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

ANONYMOUS AUTHOR(S)

Program logics and semantics tell us that when executing $(S_1; S_2)$ starting in state s_0, we execute S_1 in s_0 to arrive at s_1, then execute S_2 in s_1 to arrive at the final state s_2. This is, of course, an abstraction. Processors execute instructions out of order, due to pipelines and caches, and compilers reorder programs even more dramatically. All of this reordering is meant to be unobservable in single-threaded code, but is observable in multi-threaded code. A formal attempt to understand the resulting mess is known as a "relaxed memory model." The relaxed memory models that have been proposed to date either fail to address sequential composition directly, or overly restrict processors and compilers.

To support sequential composition while targeting modern hardware, we propose adding families of predicate transformers to the existing model of "Pomsets with Preconditions," which already supports parallel composition. When composing $(S_1; S_2)$, the predicate transformers used to validate the preconditions of events in S_2 are chosen based on the semantic dependencies from events in S_1 to events in S_2. 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), 29 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 1960s, we've been able to explain this using logic [Hoare 1969]. In Dijkstra's [1975] formulation, we think of programs as *predicate transformers*, where predicates describe the state of memory in the system. In the calculus of weakest preconditions, programs map postconditions to preconditions. We recall the definition of $wp_S(\psi)$ for loop-free code below (where r–s range over thread-local *registers* and M–N range over side-effect free *expressions*).

(D1) $wp_{\mathtt{skip}}(\psi) = \psi$
(D2) $wp_{r := M}(\psi) = \psi[M/r]$

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

(D3) $wp_{S_1;S_2}(\psi) = wp_{S_1}(wp_{S_2}(\psi))$

(D4) $wp_{\mathtt{if}(M)\{S_1\}\mathtt{else}\{S_2\}}(\psi) = ((M{\neq}0) \Rightarrow wp_{S_1}(\psi)) \wedge ((M{=}0) \Rightarrow wp_{S_2}(\psi))$

For this language, the Hoare triple $\{\phi\}\ S\ \{\psi\}$ holds exactly when $\phi \Rightarrow wp_S(\psi)$. This is an elegant explanation of sequential computation in a sequential context. Note that D2 is sound because a read from a thread-local register must be fulfilled by a preceding write in the same thread. In a concurrent context, with shared variables (x–z), the obvious generalizations

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

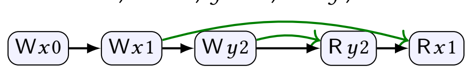
are unsound! In particular, a read from a shared memory location may be fulfilled by a write in another thread, invalidating D2c. (We assume that expressions do *not* include shared variables.)

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 overconstraining the implementation. To understand the solution, one must first understand the constraints imposed by hardware and compilers.

1.2 Memory Models

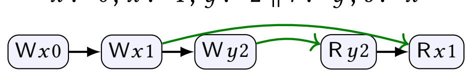
For single-threaded programs, memory can be thought of as you might expect: programs write to, and read from, memory references. This can be thought of as a total order of reads and writes (black arrows), where each read has a matching *fulfilling* write (green arrows), for example:

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



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

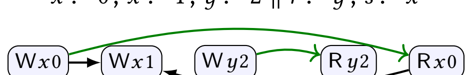
$$x := 0;\ x := 1;\ y := 2 \parallel r := y;\ s := x$$



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.

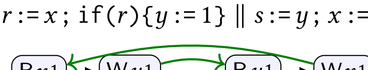
This has led to *relaxed memory models*, in which the requirement of sequential consistency is weakened to only apply *per-location* and not globally over the whole program. This allows executions which are inconsistent with program order, such as:

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



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:

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

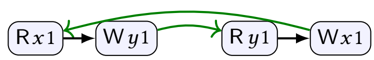


This candidate execution forms a cycle in causal order, so is disallowed, but this depends crucially on the control dependency from $(\mathsf{R}x1)$ to $(\mathsf{W}y1)$, and the data dependency from $(\mathsf{R}y1)$ to $(\mathsf{W}x1)$. If either is missing, then this execution is acyclic and hence allowed.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

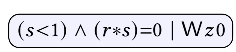
$$r := x;\ \mathtt{if}(r)\{y := 1\}\,\mathtt{else}\,\{y := 1\} \parallel s := y;\ x := s$$



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

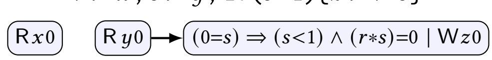
To address this, Jagadeesan et al. [2020] introduced *Pomsets with Preconditions*, where events are labeled with logical formulae. Nontrivial preconditions are introduced by store actions (modeling data dependencies) and conditionals (modeling control dependencies):

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

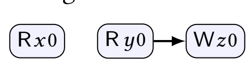


Preconditions are discharged by being ordered after a read:

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



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,

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

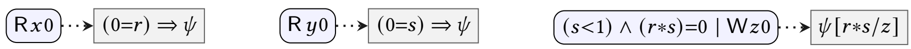
where S is the entire future computation! By formalizing sequential composition, we can show:

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

Then the equivalence holds in any context.

Predicate transformers are a good fit for logical models of dependency calculation, since both are concerned with preconditions and how they are transformed by sequential composition. Our first attempt is to associate a predicate transformer with each pomset. We visualize this in diagrams by showing how ψ is transformed, for example:

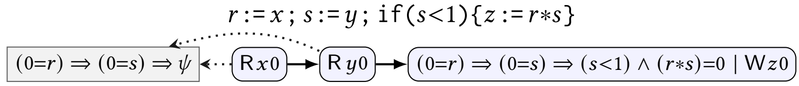
$r\,{:=}\,x$ $\quad$ $s\,{:=}\,y$ $\quad$ $\mathtt{if}(s{<}1)\{z\,{:=}\,r{*}s\}$



The predicate transformer from the write matches Dijkstra's D2b. For the reads, however, D2c defines the transformer of $r\,{:=}\,x$ to be $\psi[x/r]$, which is equivalent to $(x{=}r) \Rightarrow \psi$ under the assumption that registers are assigned at most once. Instead, we use $(0{=}r) \Rightarrow \psi$, reflecting the fact that 0 may come from a concurrent write. The obligation to find a matching write is moved from the sequential semantics of *substitution* and *implication* to the concurrent semantics of *fulfillment*.

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

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



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.

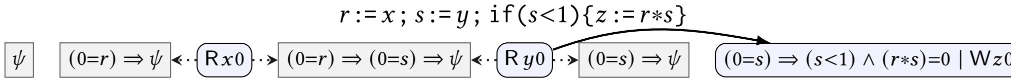
For example, for single-event pomsets, there are two predicate transformers, since there are two subsets of any one-element set. The *independent* transformer is indexed by the empty set, whereas the *dependent* transformer is indexed by the singleton. We visualize this by including more than one transformed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

$r\,{:=}\,x$ $\quad$ $s\,{:=}\,y$

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

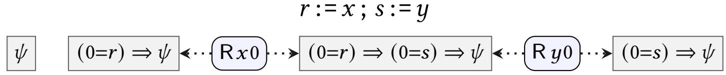
For example, we can recover the expected semantics for (†) by choosing the predicate transformer which is independent of $(\mathsf{R}\,x\,0)$ but dependent on $(\mathsf{R}\,y\,0)$, which is the transformer which maps ψ to $(0{=}s) \Rightarrow \psi$.

