strategy, which lowers non-relaxed reads to (`dmb.sy`; `ldar`).

We do not distinguish control dependencies from address dependencies, and therefore L5 forces us to drop all dependencies between reads. To achieve this, we modify the definition of κ_2' in Fig 1.

Definition 4.3. Let $[\![\cdot]\!]_2$ be defined as in Fig 1, replacing the definition of κ_2' with:

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

THEOREM 4.4. *Suppose G_1 is $(\mathsf{co}_1, \mathsf{rf}_1, \mathsf{gcb}_1)$-valid for S under the suboptimal lowering that maps non-relaxed reads to* (`dmb.sy`; `ldar`)*. Then there is a top-level pomset $P_2 \in [\![S]\!]_2$ such that $E_2 = E_1$, $\lambda_2 = \lambda_1$, $\mathsf{rf}_2 = \mathsf{rf}_1$, and $\leq_2 = \mathsf{gcb}_1$.*

PROOF. First, we establish some lemmas about Arm8.

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

Lemma 4.5. *Suppose G is $(\mathsf{co}, \mathsf{rf}, \mathsf{gcb})$-valid. Then $\mathsf{gcb} \supseteq \mathsf{fr}$.*

Proof. Using the definition of fr from A6b, we have $e \xleftarrow{\mathsf{rf}} d \xrightarrow{\mathsf{co}} c$, and therefore $\lambda(c)$ blocks $\lambda(e)$. Applying A7a, we have that either $c \xrightarrow{\mathsf{gcb}} d$ or $e \xrightarrow{\mathsf{gcb}} c$. Since gcb includes co, we have $d \xrightarrow{\mathsf{gcb}} c$, and therefore it must be that $e \xrightarrow{\mathsf{gcb}} c$. □

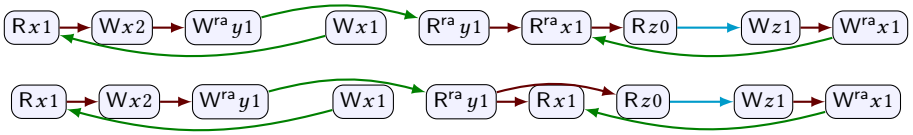
Lemma 4.6. *Suppose G is $(\mathsf{co}, \mathsf{rf}, \mathsf{gcb})$-valid and $c \xrightarrow{\mathsf{poloc}} e$, where $\lambda(c)$ blocks $\lambda(e)$. Then $c \xrightarrow{\mathsf{gcb}} e$.*

Proof. By way of contradiction, assume $e \xrightarrow{\mathsf{gcb}} c$. If $c \xrightarrow{\mathsf{rf}} e$ then by A7 we must also have $c \xrightarrow{\mathsf{gcb}} e$, contradicting the assumption that gcb is a total order. Otherwise that there is some $d \neq c$ such that $d \xrightarrow{\mathsf{rf}} e$, and therefore $d \xrightarrow{\mathsf{gcb}} e$. By transitivity, $d \xrightarrow{\mathsf{gcb}} c$. By the definition of fr, we have $e \xrightarrow{\mathsf{fr}} c$. But this contradicts A6b, since $c \xrightarrow{\mathsf{poloc}} e$. □

We show that all the order required in the pomset is also required by Arm8. M7a holds since cb_1 is consistent with co_1 and fr_1. As noted above, lob includes the order required by s3 and s7c. We need only show that the order removed from 4.4 can also be removed from the pomset. In order for to remove order from e to c, we must have $d \xrightarrow{\mathsf{rf}} e$ and $d \xrightarrow{\mathsf{poloc}} e$ but not $d \xrightarrow{\mathsf{lob}} c$. Because of our suboptimal lowering, it must be that e is a relaxed read; otherwise the `dmb.sy` would require $d \xrightarrow{\mathsf{lob}} c$. Thus we know that s7c does not require order from e to c. By chaining R4b and W4, any dependence on the read can by satisfied without introducing order in s3. □

Downgrading messes up publication:

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



4.3 Arm Compilation 2

Definition 4.7. Let $[\![\cdot]\!]_2^{\mathsf{rf}}$ be defined as for $[\![\cdot]\!]_2$ in Def 4.3 and Fig 1, changing s7b and s7c:

(s7b$^{\mathsf{rf}}$) if $\lambda_1(c)$ blocks $\lambda_2(e)$ then $d \xrightarrow{\mathsf{rf}} e$ implies $c \leq d$,
(s7c$^{\mathsf{rf}}$) if $\lambda_1(d)$ delays$'$ $\lambda_2(e)$ then $d \leq e$,
where *delays*$'$ replaces $\bowtie_{\mathsf{co}}$ in Def 3.1 of *delays* by $\bowtie_{\mathsf{lws}} = \{(\mathsf{W}x, \mathsf{W}x),\ (\mathsf{R}x, \mathsf{W}x)\}$.

The acronym lws is adopted from Arm8. It stands for *Local Write Successor*.

Note that Lem 3.5 fails for $[\![\cdot]\!]_2^{\mathsf{rf}}$, since $d \xrightarrow{\mathsf{rf}} e$ may not imply $d \leq e$ when d and e come from different sides of a sequential composition. This means that rf must be verified during pomset construction, rather than post-hoc. If one wants a post-hoc verification technique for rf, it is possible to include program order (po) in the pomset.

Example 4.8. The obvious definition of po may be cyclic, due to the conditional.

Lemma 4.9. *P is top-level iff $d \xrightarrow{\mathsf{rf}} e$ implies either*

- *external fulfillment: $d \leq e$ and if $\lambda(c)$ blocks $\lambda(e)$ then either $c \leq d$ or $e \leq c$, or*
- *internal fulfillment: $d \xrightarrow{\mathsf{po}} e$ and $(\nexists c)$ $\lambda(c)$ blocks $\lambda(e)$ and $d \xrightarrow{\mathsf{po}} c \xrightarrow{\mathsf{po}} e$.*

Theorem 4.10. *Suppose G_1 is EC-valid for S via $(\mathsf{co}_1, \mathsf{rf}_1, \mathsf{cb}_1)$ and that $\mathsf{cb}_1 \supseteq \mathsf{fr}_1$. Then there is a top-level pomset $P_2 \in [\![S]\!]_2^{\mathsf{rf}}$ such that $E_2 = E_1$, $\lambda_2 = \lambda_1$, $\mathsf{rf}_2 = \mathsf{rf}_1$, and $\leq_2 = \mathsf{cb}_1$.*