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



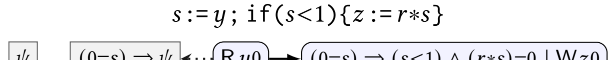
As a sanity check, we can see that sequential composition is associative in this case, since it does not matter whether we associate to the left, with intermediate step:

$r\,{:=}\,x\,;\; s\,{:=}\,y$



or to the right, with intermediate step:

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



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

1.4 Related Work

Marino et al. [2015] argue that the "silently shifting semicolon" is sufficiently problematic for programmers that concurrent languages should guarantee sequential abstraction, despite the performance penalties. In this paper, we 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 few 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. Kavanagh and Brookes [2018] define a semantics using pomsets without preconditions. Instead, their model uses syntactic dependencies, thus invalidating many compiler optimizations. They also require a fence after every relaxed read on Arm8. Pichon-Pharabod and Sewell [2016] use event structures to calculate dependencies, combined with an operational semantics that incorporates program transformations. This approach seems to require whole-program analysis.

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

Jagadeesan et al. [2020] note that the speculative models listed above all, including [Kang et al. 2017], fail to validate compositional reasoning of temporal properties—see their examples OOTA4, OOTA5, and [Lochbihler 2013, Fig. 8]). 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.

We provide a detailed comparison with [Jagadeesan et al. 2020] in §B.

1.5 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.

- §2 presents the basic model, which satisfies many desiderata, but not all.
- §3 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 a nontrivial change to the definition of sequential composition.
- §4 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 THE BASIC MODEL

After some preliminaries (§2.1–2.2), we define the basic model and establish some basic properties (§2.3 and Figure 1). We then explain the model using examples (§2.4–2.11). We encourage readers to skim the definitions and then skip to §2.4, coming back as needed.

2.1 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.

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

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

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], $]\!]$ denotes parallel composition, preserving thread state on the left after a join. In examples and sublanguages without join, we use the symmetric $\parallel$ operator.

Throughout §1–3 we require that

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

In §4, 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 $\phi \vDash \mathsf{ff}$.

2.2 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 2.1. 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 $a = (\mathsf{W}xv)$ and $b = (\mathsf{R}xv)$.
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: $\bowtie_{\mathsf{sc}} = \{(\mathsf{W}^{\mathsf{sc}}, \mathsf{W}^{\mathsf{sc}}),\ (\mathsf{R}^{\mathsf{sc}}, \mathsf{W}^{\mathsf{sc}}),\ (\mathsf{W}^{\mathsf{sc}}, \mathsf{R}^{\mathsf{sc}}),\ (\mathsf{R}^{\mathsf{sc}}, \mathsf{R}^{\mathsf{sc}})\}$.

2.3 The Semantic Domain

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

Definition 2.2. 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 $\tau(\phi) \vDash \tau(\psi)$.

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

As discussed in §1, predicate transformers suffice for sequentially consistent models, but not relaxed models, where dependency calculation is crucial. For dependency calculation, we use a *family* of predicate transformers, indexed by sets of events. In sequential composition, we will use $\tau^{\downarrow e}$ as the predicate transformer applied to event e where $d \in (\downarrow e)$ if $d < e$.

Definition 2.3. 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 2.4. A *pomset with predicate transformers* over $\mathcal{A}$ is a tuple $(E, \lambda, \kappa, \tau, \checkmark, \mathsf{rf}, \le)$ 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) $\le : E \times E$, is a partial order capturing *causality*, such that
  (M7a) if $d \xrightarrow{\mathsf{rf}} e$ and $\lambda(c)$ blocks $\lambda(e)$ then either $c \le d$ or $e \le c$.

A pomset is *top-level* if (T1) $\checkmark$ is a tautology and (T2) for every $e \in E$,
(T2a) $\kappa(e)$ is a tautology,
(T2b) if $\lambda(e)$ is a read then there is some $d \xrightarrow{\mathsf{rf}} e$.

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

- (P1) $E = (E_1 \uplus E_2)$,
- (P2) $\lambda = (\lambda_1 \cup \lambda_2)$,
- (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) $\le$ extends $\le_1$ and $\le_2$,
- (P7b) if $d \in E_1$, $e \in E_2$ and $d \xrightarrow{\mathsf{rf}} e$ then $d \le e$.

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

- (S1) $E = (E_1 \cup E_2)$,
- (S2) $\lambda = (\lambda_1 \cup \lambda_2)$,
- (S3a) if $e \in E_1 \setminus E_2$ then $\kappa(e) \vDash \kappa_1(e)$,
- (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) $\le$ extends $\le_1$ and $\le_2$,
- (S7b) if $d \in E_1$, $e \in E_2$ and $d \xrightarrow{\mathsf{rf}} e$ then $d \le e$,
- (S7c) if $\lambda_1(d)$ delays $\lambda_2(e)$ then $d \le e$.

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) $\le$ extends $\le_1$ and $\le_2$,
- (C7b) $\le \subseteq (\le_1 \cup \le_2)$.

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 $\tau^D(\psi) \vDash \psi$,
- (R5) if $E = \emptyset$ and $\mu \neq \mathsf{rlx}$ then $\checkmark \vDash \mathsf{ff}$.

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 $\checkmark \vDash \mathsf{ff}$.

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

Fig. 1. Semantics of programs

We give the semantics of programs $[\![\cdot]\!]_1$ in Figure 1.

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

The model has seven components, which can be daunting at first glance. To aid the reader, we use consistent numbering throughout. For example, item 7 always refers to the order relation.

The core of the model is a pomset, which includes a set of events (M1), a labeling (M2), and an order (M7). We also include the *reads-from* relation explicitly in the model (M6).

On top of this basic structure, M3–M5 add a layer of logic. For each pomset, M5 provides a termination condition. For each event in a pomset, M3 provides a precondition. For each set of events in a pomset, M4 provides a predicate transformer. Sequential dependency is calculated by κ'_2 in the semantics of sequential composition.

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

Lemma 2.5. *$(\mathcal{P}_1\,;\,\mathcal{P}_2)\,;\,\mathcal{P}_3 = \mathcal{P}_1\,;\,(\mathcal{P}_2\,;\,\mathcal{P}_3)$ and $\mathcal{P}\,;\,\texttt{skip} = \mathcal{P} = \texttt{skip}\,;\,\mathcal{P}$. $(\mathcal{P}_1 \parallel \mathcal{P}_2) \parallel \mathcal{P}_3 = \mathcal{P}_1 \parallel (\mathcal{P}_2 \parallel \mathcal{P}_3)$ and $\mathcal{P} \parallel \texttt{skip} = \mathcal{P}$.*

Proof. Straightforward calculation. Associativity of ; requires disjunction closure (X3). □

Definition 2.6. 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) $\le_2 \supseteq \le_1$.
(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 2.7. *If $P_1 \in [\![S]\!]_1$ and P_2 augments P_1 then $P_2 \in [\![S]\!]_1$.*

Proof. Induction on the definition of $[\![\cdot]\!]_1$. □

2.4 Pomsets

We first explain the core of model, ignoring the logic (rules 3–5). We defer discussion of *IF* to §2.7.