Proof. We show that all the order required in the pomset is also required by Arm8. M7a holds since cb_1 is consistent with co_1 and fr_1. s7b$^{\mathsf{rf}}$ follows from A8b. As noted above, lob includes the order required by s3 and s7c$^{\mathsf{rf}}$. □

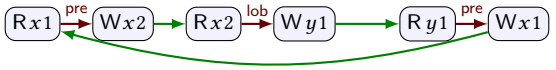
The generality of Thm 4.10 is not limited by the assumption that $\mathsf{cb}_1 \supseteq \mathsf{fr}_1$:

Lemma 4.11. *Suppose G is* EC*-valid via* (co, rf, cb)*. Then there a permutation* cb′ *of* cb *such that G is* EC*-valid via* (co, rf, cb′) *and* cb′ ⊇ fr*, where* fr *is defined in* A6b*.*

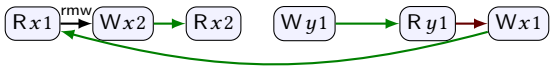
Proof. We show that any cb order that contradicts fr is incidental.
By definition of fr, $e \xleftarrow{\mathsf{rf}} d \xrightarrow{\mathsf{co}} c$, for some d. Since cb ⊇ co, we know that $d \xrightarrow{\mathsf{co}} c$.
If A8a applies to $d \xrightarrow{\mathsf{rf}} e$, then $e \xrightarrow{\mathsf{cb}} c$, since it cannot be that $c \xrightarrow{\mathsf{co}} d$.
Suppose A8b applies to $d \xrightarrow{\mathsf{rf}} e$ and c is from a different thread. Because it is a different thread, we cannot have $e \xrightarrow{\mathsf{lob}} c$, and thus the order in cb is incidental.
Suppose A8b applies to $d \xrightarrow{\mathsf{rf}} e$ and c is from the same thread. Since $c \xrightarrow{\mathsf{co}} d$, it cannot be that $c \xrightarrow{\mathsf{poloc}} d$, using A6b. It also cannot be that $d \xrightarrow{\mathsf{poloc}} c \xrightarrow{\mathsf{poloc}} e$. It must be that $e \xrightarrow{\mathsf{poloc}} c$. By A4a, we cannot have $e \xrightarrow{\mathsf{lob}} c$, and thus the order in cb is incidental. □

Bad example:

$$r := \mathsf{EXCHG}(x, 2)\,;\, s := x\,;\, y := s-1 \;\|\; r := y\,;\, x := r$$



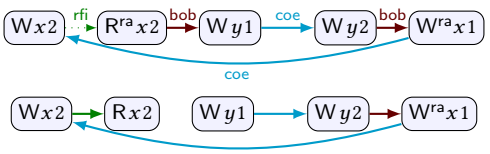
(✓Arm8)



(≤)

Anton example 1 `[rfi-coe-coe]`

$$x := 2\,;\, r := x^{\mathsf{ra}}\,;\, y := 1 \;\|\; y := 2\,;\, x^{\mathsf{ra}} := 1 \qquad \text{(RFI-COE-COE)}$$



(✓Arm8)

5 ADDITIONAL FEATURES

5.1 Register Recycling

Definition 5.1. Let $[\![\cdot]\!]_3$ be defined as for $[\![\cdot]\!]_2$ in Def 4.3 and Fig 1, changing R4 of *READ*:
(R4a) if $(E \cap D) \neq \emptyset$ then $\tau^D(\psi) \vDash v{=}s_e \Rightarrow \psi[s_e/r]$,
(R4b) if $E \neq \emptyset$ and $(E \cap D) = \emptyset$ then $\tau^D(\psi) \vDash (v{=}s_e \vee x{=}s_e) \Rightarrow \psi[s_e/r]$.
(R4c) $(\forall s)$ if $E = \emptyset$ then $\tau^D(\psi) \vDash \psi[s/r]$.

Similarly, let $[\![\cdot]\!]_3^{\mathsf{rf}}$ be defined as for $[\![\cdot]\!]_2^{\mathsf{rf}}$ in Def 4.7, with this definition of *READ*.

The semantics considered thus far assume that each register is assigned at most once in a program. We relax this by renaming.

Example 5.2. JMM causality Test Case 2 [Pugh 2004] states the following execution should be allowed "since redundant read elimination could result in simplification of $r{=}s$ to true, allowing $y := 1$ to be moved early."

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

This execution is not allowed under Def **??**, since the precondition of $(\mathsf{W}y1)$ in the independent case is

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

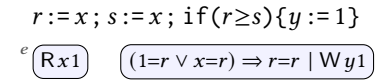
which is not a tautology. Our solution is to rename registers using the set $\mathcal{S}_{\mathcal{E}} = \{s_e \mid e \in \mathcal{E}\}$, which are banned from source programs, as per §3.1. This allows us to resolve nondeterminism in loads when merging, resulting in:

Definition 5.3 (ALPHA). Update Def **??** to:

?? $\tau^D(\psi)$ implies $v{=}s_e \Rightarrow \psi[s_e/r]$,
?? $(\forall s)\ \tau^C(\psi)$ implies $\psi[s/r]$.

Example 5.4. Revisiting Ex 5.2 and choosing $s_e = r$:

Coalescing and composing:



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

5.2 If-Closure

Definition 5.5. Let $[\![\cdot]\!]_4$ be defined as for $[\![\cdot]\!]_2$ in Def 4.3 and Fig 1, changing *WRITE* and *READ*.

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

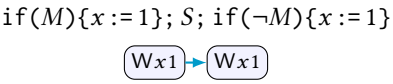
(w1) if $\theta_d \land \theta_e$ is satisfiable then $d = e$,
(w2) $\lambda(e) = \mathsf{W}^{\mu} x v_e$,
(w3) $\kappa(e) \vDash \theta_e \land M{=}v_e$,
(w4) $\tau^D(\psi) \vDash \theta_e \Rightarrow \psi[M/x]$,
(w5) $\checkmark \vDash \theta_e \Rightarrow M{=}v_e$,

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

(R1) if $\theta_d \land \theta_e$ is satisfiable then $d = e$,
(R2) $\lambda(e) = \mathsf{R}^{\mu} x v_e$
(R3) $\kappa(e) \vDash \theta_e$,
(R4a) $(\forall e \in E \cap D)\ \tau^D(\psi) \vDash \theta_e \Rightarrow v_e{=}s_e \Rightarrow \psi[s_e/r]$,
(R4b) $(\forall e \in E \setminus D)\ \tau^D(\psi) \vDash \theta_e \Rightarrow (v_e{=}s_e \lor x{=}s_e) \Rightarrow \psi[s_e/r]$,
(R4c) $(\forall s)\ \tau^D(\psi) \vDash (\bigwedge_{e \in E} \neg\theta_e) \Rightarrow \psi[s/r]$.

Similarly, let $[\![\cdot]\!]_4^{\mathsf{rf}}$ be defined as for $[\![\cdot]\!]_2^{\mathsf{rf}}$ in Def 4.7, with these definitions of *WRITE* and *READ*.

Example 5.6. If $S = (x := 1)$, then Def **??** does *not* allow:



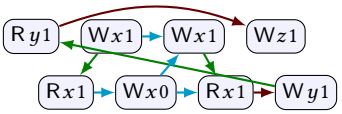
However, if $S = (\texttt{if}(\neg M)\{x := 1\};\ \texttt{if}(M)\{x := 1\})$, then it *does* allow the execution. Looking at the initial program:

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

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.

This is not simply a theoretical question; it is observable. For example, Def ?? does not allow the following.

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



Definition 5.7 (ALPHA/IF). Update Def ?? to:
If $P \in \mathit{WRITE}(x, M, \mu)$ then $(\exists v : E \to \mathcal{V})\ (\exists \theta : E \to \Phi)$
?? if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
?? $\lambda(e) = \mathsf{W}xv_e$,
?? $\kappa(e)$ implies $\theta_e \wedge M{=}v$,
?? $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow (\psi \wedge M{=}v)$,
?? $\tau^C(\psi)$ implies $(\nexists e \in E \cap C \mid \theta_e) \Rightarrow \psi$,
If $P \in \mathit{READ}(r, x, \mu)$ then $(\exists v : E \to \mathcal{V})\ (\exists \theta : E \to \Phi)$
?? if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
?? $\lambda(e) = \mathsf{R}xv_e$,
?? $\kappa(e)$ implies θ_e.
?? $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow v_e{=}s_e \Rightarrow \psi[s_e/r]$,
?? $(\forall s)\ \tau^C(\psi)$ implies $(\nexists e \in E \mid \theta_e) \Rightarrow \psi[s/r]$.

Example 5.8. Revisiting Ex 5.6, we can split the middle command:

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

Coalescing events gives the desired result.

These examples show that we must allow inconsistent predicates in a single pomset, unlike [Jagadeesan et al. 2020].

5.3 Address Calculation

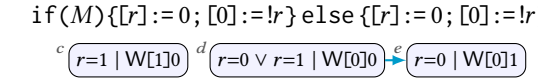
Definition 5.9. Let $[\![\cdot]\!]_5$ be defined as for $[\![\cdot]\!]_2$ in Def 4.3 and Fig 1, changing *WRITE* and *READ*.
If $P \in \mathit{WRITE}(L, M, \mu)$ then $(\exists \ell : E \to \mathcal{V})\ (\exists v : E \to \mathcal{V})\ (\exists \theta : E \to \Phi)$
(w1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
(w2) $\lambda(e) = \mathsf{W}^{\mu}[\ell]v_e$,
(w3) $\kappa(e) \vDash \theta_e \wedge L{=}\ell_e \wedge M{=}v_e$,
(w4a) $\tau^D(\psi) \vDash \theta_e \Rightarrow \psi[M/[\ell]]$,
(w4b) $(\forall k)$ $\tau^D(\psi) \vDash (\bigwedge_{e \in E} \neg\theta_e) \Rightarrow (L{=}k) \Rightarrow \psi[M/[k]]$
(w5a) $\checkmark \vDash \theta_e \Rightarrow L{=}\ell_e \wedge M{=}v_e$,
(w5b) $\checkmark \vDash \bigvee_{e \in E} \theta_e$.
If $P \in \mathit{READ}(r, L, \mu)$ then $(\exists \ell : E \to \mathcal{V})\ (\exists v : E \to \mathcal{V})\ (\exists \theta : E \to \Phi)$
(R1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,

(R2) $\lambda(e) = \mathsf{R}^{\mu}[\ell]v_e$
(R3) $\kappa(e) \vDash \theta_e \wedge L{=}\ell_e$,
(R4a) $(\forall e \in E \cap D)\ \tau^D(\psi) \vDash \theta_e \Rightarrow (L{=}\ell_e \Rightarrow v_e{=}s_e) \Rightarrow \psi[s_e/r]$,
(R4b) $(\forall e \in E \setminus D)\ \tau^D(\psi) \vDash \theta_e \Rightarrow ((L{=}\ell_e \Rightarrow v_e{=}s_e) \vee (L{=}\ell_e \Rightarrow [\ell]{=}s_e)) \Rightarrow \psi[s_e/r]$,
(R4c) $(\forall s)\ \tau^D(\psi) \vDash (\bigwedge_{e\in E} \neg\theta_e) \Rightarrow \psi[s/r]$.

Similarly, let $[\![\cdot]\!]^{\mathsf{rf}}_5$ be defined as for $[\![\cdot]\!]^{\mathsf{rf}}_2$ in Def 4.7, with these definitions of *WRITE* and *READ*.

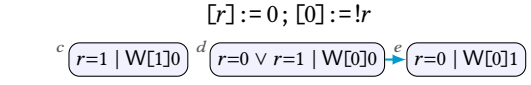
Example 5.10. Def **??** is naive with respect to merging events. Consider the following example from [Jagadeesan et al. 2020]:

Merging, we have:



The precondition of W[0]0 is a tautology; however, this is not possible for ([r] := 0 ; [0] := !r) alone, using Def **??**. The full semantics, given in Fig **??**, enables this execution using if-closure. The individual commands have the pomsets:

Sequencing and merging, we have:



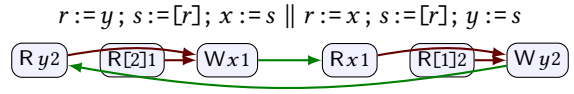
The precondition of (W[0]0) is a tautology, as required.

Example 5.11. The combination of read-read independency and address calculation (ADDR/RRD) is somewhat delicate. Combing Def **??** and Def **??** we have:

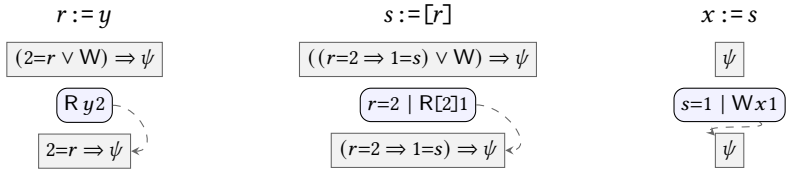
?? $\tau^D(\psi) \vDash (L{=}\ell \Rightarrow v{=}r) \Rightarrow \psi$,
?? $\tau^C(\psi) \vDash ((L{=}\ell \Rightarrow v{=}r) \vee \mathsf{W}) \Rightarrow \psi$.