Reads, writes, and fences map to pomsets with at most one event. `skip` maps to the empty pomset. Ignoring the logic, the definitions are straightforward. Note only that $[\![x := 1]\!]_1$ can write any value v; the fact that v must be 1 is captured in the logic (see §2.5).

The structural rules combine pomsets: Parallel composition is disjoint union, inheriting labeling, order and rf from the two sides. Any rf edges added between the two sides must also be added to the order (P7b). Sequential composition is similar, with two changes: S1 does not require disjointness (see §2.5), and S7c may require order (see example PUB, below).

Note that reads-from implies order.

Lemma 2.8. *For any P in the range of $[\![\cdot]\!]_1$, $d \xrightarrow{\mathsf{rf}} e$ implies $d \le e$.*

Proof. Induction on the definition of $[\![\cdot]\!]_1$, using P7b and S7b. □

In top-level pomsets, every read must have a matching write in rf (T2b). Together with M6a and M7a, the lemma guarantees that reads are *fulfilled* at top-level, as in [Jagadeesan et al. 2020, §2.7].[1]

From Definition 2.1, recall that a *delays* b if $a \bowtie_{\mathsf{co}} b$ or $a \ltimes_{\mathsf{sync}} b$ or $a \bowtie_{\mathsf{sc}} b$. S7c guarantees that sequential order is enforced between conflicting accesses of the same location ($\bowtie_{\mathsf{co}}$), into a release and out of an acquire ($\ltimes_{\mathsf{sync}}$), and between SC accesses ($\bowtie_{\mathsf{sc}}$). Combined with the fulfillment

[1] The basic model would be the same if we move rf from the model itself to be existentially quantified in the definition of top-level pomset, along with M6a and M7a. This was the approach of Jagadeesan et al. We include rf explicitly for use in §3.3, where we introduce a variant semantics $[\![\cdot]\!]_2^{\mathsf{rf}}$ for which Lemma 2.8 fails.

requirements (M6a, M7a and Lemma 2.8), these ensure coherence, publication, subscription and other idioms. For example, consider the following:

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

$$\begin{tikzcd} \boxed{\mathsf{W}x0} \arrow[r] & \boxed{\mathsf{W}x1} \arrow[r] & \boxed{\mathsf{W}^{\mathsf{ra}}y1} \arrow[r, "\mathsf{rf}"] & \boxed{\mathsf{R}^{\mathsf{ra}}y1} \arrow[r] & \boxed{\mathsf{R}x0} \arrow[lll, bend right] \end{tikzcd} \tag{PUB}$$

The execution is disallowed due to the evident cycle. All of the order shown is required at top-level: The intra-thread order comes from S7c: $(\mathsf{W}x0) \rightarrow (\mathsf{W}x1)$ is required by $\bowtie_{\mathsf{co}}$. $(\mathsf{W}x1) \rightarrow (\mathsf{W}^{\mathsf{ra}}y1)$ and $(\mathsf{R}^{\mathsf{ra}}y1) \rightarrow (\mathsf{R}x0)$ are required by $\ltimes_{\mathsf{sync}}$. The cross-thread order is required by fulfillment: T2b requires that all top-level reads are in the image of $\xrightarrow{\mathsf{rf}}$. M6a ensures that $(\mathsf{W}^{\mathsf{ra}}y1) \xrightarrow{\mathsf{rf}} (\mathsf{R}^{\mathsf{ra}}y1)$, and S7b subsequently ensures that $(\mathsf{W}^{\mathsf{ra}}y1) \rightarrow (\mathsf{R}^{\mathsf{ra}}y1)$. The *antidependency* $(\mathsf{R}x0) \rightarrow (\mathsf{W}x1)$ is required by M7a. (Alternatively, we could have $(\mathsf{W}x1) \rightarrow (\mathsf{W}x0)$, again resulting in a cycle.)

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

2.5 Termination

In top-level pomsets, T1 requires that $\checkmark$ is a tautology, capturing termination. Terminated pomsets are often called *complete*, whereas nonterminated pomsets are *incomplete*.

Ignoring predicate transformers, the structural rules, P5 and S5, take $\checkmark$ to be $\checkmark_1 \land \checkmark_2$. This is as expected: the program terminates if both subprograms terminate.

The interesting rules are *READ*, *FENCE*, and *WRITE*.

In *READ*, there is no restriction on $\checkmark$ for relaxed reads. From this, it is easy to see that $[\![r := x]\!]_1 \supseteq [\![\mathtt{skip}]\!]_1$ is a valid refinement (where the default mode is rlx).

In *FENCE*, instead, F5 ensures that all fences are included at top-level.

In *WRITE*, W5b is similar. In addition, W5a ensures that top-level pomsets do not include bogus writes. Suppose $P \in [\![x := 1]\!]_1$. As we noted above, P can include $(\mathsf{W}xv)$, for any value v. At top-level, however, W5a requires that $\checkmark$ implies $1{=}v$.

In the structural rules, if $d \in E_1$ and $e \in E_2$, we say that d and e *coalesce* if $d = e$.

S1 allows *mumbling* [Brookes 1996] by coalescing events. For example $[\![x := 1\,;\ x := 1]\!]_1$ includes the singleton pomset $\boxed{\mathsf{W}x1}$. From this it is easy to see that $[\![x := 1\,;\ x := 1]\!]_1 \supseteq [\![x := 1]\!]_1$ is a valid refinement. It is equally obvious that $[\![x := 1]\!] \not\supseteq [\![x := 1\,;\ x := 1]\!]$ is not a valid refinement, since the latter includes a two-element pomset, but the former does not.[2]

2.6 Data Dependencies, Preconditions, and Predicate Transformers

In top-level pomsets, T2a requires that every precondition $\kappa(e)$ is a tautology.

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

Before looking at the details, it is useful to have a high-level view of how nontrivial preconditions and predicate transformers are introduced. (We discuss address dependencies in §4.2.)

Preconditions are introduced in:	Predicate transformers are introduced in:
(S3d) for release actions,	(R4a) for reads in the dependent set,
(C3) for control dependencies,	(R4b) for reads in the independent set,
(W3) for data dependencies on writes.	(W4) for writes.

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

The rules track dependencies. We discuss data dependencies (w3) here and control dependencies (c3) in §2.7. Unless otherwise noted, we assume pomsets are *complete* and *augment-minimal*. We do not discuss s3d further. It simply ensures that all writes are present before a release, even for incomplete pomsets (see §2.5).