If we replace the use of $(L{=}\ell \Rightarrow v{=}r)$ by $(v{=}r)$, thin air reads are possible. The subsection of §B on Causal Strengthening discusses this example using the semantics of [Jagadeesan et al. 2020].

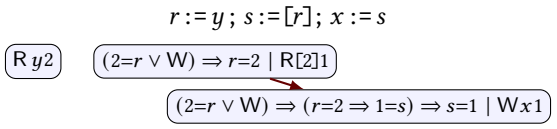
Consider the following program, from [Jagadeesan et al. 2020, §5], where initially $x = 0$, $y = 0$, $[0] = 0$, $[1] = 2$, and $[2] = 1$. It should only be possible to read 0, disallowing the attempted execution below:



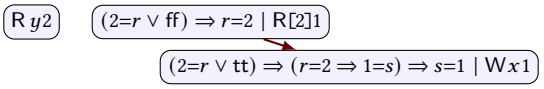
Looking at the left thread:



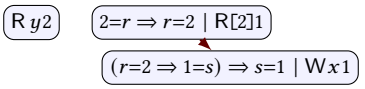
Composing, we have:



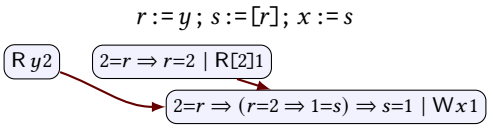
Substituting for W:



Which is:



The precondition of (R[2]1) is a tautology, but the precondition of (Wx1) is not. This forces a dependency:



All the preconditions are now tautologies.

5.4 Access Elimination

For reads, get rid of ff/Q in **??**.

For writes, change the label rules of sequential composition to:

(1) if $e \in E_1 \setminus E_2$ then $\lambda(e) = \lambda_1(e)$,
(2) if $e \in E_2 \setminus E_1$ then $\lambda(e) = \lambda_2(e)$,
(3) if $e \in E_1 \cap E_2$ then $\lambda(e) \in \mathsf{merge}(\lambda_1(e),\ \lambda_2(e))$.

Definition 5.12.

$$\mathsf{merge}(\mathsf{R}^{\mu} x v,\ \mathsf{R}^{\nu} x v) = \{\mathsf{R}^{\mu \sqcup \nu} x v\}$$
$$\mathsf{merge}(\mathsf{W}^{\mu} x v,\ \mathsf{W}^{\nu} x w) = \{\mathsf{W}^{\mu \sqcup \nu} x w\}$$
$$\mathsf{merge}(\mathsf{F}^{\mu},\ \mathsf{F}^{\nu}) = \{\mathsf{F}^{\mu \sqcup \nu}\}$$
$$\mathsf{merge}(a,\ b) = \emptyset,\ \text{otherwise}$$

5.5 Merging Different labels

Reordering and Merging: [Kang 2019, §7.1] [Chakraborty and Vafeiadis 2017, §E]

Examples of Unsafe Reorderings [Chakraborty and Vafeiadis 2017, §D] See the slides for this paper...

Note that for associativity, you have to take the join of modes.

Definition 5.13. Define $\mathsf{merge} : \mathcal{A} \times \mathcal{A} \to 2^{\mathcal{A}}$ as follows. If $a_0 \in \mathsf{merge}(a_1,\ a_2)$, then a_1 and a_2 can coalesce, resulting in a_0. This is useful for replacing $(x := 1\,;\ x := 2)$ by $(x := 2)$.

$$\begin{aligned}
\mathsf{merge}(\mathsf{R}^{\mu} x v,\ \mathsf{R}^{\nu} x v) &= \{\mathsf{R}^{\mu \sqcup \nu} x v\} \\
\mathsf{merge}(\mathsf{W}^{\mu} x v,\ \mathsf{W}^{\nu} x w) &= \{\mathsf{W}^{\mu \sqcup \nu} x w\} \\
\mathsf{merge}(\mathsf{W}^{\mu} x v,\ \mathsf{R}^{\mathsf{rlx}} x v) &= \{\mathsf{W}^{\mu} x v\} \\
\mathsf{merge}(\mathsf{W}^{\nu} x v,\ \mathsf{R}^{\sqsupseteq \mathsf{ra}} x v) &= \{\mathsf{W}^{\mathsf{sc}} x v\} \\
\mathsf{merge}(\mathsf{F}^{\mu},\ \mathsf{F}^{\nu}) &= \{\mathsf{F}^{\mu \sqcup \nu}\} \\
\mathsf{merge}(a,\ b) &= \emptyset,\ \text{otherwise}
\end{aligned}$$

(1) if $e \in E_1 \setminus E_2$ then $\lambda(e) = \lambda_1(e)$,
(2) if $e \in E_2 \setminus E_1$ then $\lambda(e) = \lambda_2(e)$,
(3) if $e \in E_1 \cap E_2$ then $\lambda(e) \in \mathsf{merge}(\lambda_1(e),\ \lambda_2(e))$, the first has no rf,

5.6 Read-Modify-Write Operations (RMW)

Extend the syntax

$$S \ ::= \ \cdots \ \mid \ r := \mathsf{CAS}^{\mu_1,\mu_2}([L], M, N) \ \mid \ r := \mathsf{FADD}^{\mu_1,\mu_2}([L], M) \ \mid \ r := \mathsf{EXCHG}^{\mu_1,\mu_2}([L], M)$$

From the data model, we require an additional binary relation over $\mathcal{A} \times \mathcal{A}$: *overlaps*. For the actions in this paper, we say *a overlaps b* if they access the same location.

We give the semantics without if-closure or address calculation.

Definition 5.14. Let *READ′* be defined as for *READ*, adding the constraint:

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

If $P \in \mathit{FADD}(r, x, M, \mu_1, \mu_2)$ then $(\exists P_1 \in \mathit{READ}'(r,\ x,\ \mu_1)\,;\ \mathit{WRITE}(x,\ r{+}M,\ \mu_2))$
(U1) if $\lambda_1(e)$ is a write then there is a read $\lambda_1(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.
If $P \in \mathit{EXCHG}(r, x, M, \mu_1, \mu_2)$ then $(\exists P_1 \in \mathit{READ}'(r,\ x,\ \mu_1)\,;\ \mathit{WRITE}(x,\ M,\ \mu_2))$
(U1) if $\lambda_1(e)$ is a write then there is a read $\lambda_1(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.
If $P \in \mathit{CAS}(r, x, M, N, \mu_1, \mu_2)$ then $(\exists P_1 \in \mathit{READ}'(r,\ x,\ \mu_1)\,;\ \mathit{IF}(r{=}M,\ \mathit{WRITE}(x,\ N,\ \mu_2),\ \mathit{SKIP}))$
(U1) if $\lambda_1(e)$ is a write then there is a read $\lambda_1(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.