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

$$\begin{array}{cc} r := x & y := r \\ \boxed{(x{=}r \lor v{=}r) \Rightarrow \psi}\ \ \boxed{\mathsf{R}\,x\,v}^{d} \dashrightarrow \boxed{v{=}r \Rightarrow \psi} & \boxed{\psi[r/y]}\ \ \boxed{r{=}w \mid \mathsf{W}\,y\,w}^{e} \dashrightarrow \boxed{\psi[r/y]} \end{array} \qquad (\ddagger)$$

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

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

For the read, the dependent transformer $\tau_1^{\{d\}}$ is $1{=}r \Rightarrow \psi$; the independent transformer $\tau_1^{\emptyset}$ is $(x{=}r \lor 1{=}r) \Rightarrow \psi$. These are determined by R4a and R4b, respectively. For the write both $\tau_2^{\{e\}}$ and $\tau_2^{\emptyset}$ are $\psi[r/y]$, as are determined by w4. Combining these into a single pomset, we have:

$$\begin{array}{c} r := x\,;\, y := r \\ \boxed{(x{=}r \lor 1{=}r) \Rightarrow \psi[r/y]}\ \ \boxed{\mathsf{R}\,x\,1}^{d} \dashrightarrow \boxed{1{=}r \Rightarrow \psi[r/y]} \qquad \boxed{\phi \mid \mathsf{W}\,y\,1}^{e} \end{array}$$

By s4, predicate transformers are determined by composition; thus $\tau^D(\psi)$ is $\tau_1^D(\tau_2^D(\psi))$. Since the transformer does not depend on whether the write is included, we do not draw dependencies for the write in the diagram.

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

- If $d \leq e$ then we apply the dependent transformer and $\phi = (1{=}r \Rightarrow r{=}1)$, a tautology.
- If $d \not\leq e$ then we apply the independent transformer and $\phi = ((x{=}r \lor 1{=}r) \Rightarrow r{=}1)$. Under the assumption that r is bound, this is logically equivalent to $(x{=}1)$. (We make this more precise in §4.1.)

Eliding transformers, the two outcomes are:

$$\begin{array}{cc} r := x\,;\, y := r & r := x\,;\, y := r \\ \boxed{\mathsf{R}\,x\,1}^{d} \rightarrow \boxed{\mathsf{W}\,y\,1}^{e} & \boxed{\mathsf{R}\,x\,1}^{d} \quad \boxed{x{=}1 \mid \mathsf{W}\,y\,1}^{e} \end{array}$$

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

$$\begin{array}{cc} x := 1 & x := 1\,;\, r := x\,;\, y := r \\ \boxed{\psi[1/x]}\ \ \boxed{1{=}1 \mid \mathsf{W}\,x\,1}^{c} \dashrightarrow \boxed{\psi[1/x]} & \boxed{1{=}1 \mid \mathsf{W}\,x\,1}^{c} \quad \boxed{\mathsf{R}\,x\,1}^{d} \quad \boxed{1{=}1 \mid \mathsf{W}\,y\,1}^{e} \end{array}$$

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

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

$$\begin{array}{cc} r := x\,;\, y := r & r := x\,;\, y := r \\ \boxed{\mathsf{R}\,x\,0}^{d} \rightarrow \boxed{0{=}r \Rightarrow r{=}1 \mid \mathsf{W}\,y\,1}^{e} & \boxed{\mathsf{R}\,x\,0}^{d} \quad \boxed{(x{=}r \lor 0{=}r) \Rightarrow r{=}1 \mid \mathsf{W}\,y\,1}^{e} \end{array}$$

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

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

$$\begin{array}{c} r := x\,;\, y := 1 \\ \boxed{(x{=}r \lor 0{=}r) \Rightarrow \psi[r/y]}\ \ \boxed{\mathsf{R}\,x\,0}^{d} \dashrightarrow \boxed{0{=}r \Rightarrow \psi[r/y]} \qquad \boxed{(x{=}r \lor 0{=}r) \Rightarrow 1{=}1 \mid \mathsf{W}\,y\,1}^{e} \end{array}$$

2.7 Control Dependencies

In $IF(\phi, \mathcal{P}_1, \mathcal{P}_2)$, the predicate transformer (c4) is $(\phi \wedge \tau_1^D(\psi)) \vee (\neg\phi \wedge \tau_2^D(\psi))$, which is the disjunctive equivalent of Dijkstra's conjunctive formulation: $(\phi \Rightarrow \tau_1^D(\psi)) \wedge (\neg\phi \Rightarrow \tau_2^D(\psi))$.

For events from E_1, c3a requires $\phi \wedge \kappa_1(e)$. For events from E_2, c3b requires $\neg\phi \wedge \kappa_2(e)$. For coalescing events in $E_1 \cap E_2$, c3c requires $(\phi \wedge \kappa_1(e)) \vee (\neg\phi \wedge \kappa_2(e))$. This semantics allows common code to be lifted out of a conditional, validating the transformation $[\![\mathtt{if}(M)\{S\}\,\mathtt{else}\,\{S\}]\!] \supseteq [\![S]\!]$. The use of *extends* in c7a and c6a ensures that no new order is introduced between events in $E_1 \cap E_2$ when coalescing; see §3.3.

By allowing events to coalesce, c3c ensures that control dependencies are calculated semantically. For example, consider $P \in [\![\mathtt{if}(r{=}1)\{y := r\}\,\mathtt{else}\,\{y := 1\}]\!]$, which is build from $P_1 \in [\![y := r]\!]$ and $P_2 \in [\![y := 1]\!]$ such as:

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

$$\boxed{r{=}1 \mid \mathsf{W}\,y\,1} \qquad\qquad \boxed{1{=}1 \mid \mathsf{W}\,y\,1} \qquad\qquad \boxed{(r{=}1 \Rightarrow r{=}1) \wedge (r{\neq}1 \Rightarrow 1{=}1) \mid \mathsf{W}\,y\,1}$$

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

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

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

$$\boxed{(x{=}r \vee v{=}r) \Rightarrow \psi}\ \boxed{\mathsf{R}\,x\,v} \xrightarrow{d} \boxed{v{=}r \Rightarrow \psi} \qquad\qquad \boxed{r{=}1 \Rightarrow \psi[1/y]}\ \boxed{r{=}1 \mid \mathsf{W}\,y\,w} \xrightarrow{e} \boxed{r{=}1 \Rightarrow \psi[1/y]}$$

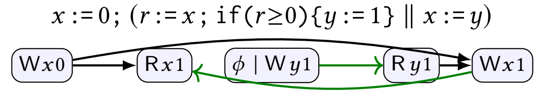
Reasoning as we did for (‡) in §2.6, there are two possibilities:

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

$$\boxed{\mathsf{R}\,x\,1}^{d} \rightarrow \boxed{\mathsf{W}\,y\,1}^{e} \qquad\qquad \boxed{\mathsf{R}\,x\,1}^{d}\ \ \boxed{x{=}1 \mid \mathsf{W}\,y\,1}^{e}$$

As another example, consider JMM causality test case 1 [Pugh 2004]:

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



The precondition ϕ is $((1{=}r \vee x{=}r) \Rightarrow r{\geq}0)[0/x]$ which is $((1{=}r \vee 0{=}r) \Rightarrow r{\geq}0)$ which is a tautology.

2.8 Reordering Transformations

The semantics validates many peephole optimizations. Most apply only to relaxed access.

$$[\![r := x\,;\ s := y]\!]_1 = [\![s := y\,;\ r := x]\!]_1 \qquad \text{if } r \neq s$$
$$[\![x := M;\ y := N]\!]_1 = [\![y := N;\ x := M]\!]_1 \qquad \text{if } x \neq y$$
$$[\![x := M;\ s := y]\!]_1 = [\![s := y\,;\ x := M]\!]_1 \qquad \text{if } x \neq y \text{ and } s \notin \mathsf{id}(M)$$

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

$$[\![x^{\mu} := M;\ s := y]\!]_1 \supseteq [\![s := y\,;\ x^{\mu} := M]\!]_1 \qquad \text{if } x \neq y \text{ and } s \notin \mathsf{id}(M)$$
$$[\![x := M;\ s := y^{\mu}]\!]_1 \supseteq [\![s := y^{\mu};\ x := M]\!]_1 \qquad \text{if } x \neq y \text{ and } s \notin \mathsf{id}(M)$$

2.9 Associativity and Skolemization

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

(R4a′) if $(E \cap D) \neq \emptyset$ then $\tau^D(\psi) \vDash \psi[v/r]$, Recall that R4c says that ψ must be independent of r in order to appear in a top-level pomset: if $E = \emptyset$ then $\tau^D(\psi) \vDash \psi$. This choice for R4c is forced by Definition 2.3, which states that the predicate transformer for a small subset of E must imply the transformer for a larger subset.

Sadly, this definition fails associativity.

Consider the following, eliding transformers for the writes:

$r := y$	$x := !r$	$x := !!r$	$x := 0$
$\boxed{(y{=}r \lor 1{=}r) \Rightarrow \psi}$ $(\mathsf{R}\,y\,1) \dashrightarrow \boxed{1{=}r \Rightarrow \psi}$	$(r{=}0 \mid \mathsf{W}\,x\,1)$	$(r{\neq}0 \mid \mathsf{W}\,x\,1)$	$(\mathsf{W}\,x\,0)$

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

$r := y$	$x := !r\,;\, x := !!r\,;\, x := 0$	$r := y\,;\, (x := !r\,;\, x := !!r\,;\, x := 0)$
$(\mathsf{R}\,y\,1)$	$(\mathsf{W}\,x\,1) \rightarrow (\mathsf{W}\,x\,0)$	$(\mathsf{R}\,y\,1)$ $\quad (\mathsf{W}\,x\,1) \rightarrow (\mathsf{W}\,x\,0)$

The precondition of $(\mathsf{W}\,x\,1)$ is a tautology. Associating to the left and the coalescing, instead:

$r := y\,;\, x := !r$	$x := !!r\,;\, x := 0$	$(r := y\,;\, x := !r)\,;\, (x := !!r\,;\, x := 0)$
$(\mathsf{R}\,y\,1)$ $\quad ((y{=}r \lor 1{=}r) \Rightarrow r{=}0 \mid \mathsf{W}\,x\,1)$	$(r{\neq}0 \mid \mathsf{W}\,x\,1) \rightarrow (\mathsf{W}\,x\,0)$	$(\mathsf{R}\,y\,1)$ $\quad (\phi \mid \mathsf{W}\,x\,1) \rightarrow (\mathsf{W}\,x\,0)$

where $\phi = ((y{=}r \lor 1{=}r) \Rightarrow r{=}0) \lor (r{\neq}0)$. The precondition ϕ is not a tautology. In a top-level pomset, this forces dependency order from $(\mathsf{R}\,y\,1)$ to $(\mathsf{W}\,x\,1)$.

Our solution is to Skolemize, replacing uses of $\psi[v/r]$ by $(r{=}v) \Rightarrow \psi$, for uniquely chosen r. The proof of associativity requires that predicate transformers distribute through disjunction (Definition 2.2). The attempt to define predicate transformers using substitution fails for R4c because the predicate transformer $\tau(\psi) = (\forall r)\psi$ does not distribute through disjunction: $\tau(\psi_1 \lor \psi_2) = (\forall r)(\psi_1 \lor \psi_2) \neq ((\forall r)(\psi_1)) \lor ((\forall r)(\psi_2)) = \tau(\psi_1) \lor \tau(\psi_2)$. Since $\tau(\psi) = (\forall r)\psi$ does not distribute through disjunction, we use $\tau(\psi) = \psi$ instead (which trivially distributes through disjunction). Unforutunately, this change means we cannot use substitution, since ψ does not imply $\psi[v/r]$. Fortunately, Skolemizing solves this problem, since ψ implies $(r{=}v) \Rightarrow \psi$.

2.10 Comparison with Weakest Preconditions

We compare traditional transformers to the dependent-case transformers of Figure 1; 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 §2–3.

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 on the left below

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

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. Our predicate transformers for the dependent case are shown on the right above. Only the read rule differs from the traditional one.

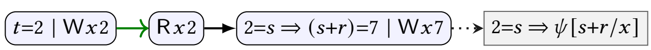
For programs where every register is bound and every read is fulfilled, our dependent transformers are the same as the traditional ones. In our semantics, thus, we only consider totally-ordered executions where every read could be fulfilled by prepending some writes. For example, we ignore pomsets of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

For example, let S_i be defined:

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

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

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



The predicate transformers are:

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

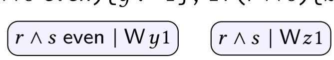
2.11 Substitutions

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

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

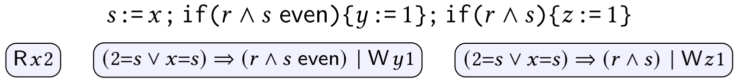
Perhaps surprisingly, this semantics is incomparable with that of Figure 1. Consider the following:

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



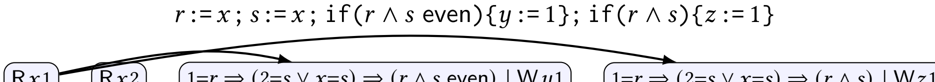
Prepending $(s := x)$, we get the same result regardless of whether we substitute $[s/x]$, since x does not occur in either precondition. Here we show the independent case:

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

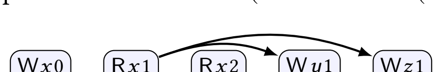


Since the preconditions mention x, prepending $(r := x)$, we now get different results depending on whether we perform the substitution. Without any substitution, we have:

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

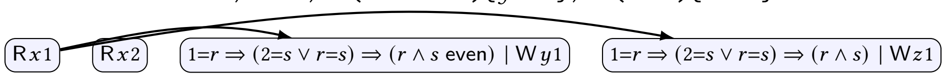


Prepending $(x := 0)$, which substitutes $[0/x]$, the precondition of $(\mathsf{W}y1)$ becomes $(1{=}r \Rightarrow (2{=}s \vee 0{=}s) \Rightarrow (r \wedge s\ \text{even}))$, which is a tautology, whereas the precondition of $\mathsf{W}z1$ becomes $(1{=}r \Rightarrow (2{=}s \vee 0{=}s) \Rightarrow (r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

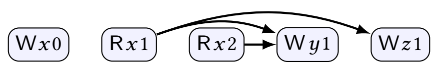


The situation reverses with the substitution $[r/x]$:

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



Prepending $(x := 0)$:



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

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

3 ARM

For simplicity, we restrict to top level parallel composition and ignore fences[3].