RMW operations are formalized by adding a relation $\xrightarrow{\mathsf{rmw}} \subseteq E \times E$ that relates the read of a successful RMW to the succeeding write. Extend the definition of a pomset as follows.

(M8) $\mathsf{rmw} : E \to E$ is a partial function capturing read-modify-write *atomicity*, such that
(M8a) if $d \xrightarrow{\mathsf{rmw}} e$ then $\lambda(e)$ blocks $\lambda(d)$,
(M8b) if $d \xrightarrow{\mathsf{rmw}} e$ then $d \le e$,
(M8c) if $\lambda(c)$ overlaps $\lambda(d)$ then
(i) if $d \xrightarrow{\mathsf{rmw}} e$ then $c \le e$ implies $c \le d$,
(ii) if $d \xrightarrow{\mathsf{rmw}} e$ then $d \le c$ implies $e \le c$.

Extend the definition of par, if, seq to include:

(P0) (S0) (C0) $\mathsf{rmw} = (\mathsf{rmw}_1 \cup \mathsf{rmw}_2)$,

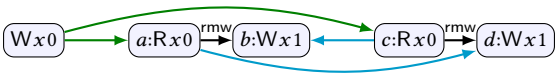
Example 5.15. This definition ensures atomicity, disallowing executions such as [Podkopaev et al. 2019, Ex. 3.2]:

$$x := 0\,;\ s := \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x, 1) \ \|\ x := 2\,;\ s := x$$

By M8c(i), since $(\mathsf{W}x2) \to (\mathsf{W}x1)$, it must be that $(\mathsf{W}x2) \to (\mathsf{R}x0)$, creating a cycle.

Example 5.16. Two successful RMWs cannot see the same write:

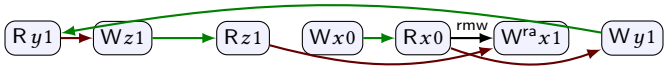
$$x := 0;\ (\mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x,1) \parallel \mathsf{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x,1))$$



The order from read-to-write is required by fulfillment. Apply ?? to $a \to d$, we have that $a \to c$. Subsequently applying ??, we have $b \to c$, creating a cycle.

Example 5.17. By using two actions rather than one, the definition allows examples such as the following, which is allowed by Arm8 [Podkopaev et al. 2019, Ex. 3.10]:

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



Example 5.18. For RMW operations, the independent case for a read should be the same as the empty case. To see why, consider the semantics of local invariant reasoning (LIR) from Def ??:

?? $\tau^D(\psi) \vDash \psi[M/x] \wedge M{=}v$,
?? $\tau^C(\psi) \vDash \psi[M/x]$,
?? $\tau^D(\psi) \vDash v{=}r \Rightarrow \psi$,
?? $\tau^C(\psi) \vDash (v{=}r \vee x{=}r) \Rightarrow \psi$, when $E \neq \emptyset$,
?? $\tau^B(\psi) \vDash \psi$, when $E = \emptyset$.

Consider the relaxed variant of the CDRF example from [Lee et al. 2020], using a semantics for FADD that simply composes the rules for load and store above.

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

Looking at the independent transformers of the second thread and initializer, we have:

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

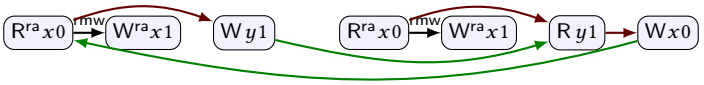
After sequencing, the precondition of $(\mathsf{W}y1)$ is a tautology: $(0{=}r \vee 0{=}r) \Rightarrow r{=}0$.

Here, local invariant reasoning is using the initializing write to x to justify the independence of the write to y. But this write is made unavailable by the first thread's successful RMW.

As a result, we disallow the use of ?? when treating the read event in an RMW.
[Todo: write out the rules.]

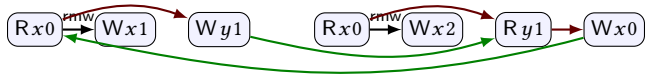
Example 5.19. Consider the CDRF example from [Lee et al. 2020]:

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



Example 5.20. Consider this example from [Lee et al. 2020, §C]:

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



For case analysis of RMWs, we can use a general purpose expansion operator:
If $P \in \mathit{EXPAND}(\mathcal{P})$ then $(\exists P_1, \ldots, P_n \in \mathcal{P})\ (\exists \theta_1, \ldots, \theta_n \in \Phi)$

(E0a) if $\theta_i \wedge \theta_j$ is satisfiable then $i = j$,
(E0b) $\bigvee_i \theta_i \vDash \mathtt{tt}$,
(E1a) if $E_i \cap E_j \neq \emptyset$ then $i = j$,
(E1b) $E = \bigcup_i E_i$,
(E2) $\lambda = \bigcup_i \lambda_i$,
(E3) $\kappa(e) \vDash \theta_e \wedge \kappa_e(e)$,
(E4) $\tau^D(\psi) \vDash \bigvee_i(\theta_i \wedge \tau_i^D(\psi))$,
(E5) $\checkmark \vDash \bigvee_i(\theta_i \wedge \checkmark_i)$,
(E6) $\mathsf{rf} = \bigcup_i \mathsf{rf}_i$,
(E7) $\leq = \bigcup_i \leq_i$.

REFERENCES

Jade Alglave. 2020. This commit adds three alternative formulations of the Arm model, both for non-mixed and mixed size accesses. https://github.com/herd/herdtools7/commit/685ee4.

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

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

Arm Limited. 2020. Arm Architecture Reference Manual: Armv8, for Armv8-A Architecture Profile (Issue F.c). https://developer.arm.com/documentation/ddi0487/latest.

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

Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 100–110. http://dl.acm.org/citation.cfm?id=3049844

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *PACMPL* 3, POPL (2019), 70:1–70:28. https://doi.org/10.1145/3290383

Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. 2021. Modular Data-Race-Freedom Guarantees in the Promising Semantics. *Proc. ACM Program. Lang.* 3, OOPSLA (2021). To Appear.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Anton Podkopaev. 2020. Private correspondence.