3.1 Arm executions

Definition 3.1. 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) $\lambda : E \longrightarrow \mathcal{A}$ 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,

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 3.2. Execution G is $(\mathsf{co}, \mathsf{rf}, \mathsf{gcb})$*-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 $e \xrightarrow{\mathsf{gcb}} c$,

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

(A7b) if $e \xrightarrow{\textsf{lob}} c$ then either $e \xrightarrow{\textsf{gcb}} c$ or $(\exists d)\ d \xrightarrow{\textsf{rf}} e$ and $d \xrightarrow{\textsf{poloc}} e$ but not $d \xrightarrow{\textsf{lob}} c$.

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

(A5) and (A6), as for EGC,

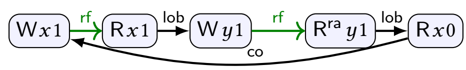
(A8) cb $\supseteq$ (co $\cup$ lob) is a linear order such that if $d \xrightarrow{\textsf{rf}} e$ then either

(A8a) $d \xrightarrow{\textsf{cb}} e$ and if $\lambda(c)$ blocks $\lambda(e)$ then either $c \xrightarrow{\textsf{cb}} d$ or $e \xrightarrow{\textsf{cb}} c$, or

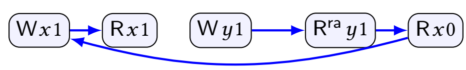
(A8b) $d \xleftarrow{\textsf{cb}} e$ and $d \xrightarrow{\textsf{poloc}} e$ and $(\nexists c)\ \lambda(c)$ blocks $\lambda(e)$ and $d \xrightarrow{\textsf{poloc}} c \xrightarrow{\textsf{poloc}} e$.

Alglave et al. [2021] explain EGC and EC using the following example, which is allowed by Arm.[4]

$$x := 1\,;\ r := x\,;\ y := r \parallel 1 := y^{\textsf{ra}}\,;\ s := x$$

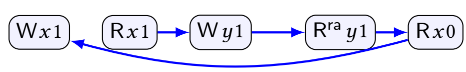


EGC drops lob-order in the first thread using A7b, since (Wx1) is not lob-ordered before (Wy1).



(gcb)

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



(cb)

3.2 Arm Compilation 1

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

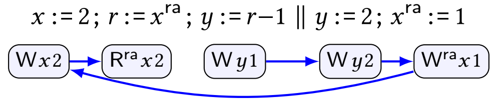
Definition 3.3. Let $[\![\cdot]\!]_2$ be defined as in Figure 1, replacing the definition of κ_2' with:

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

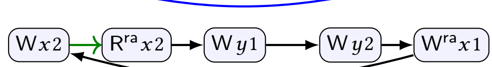
Even with this small change, the optimal lowering for Arm8 is unsound for our semantics. The optimal lowering maps relaxed access to `ldr`/`str` and non-relaxed access to `ldar`/`stlr` [Podkopaev et al. 2019]. In this section, we consider a suboptimal strategy, which lowers non-relaxed reads to (`dmb.sy`; `ldar`). Significantly, we retain the optimal lowering for relaxed access. In the next section we recover the optimal lowering by adopting an alternative semantics for s7.

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

$$x := 2\,;\ r := x^{\textsf{ra}}\,;\ y := r{-}1 \parallel y := 2\,;\ x^{\textsf{ra}} := 1$$



(gcb)



($\leq$)

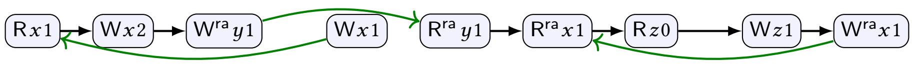
This attempted execution is allowed by Arm8, but disallowed by our semantics.

If the read of x in the execution above is changed from acquiring to relaxed, then our semantics allows the execution, using the independent case for the read and satisfying the precondition of (Wy1) by prepending (Wx2). It may be tempting, therefore to adopt a strategy of *downgrading*

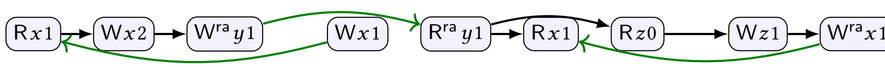
[4]We have changed an address dependency in the first thread to a data dependency.

acquires in certain cases. Unfortunately, it is not possible to do this locally without violating important idioms such as publication. For example, consider that $(\mathsf{R}^{\mathsf{ra}}x1)$ *not* possible for the second thread in the following attempted execution, due to publication of $(\mathsf{W}x2)$ via y:

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



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



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

Theorem 3.4. *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.

Lemma 3.5. *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 3.6. *Suppose G is $(\mathsf{co}, \mathsf{rf}, \mathsf{gcb})$-valid and $c \xrightarrow{\mathsf{poloc}} e$, where $\lambda(c)$ blocks $\lambda(e)$. Then $c \xrightarrow{\mathsf{gcb}} e$.*

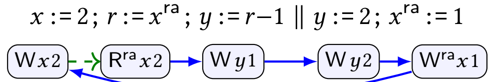
Proof. By way of contradiction, assume $e \xrightarrow{\mathsf{gcb}} c$. If $c \xrightarrow{\mathsf{rf}} e$ then by A7 we must also have $c \xrightarrow{\mathsf{gcb}} e$, contradicting the assumption that gcb is a total order. Otherwise that there is some $d \neq c$ such that $d \xrightarrow{\mathsf{rf}} e$, 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 A7b can also be removed from the pomset. In order for A7b to remove order from e to c, we must have $d \xrightarrow{\mathsf{rf}} e$ and $d \xrightarrow{\mathsf{poloc}} e$ but not $d \xrightarrow{\mathsf{lob}} c$. Because of our suboptimal lowering, it must be that e is a relaxed read; otherwise the `dmb.sy` would require $d \xrightarrow{\mathsf{lob}} c$. Thus we know that 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. □

3.3 Arm Compilation 2

We can achieve optimal lowering for Arm by weakening the semantics of sequential composition slightly. In particular, we must lose Lemma 2.8, which states that $d \xrightarrow{\mathsf{rf}} e$ implies $d \leq e$. Revisiting the example in the last subsection, we essentially mimic the EC characterization:

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



(cb)

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

The change to the semantics is small: we weaken relationship between rf and $\leq$ in s7b. Rather than ensuring that there is no *global* blocker for a sequentially fulfilled read s7b, we require only that there is no *thread-local* blocker s7b$^{\mathsf{rf}}$. This change both allows and requires us to weaken the definition of *delays* to drop write-to-read order from $\bowtie_{\mathsf{co}}$.

Definition 3.7. Let $[\![\cdot]\!]_2^{\mathsf{rf}}$ be defined as for $[\![\cdot]\!]_2$ in Definition 3.3/Figure 1, changing s7b and s7c:

(s7b$^{\mathsf{rf}}$) if $\lambda_1(c)$ blocks $\lambda_2(e)$ then $d \xrightarrow{\mathsf{rf}} e$ implies $c \le d$,
(s7c$^{\mathsf{rf}}$) if $\lambda_1(d)$ delays$'$ $\lambda_2(e)$ then $d \le e$,
where *delays*$'$ replaces $\bowtie_{\mathsf{co}}$ in Definition 2.1 of *delays* by $\ltimes_{\mathsf{lws}} = \{(\mathsf{W}x, \mathsf{W}x),\ (\mathsf{R}x, \mathsf{W}x)\}$.