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

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

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

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

A DISCUSSION

A.1 Downset Closure

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

Definition A.1. P_2 is an *downset* of P_1 if

(1) $E_2 \subseteq E_1$,
(2) $(\forall e \in E_2)\ \lambda_2(e) = \lambda_1(e)$,
(3) $(\forall e \in E_2)\ \kappa_2(e) = \kappa_1(e)$,
(4) $(\forall e \in E_2)\ \tau_2^D(e) = \tau_1^D(e)$,
(5) $(\forall d \in E_2)\ (\forall e \in E_2)\ d \leq_2 e$ if and only if $d \leq_1 e$,
(6) $(\forall d \in E_1)\ (\forall e \in E_2)$ if $d \leq_1 e$ then $d \in E_2$.

Downset closure fails due to RRD and LIR. The key property is that the empty set transformer should behave the same as the independent transformer.

Example A.2. For RRD, Def ?? states:

?? $\tau^D(\psi) \vDash v{=}r \Rightarrow \psi$,
?? $\tau^C(\psi) \vDash (v{=}r \vee \mathsf{W}) \Rightarrow \psi$,
?? $\tau^B(\psi) \vDash \psi$, when $E = \emptyset$.

This semantics is not downset closed due to the lack of read-read dependencies. In both cases, for subsequent writes, ?? is the same as ??. For subsequent reads, ?? is the same as ??. Consider

$$r := x\,;\ \texttt{if}(!r)\{s := y\}$$
$$\boxed{\mathsf{R}\,x\,0} \qquad \boxed{\mathsf{R}\,y\,0}$$

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

$$r := x \qquad\qquad \texttt{if}(!r)\{s := y\}$$
$$\boxed{\mathsf{R}\,x\,0} \qquad\qquad \emptyset$$

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

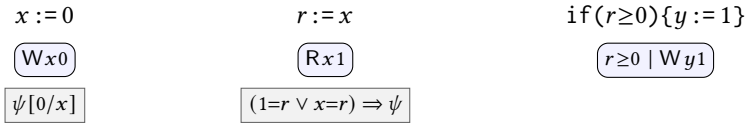
$$r := x \qquad\qquad \texttt{if}(!r)\{s := y\}$$
$$\emptyset \qquad\qquad \boxed{r{=}0 \mid \mathsf{R}\,y\,0}$$

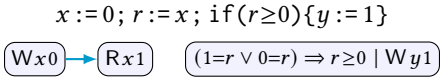
Since r appears only once in the program, this pomset cannot contribute to a top-level pomset.

Example A.3. For LIR, Def ?? states:

?? $\tau^D(\psi) \vDash v{=}r \Rightarrow \psi$,
?? $\tau^C(\psi) \vDash (v{=}r \vee x{=}r) \Rightarrow \psi$, when $E \neq \emptyset$,
?? $\tau^B(\psi) \vDash \psi$, when $E = \emptyset$.

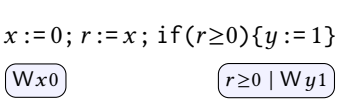
This semantics is not downset closed: The independency reasoning of **??** is only applicable for pomsets where the ignored read is present! Revisiting Ex **??**





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

Taking the empty set for the read, however, we have:



The precondition of $(\mathsf{W}y1)$ is not a tautology.

A.2 Comparison with Weakest Preconditions

We compare traditional transformers to the dependent-case transformers of Def **??**; thus we consider only totally ordered executions. Because we only consider the dependent case, we drop the superscript E on τ^E throughout this section. We also assume that each register appears at most once in a program, as we did throughout §3–4.

Because of augment closure, we are not interested in isolating the *weakest* precondition. Thus we think of transformers as Hoare triples. In addition, all programs in our language are strongly normalizing, so we need not distinguish strong and weak correctness. In this setting, the Hoare triple $\{\phi\}\ S\ \{\psi\}$ holds exactly when $\phi \Rightarrow wp_S(\psi)$.

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

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

Here we have chosen an alternative formulation for the read rule, which is equivalent the more traditional $\psi[x/r]$, as long as registers are assigned at most once in a program. In Def **??**, the transformers for the dependent case are as follows:

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

Only the read rule differs from the traditional one.

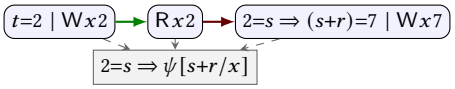
For programs where every register is bound and every read is fulfilled, our dependent transformers are the same as the traditional ones. In our semantics, thus, we only consider totally-ordered executions where every read could be fulfilled by prepending some writes. For example, we ignore pomsets of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

For example, let S_i be defined:

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

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

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



The predicate transformers are:

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

A.3 Substitutions

Recall the load rules from §??:

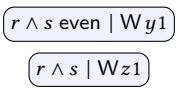
?? $\tau^D(\psi) \vDash v{=}r \Rightarrow \psi$,
?? $\tau^C(\psi) \vDash (v{=}r \vee x{=}r) \Rightarrow \psi$, when $E \neq \emptyset$,
?? $\tau^B(\psi) \vDash \psi$, when $E = \emptyset$.

It is also possible to collapse x and r when doing a load:

?? $\tau^D(\psi) \vDash v{=}r \Rightarrow \psi[r/x]$,
?? $\tau^C(\psi) \vDash (v{=}r \vee x{=}r) \Rightarrow \psi[r/x]$, when $E \neq \emptyset$.
?? $\tau^B(\psi) \vDash \psi[r/x]$, when $E = \emptyset$.

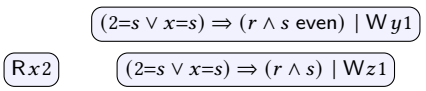
Perhaps surprisingly, these two semantics are incomparable. Consider the following:

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



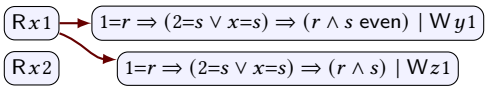
Prepending $(s := x)$, we get the same result regardless of whether we substitute $[s/x]$, since x does not occur in either precondition. Here we show the independent case:

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



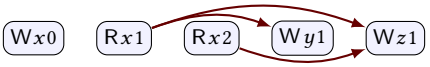
Prepending $(r := x)$, we now get different results since the preconditions mention x. Without substitution:

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



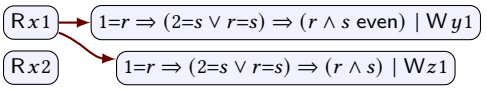
Prepending $(x := 0)$, which substitutes $[0/x]$, the precondition of $(\mathsf{W}y1)$ becomes $(1{=}r \Rightarrow (2{=}s \vee 0{=}s) \Rightarrow (r \wedge s\ \text{even}))$, which is a tautology, whereas the precondition of $\mathsf{W}z1$ becomes $(1{=}r \Rightarrow$

$(2{=}s \vee 0{=}s) \Rightarrow (r \wedge s))$, which is not. In order to be top-level, $\mathsf{W}z1$ must depend on $\mathsf{R}x2$; in this case the precondition becomes $(1{=}r \Rightarrow 2{=}s \Rightarrow (r \wedge s))$, which is a tautology.