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

With the weakening of s7b$^{\mathsf{rf}}$, we must be careful not to allow spurious pairs to be added to the rf relation. The use of *extends* in c6a does this, ensuring that no new rf is introduced between events in $E_1 \cap E_2$ when coalescing. This is necessary to ensure that $[\![\mathtt{if}(b)\{r := x \parallel x := 1\}\mathtt{else}\{r := x;\ x := 1\}]\!]_2^{\mathsf{rf}}$ does not include $\mathsf{R}x1 \rightleftarrows \mathsf{W}x1$, taking rf from the left and $\le$ from the right.

We emphasize that Lemma 2.8 fails for $[\![\cdot]\!]_2^{\mathsf{rf}}$, since $d \xrightarrow{\mathsf{rf}} e$ may not imply $d \le e$ when d and e come from different sides of a sequential composition. This means that rf must be verified during pomset construction, rather than post-hoc. The following lemma gives a post-hoc verification technique for rf, using program order (po).[5]

Lemma 3.8. *Any P in the image of $[\![\cdot]\!]_2^{\mathsf{rf}}$ is top-level iff for every $d \xrightarrow{\mathsf{rf}} e$ either*

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

Theorem 3.9. *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 $\le_2 = \mathsf{cb}_1$.*

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 Theorem 3.9 is not limited by the assumption that $\mathsf{cb}_1 \supseteq \mathsf{fr}_1$:

Lemma 3.10. *Suppose G is* ec*-valid via $(\mathsf{co}, \mathsf{rf}, \mathsf{cb})$. Then there a permutation cb' of cb such that G is* ec*-valid via $(\mathsf{co}, \mathsf{rf}, \mathsf{cb}')$ and $\mathsf{cb}' \supseteq \mathsf{fr}$, where fr is defined in* 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 $\mathsf{cb} \supseteq \mathsf{co}$, 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. □

4 ADDITIONAL FEATURES

In the paper so far, we have assumed that registers are assigned at most once. We have done this primarily for readability. In the first subsection below, we drop this assumption, instead using substitution to rename registers. We use the set $\mathcal{S}_{\mathcal{E}} = \{s_e \mid e \in \mathcal{E}\}$. By assumption (§2.1), these registers do not appear in programs: $S[N/s_e] = S$. The resulting semantics satisfies redundant read elimination.

[5] It is obvious how to enhance the semantics of most operators to define po. When combining pomsets using the conditional, the obvious definition of po may result in cycles, since po-ordered events may coalesce. In this case we include a separate pomset for each way of breaking these po cycles.

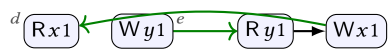
In the rest of this section we consider several orthogonal features: address calculation, if-closure, read-modify-write operations, and access elimination.

These extensions preserve all of the valid transformations discussed thus far. We state the extensions with respect to the base semantics of Figure 1, but they apply equally to the variants described in §3.

4.1 Register Recycling and Redundant Read Elimination

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

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



This execution is not allowed by the semantics $[\![\cdot]\!]_1$ of Figure 1: the precondition of e in the independent case is

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

which is not a tautology, and thus $[\![\cdot]\!]_1$ requires order from d to e.

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

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

which is a tautology.

Definition 4.1. Let $[\![\cdot]\!]_3$ be defined as in Figure 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 $\tau^D(\psi) \vDash (v{=}s_e \lor x{=}s_e) \Rightarrow \psi[s_e/r]$,
(R4c) if $E = \emptyset$ then $(\forall s)\ \tau^D(\psi) \vDash \psi[s/r]$.

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

$$[\![r := x^\mu\,;\ s := x^\mu]\!]_3 \supseteq [\![r := x^\mu\,;\ s := r]\!]_3$$

4.2 Address Calculation

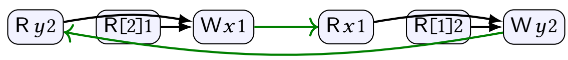
Inevitably, address calculation complicates the definitions of *WRITE* and *READ*.

Definition 4.2. Let $[\![\cdot]\!]_4$ be defined as in Figure 1, changing *WRITE* and *READ*:
If $P \in \mathit{WRITE}(L, M, \mu)$ then $(\exists \ell \in \mathcal{V})\ (\exists v \in \mathcal{V})$
(W1) if $d, e \in E$ then $d = e$,
(W2) $\lambda(e) = \mathsf{W}^\mu[\ell]v$,
(W3) $\kappa(e) \vDash L{=}\ell \land M{=}v$,
(W4a) if $E \neq \emptyset$ then $\tau^D(\psi) \vDash (L{=}\ell) \Rightarrow \psi[M/[\ell]]$,
(W4b) if $E = \emptyset$ then $(\forall k)\ \tau^D(\psi) \vDash (L{=}k) \Rightarrow \psi[M/[k]]$
(W5a) if $E \neq \emptyset$ then $\checkmark \vDash L{=}\ell \land M{=}v$,
(W5b) if $E = \emptyset$ then $\checkmark \vDash \mathsf{ff}$.

If $P \in \mathit{READ}(r, L, \mu)$ then $(\exists \ell \in \mathcal{V})\ (\exists v \in \mathcal{V})$
(R1) if $d, e \in E$ then $d = e$,
(R2) $\lambda(e) = \mathsf{R}^\mu[\ell]v$
(R3) $\kappa(e) \land L{=}\ell$,
(R4a) $(\forall e \in E \cap D)\ \tau^D(\psi) \vDash (L{=}\ell \Rightarrow v{=}s_e) \Rightarrow \psi[s_e/r]$,
(R4b) $(\forall e \in E \setminus D)\ \tau^D(\psi) \vDash ((L{=}\ell \Rightarrow v{=}s_e) \lor (L{=}\ell \Rightarrow [\ell]{=}s_e)) \Rightarrow \psi[s_e/r]$,
(R4c) $(\forall s)$ if $E = \emptyset$ then $\tau^D(\psi) \vDash \psi[s/r]$.

The combination of read-read independency (Definition 3.3) and address calculation is somewhat delicate. Consider the following program, from [Jagadeesan et al. 2020, §5], where initially $x = 0$, $y = 0$, $[0] = 0$, $[1] = 2$, and $[2] = 1$. It should only be possible to read 0, disallowing the attempted execution below:

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

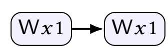


This execution is possible, however, if we replace $(L{=}\ell \Rightarrow v{=}s_e)$ by $(v{=}s_e)$ in R4a. In this case, there is not necessarily a dependency order from $(\mathsf{R}y2)$ to $(\mathsf{W}x1)$.

4.3 If-Closure

In order to model sequential composition, we must allow inconsistent predicates in a single pomset, unlike [Jagadeesan et al. 2020]. For example, if $S = (x := 1)$, then $[\![\cdot]\!]_1$ does *not* allow:

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



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

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

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

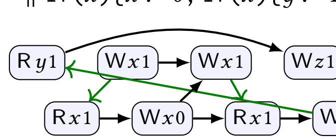
Definition 4.3 allows the execution, by splitting the middle command:

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

Coalescing events gives the desired result.

This is not simply a theoretical question; it is observable. For example, $[\![\cdot]\!]_1$ does not allow the following.