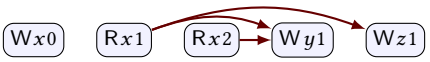


The situation reverses with the substitution $[r/x]$:

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



Prepending $(x := 0)$:



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

Thinking in terms of hardware, the difference is whether reads update the cache, thus clobbering preceding writes. With $[r/x]$, reads clobber the cache, whereas without the substitution, they do not. Since most caches work this way, the model with $[r/x]$ is likely preferred for modeling hardware. However, this substitution only makes sense in a model with read-read coherence and dependency. By leaving out the substitution, we also ensure that downgraded reads are fulfilled by preceding writes, not reads.

B DIFFERENCES WITH OOPSLA

B.1 Substitution

[Jagadeesan et al. 2020] uses substitution rather than Skolemizing. Indeed our use of Skolemization is motivated by disjunction closure for predicate transformers, which do not appear in [Jagadeesan et al. 2020]. In §?? on local invariant reasoning (LIR), we gave the semantics of load for nonempty pomsets as:

?? $\tau^D(\psi) \vDash v{=}r \Rightarrow \psi$,
?? $\tau^C(\psi) \vDash (v{=}r \vee x{=}r) \Rightarrow \psi$.

In [Jagadeesan et al. 2020], the definition is roughly as follows:

?? $\tau^D(\psi) \vDash \psi[v/r][v/x]$,
?? $\tau^C(\psi) \vDash \psi[v/r][v/x] \wedge \psi[x/r]$.

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

The substitutions collapse x and r, allowing local invariant reasoning, as in §??. Without Skolemizing it is necessary to substitute $[x/r]$, since the reverse substitution $[r/x]$ is useless when r is bound—compare with §A.3. As discussed below (§B.2), including this substitution affects the interaction of LIR and downset closure.

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

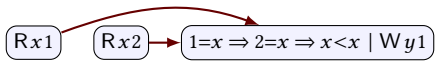
[4] $(\psi_1 \vee \psi_2)' = (\psi_1' \vee \psi_2')$ and $(\psi_1 \vee \psi_2)'' = (\psi_1'' \vee \psi_2'')$.

The substitution $[x/r]$ does not work with Skolemization, even for the dependent case, since we lose the unique marker for each read. In effect, this forces the reads to the same values. To be concrete, the candidate definition would modify **??** to be:

??) $\tau^D(\psi) \vDash v{=}x \Rightarrow \psi[x/r]$.

Using this definition, consider the following:

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



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

B.2 Downset closure

[Jagadeesan et al. 2020] enforces downset closure in the prefixing rule. Even without this, downset closure would be different for the two semantics, due to the use of substitution in [Jagadeesan et al. 2020]. Consider the final pomset of Ex A.3, under the semantics of this paper, which elides the middle read event:

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

$$\boxed{W\,x0} \qquad \boxed{r{\geq}0 \mid W\,y1}$$

In [Jagadeesan et al. 2020], the substitution $[x/r]$ is performed by the middle read regardless of whether it is included in the pomset, with the subsequent substitution of $[0/x]$ by the preceding write, we have $[x/r][0/x]$, which is $[0/r][0/x]$, resulting in:

$$\boxed{W\,x0} \qquad \boxed{0{\geq}0 \mid W\,y1}$$

B.3 Consistency

[Jagadeesan et al. 2020] imposes *consistency*, which requires that for every pomset P, $\bigwedge_e \kappa(e)$ is satisfiable. Associativity requires that we allow pomsets with inconsistent preconditions. Consider a variant of Ex 5.6 from §**??**.

$$\underset{\boxed{M \mid W\,x1}}{\texttt{if}(M)\{x := 1\}} \qquad \underset{\boxed{\neg M \mid W\,x1}}{\texttt{if}(!M)\{x := 1\}} \qquad \underset{\boxed{M \mid W\,y1}}{\texttt{if}(M)\{y := 1\}} \qquad \underset{\boxed{\neg M \mid W\,y1}}{\texttt{if}(!M)\{y := 1\}}$$

Associating left and right, we have:

$$\underset{\boxed{W\,x1}}{\texttt{if}(M)\{x := 1\};\ \texttt{if}(!M)\{x := 1\}} \qquad \underset{\boxed{W\,y1}}{\texttt{if}(M)\{y := 1\};\ \texttt{if}(!M)\{y := 1\}}$$

Associating into the middle, instead, we require:

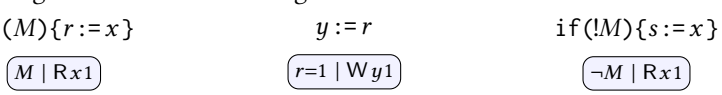
$$\underset{\boxed{M \mid W\,x1}}{\texttt{if}(M)\{x := 1\}} \qquad \underset{\boxed{\neg M \mid W\,x1}\ \boxed{M \mid W\,y1}}{\texttt{if}(!M)\{x := 1\};\ \texttt{if}(M)\{y := 1\}} \qquad \underset{\boxed{\neg M \mid W\,y1}}{\texttt{if}(!M)\{y := 1\}}$$

Joining left and right, we have:

$$\underset{\boxed{W\,x1}\ \boxed{W\,y1}}{\texttt{if}(M)\{x := 1\};\ \texttt{if}(!M)\{x := 1\};\ \texttt{if}(M)\{y := 1\};\ \texttt{if}(!M)\{y := 1\}}$$

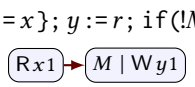
B.4 Causal Strengthening

Causal Strengthening [Jagadeesan et al. 2020] imposes *causal strengthening*, which requires for every pomset P, if $d \leq e$ then $\kappa(e) \vdash \kappa(d)$. Associativity requires that we allow pomsets without causal strengthening. Consider the following.



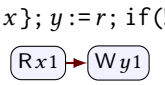
Associating left, with causal strengthening:

Finally, merging:



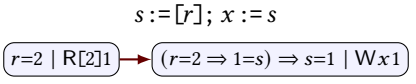
Instead, associating right:

Merging:

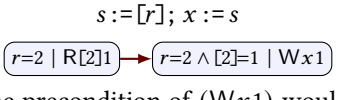


With causal strengthening, the precondition of $\mathsf{W}y1$ depends upon how we associate. This is not an issue in [Jagadeesan et al. 2020], which always associates to the right.

One use of causal strengthening is to ensure that address dependencies do not introduce thin air reads. Associating to the right, the intermediate state of Ex 5.11 is:



In [Jagadeesan et al. 2020], we have, instead:



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

B.5 Parallel Composition

In [Jagadeesan et al. 2020, §2.4], parallel composition is defined allowing coalescing of events. Here we have forbidden coalescing. This difference appears to be arbitrary. In [Jagadeesan et al. 2020], however, there is a mistake in the handling of termination actions. The predicates should be joined using $\wedge$, not $\vee$.

B.6 Read-Modify-Write Actions

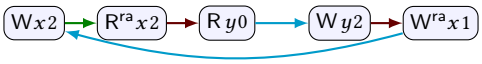
In [Jagadeesan et al. 2020], the atomicity axioms M8c erroneously applies only to overlapping writes, not overlapping reads. The difficulty can be seen in Ex 5.16.

[Jagadeesan et al. 2020] does not specify the calculation of dependency for RMWs, as discussed in Ex 5.18.

B.7 Downgrading Internal Acquiring Reads

Shortly after publication, Podkopaev [2020] noticed a shortcoming of the implementation on Arm8 in [Jagadeesan et al. 2020, §7]. The proof given there assumes that all internal reads can be dropped. However, this is not the case for acquiring reds. For example, [Jagadeesan et al. 2020] disallows the following execution, which is allowed by Arm8 and TSO.

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



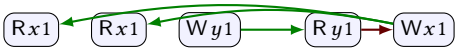
The solution we have adopted is to allow an acquiring read to be downgraded to a relaxed read when it is preceded (sequentially) by a relaxed write that could fulfill it. This solution allows executions that are not allowed under Arm8 since we do not insist that the local relaxed write is actually read from. This may seem counterintuitive, but we don't see a local way to be more precise.

As a result, we use a different proof strategy for Arm8 implementation, which does not rely on read elimination. The proof idea uses a recent alternative characterization of Arm8 [Alglave 2020; Arm Limited 2020].

B.8 Redundant Read Elimination

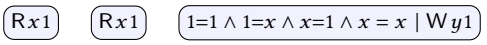
Contrary to the claim, redundant read elimination fails for [Jagadeesan et al. 2020]. We discussed redundant read elimination in §??. Consider JMM Causality Test Case 2, which we discussed there.

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



Under the semantics of [Jagadeesan et al. 2020], we have

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



The precondition of $(\mathsf{W}y1)$ is *not* a tautology, and therefore redundant read elimination fails. (It is a tautology in $r := x;\ s := r;\ \texttt{if}(r{=}s)\{y := 1\}$.) In [Jagadeesan et al. 2020, §3.1], we incorrectly stated that the precondition of $(\mathsf{W}y1)$ was $1{=}1 \wedge x{=}x$.

B.9 Stupid Bug in LDRF

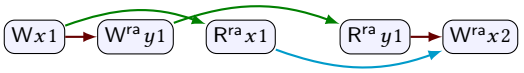
Definition of race is wrong. Should say that at least one is relaxed.

C MORE STUFF

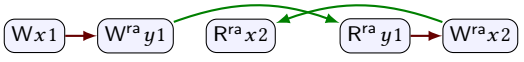
C.1 A Note on Mixed-Mode Data Races

In preparing this paper, we came across the following example, which appears to invalidate Theorem 4.1 of [Dongol et al. 2019].

$$x := 1;\ y^{\mathsf{ra}} := 1;\ r := x^{\mathsf{ra}} \ \|\ \texttt{if}(y^{\mathsf{ra}})\{x^{\mathsf{ra}} := 2\}$$



(*)



(†)

The program is data-race free. The two executions shown are the only top-level executions that include $(\mathsf{W}^{\mathsf{ra}}x2)$.

Theorem 4.1 of [Dongol et al. 2019] is stated by extending execution sequences. In the terminology of [Dongol et al. 2019], a read is *L-weak* if it is sequentially stale. Let $\rho = (\mathsf{W}x1)(\mathsf{W}^{\mathsf{ra}}y1)(\mathsf{R}^{\mathsf{ra}}y1)(\mathsf{W}^{\mathsf{ra}}x2)$ be a sequence and $\alpha = (\mathsf{R}^{\mathsf{ra}}x1)$. ρ is L-sequential and α is L-weak in $\rho\alpha$. But there is no execution of this program that includes a data race, contradicting the theorem. The error seems to be in Lemma A.4 of [Dongol et al. 2019], which states that if α is L-weak after an L-sequential ρ, then α must be in a data race. That is clearly false here, since $(\mathsf{R}^{\mathsf{ra}}x1)$ is stale, but the program is data race free.

In proving the SC-LDRF result in [Jagadeesan et al. 2020, §8], we noted that our proof technique is more robust than that of [Dongol et al. 2019], because it limits the prefixes that must be considered. In (∗), the induction hypothesis requires that we add $(\mathsf{R}^{\mathsf{ra}}x1)$ before $(\mathsf{W}^{\mathsf{ra}}x2)$ since $(\mathsf{R}^{\mathsf{ra}}x1) \rightarrow (\mathsf{W}^{\mathsf{ra}}x2)$. In particular,

is not a downset of (∗), because $(\mathsf{R}^{\mathsf{ra}}x1) \rightarrow (\mathsf{W}^{\mathsf{ra}}x2)$. As we noted in [Jagadeesan et al. 2020, §8], this affects the inductive order in which we move across pomsets, but does not affect the set of pomsets that are considered. In particular,

is a downset of (∗).

C.2 If Closure and Address Dependencies

An optimization (p/q are registers):

$$r := [p];\ s := [q]$$

vs

$$r := [p];\ \texttt{if}(p{=}q)\{s := r\}\ \texttt{else}\ \{s := [q]\}$$

$$r := \texttt{new};\ [r] := 42;\ s := [r];\ x := r \parallel r := x;\ [r] := 7$$

If closure is at odds with Java Final field semantics.

Do sequencing and if commute?

C.3 About Arm

Hypothesis: gcb cannot contradict (poloc minus RxR).