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



Definition 4.3. Let $[\![\cdot]\!]_5$ be defined as in Figure 1, changing *WRITE* and *READ*:
If $P \in \mathit{WRITE}(x, M, \mu)$ then $(\exists v : E \to \mathcal{V})\ (\exists \theta : E \to \Phi)$

- (w1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
- (w2) $\lambda(e) = \mathsf{W}^{\mu} x v_e$,
- (w3) $\kappa(e) \vDash \theta_e \wedge M{=}v_e$,
- (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 $\theta_d \wedge \theta_e$ 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) $(\forall e \in E \setminus D)\ \tau^D(\psi) \vDash \theta_e \Rightarrow (v_e{=}s_e \vee x{=}s_e) \Rightarrow \psi[s_e/r]$,
(R4c) $(\forall s)\ \tau^D(\psi) \vDash (\bigwedge_{e \in E} \neg\theta_e) \Rightarrow \psi[s/r]$.

4.4 Combining Address Calculation and If-Closure

Definition 4.2 is naive with respect to merging events. Consider the following example:

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

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

Merging, we have:

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

$$ {}^{c}\boxed{r{=}1 \mid \mathsf{W}[1]0} \quad {}^{d}\boxed{r{=}0 \vee r{=}1 \mid \mathsf{W}[0]0} \longrightarrow {}^{e}\boxed{r{=}0 \mid \mathsf{W}[0]1}$$

The precondition of $\mathsf{W}[0]0$ is a tautology; however, this is not possible for $([r] := 0\, ;\, [0] := !r)$ alone, using Definition 4.2.

Definition 4.4, enables this execution using if-closure. Under this semantics, we have:

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

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

Sequencing and merging:

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

$$ {}^{c}\boxed{r{=}1 \mid \mathsf{W}[1]0} \quad {}^{d}\boxed{r{=}0 \vee r{=}1 \mid \mathsf{W}[0]0} \longrightarrow {}^{e}\boxed{r{=}0 \mid \mathsf{W}[0]1}$$

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

Definition 4.4. Let $[\![\cdot]\!]_6$ be defined as in Figure 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) $\tau^D(\psi) \vDash \theta_e \Rightarrow (L{=}\ell) \Rightarrow \psi[M/[\ell]]$,
(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) $(\forall s)\ \tau^D(\psi) \vDash (\bigwedge_{e \in E} \neg\theta_e) \Rightarrow \psi[s/r]$.

4.5 Read-Modify-Write Operations

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

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.

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.

Definition 4.5. Extend the definition of a pomset as follows.

(M8) $\mathsf{rmw} : E \rightarrow E$ 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) $\mathsf{rmw} = (\mathsf{rmw}_1 \cup \mathsf{rmw}_2)$,

To define specific operations, we extend the syntax:

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

The corresponding semantic functions are as follows.

Definition 4.6. 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 $(\exists P_1 \in \mathit{SEQ}(\mathit{READ}'(r,\ x,\ \mu_1),\ \mathit{WRITE}(x,\ r{+}M,\ \mu_2)))$
 (U1) if $\lambda_1(e)$ is a write then there is a read $\lambda_1(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.

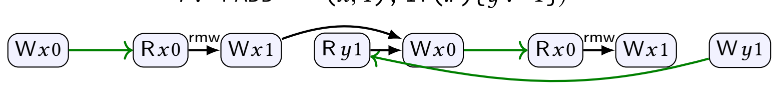
If $P \in \mathit{EXCHG}(r, x, M, \mu_1, \mu_2)$ then $(\exists P_1 \in \mathit{SEQ}(\mathit{READ}'(r,\ x,\ \mu_1),\ \mathit{WRITE}(x,\ M,\ \mu_2)))$
 (U1) if $\lambda_1(e)$ is a write then there is a read $\lambda_1(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.

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

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

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

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



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

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

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

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

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

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

For case analysis of RMWs, we can use a general purpose expansion operator:

Definition 4.7. 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) $\bigvee_i \theta_i \vDash \mathrm{tt}$,
(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) $\leq\; = \bigcup_i \leq_i$.

4.6 Access Elimination

As noted in §2.5, the semantics of Figure 1 validates elimination of irrelevant relaxed reads. In §4.1, we discussed redundant read elimination. Figure 1 also validates elimination of writes of the same value. However, Figure 1 does not validate general write elimination, where, for example, $x := 1;\ x := 2$ can be refined to $x := 2$. A list of valid merge operations can be found in [Chakraborty and Vafeiadis 2017, §E] and [Kang 2019, §7.1]. For examples of unsafe merges and reorderings, see [Chakraborty and Vafeiadis 2017, §D].

To accommodate such merges, define merge : $\mathcal{A} \times \mathcal{A} \to 2^{\mathcal{A}}$ as follows, where $\sqcup$ is the least upper bound with respect the order on modes from §2.2.

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

Then we can replace s2 in Figure 1 by:
(s2a) if $e \in E_1 \setminus E_2$ then $\lambda(e) = \lambda_1(e)$,
(s2b) if $e \in E_2 \setminus E_1$ then $\lambda(e) = \lambda_2(e)$,
(s2c) 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,

For associativity, it is important that merge takes the join of two modes.

5 FUTURE WORK

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

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

In §3.2 we presented a compilation strategy to Arm8 for a simplified model, but which introduces fences to acquiring reads. These fences are not required in §3.3, but at the cost of model complexity. It would be illuminating to find out what the performance penalty is for these fences.

An earlier version of this paper has been mechanized in Agda; it would be reassuring to update the mechanization to bring it in line with the current state.

REFERENCES

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

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

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

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

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

Soham Chakraborty and Viktor Vafeiadis. 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, 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

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

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 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) $\checkmark_2 \vDash \checkmark_1$,
(6) $(\forall d \in E_2)\ (\forall e \in E_2)\ d \mathrel{\mathsf{rf}_2} e$ iff $d \mathrel{\mathsf{rf}_1} e$,,
(7a) $(\forall d \in E_2)\ (\forall e \in E_2)\ d \le_2 e$ iff $d \le_1 e$,
(7b) $(\forall d \in E_1)\ (\forall e \in E_2)$ if $d \le_1 e$ then $d \in E_2$.

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

First, downset closure fails for Definition 3.3, because it does not enforce read-read dependencies. Consider

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

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

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

$r := x$	$\texttt{if}(!r)\{s := y\}$
$\boxed{\mathsf{R}x0}$	$\emptyset$

Attempting to get (Ry0), we instead get:

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

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

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

Second, the semantics is not downset closed because the independency reasoning of R4b is only applicable for pomsets where the ignored read is present! Revisiting JMM causality test case 1 from the end of §2.7:

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

$$\boxed{\mathsf{W}\,x0} \qquad\qquad \boxed{\mathsf{R}\,x1} \qquad\qquad \boxed{r{\ge}0 \mid \mathsf{W}\,y1}$$

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

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

$$\boxed{\mathsf{W}\,x0} \longrightarrow \boxed{\mathsf{R}\,x1} \qquad \boxed{(1{=}r \vee 0{=}r) \Rightarrow r{\ge}0 \mid \mathsf{W}\,y1}$$

The precondition of (Wy1) is a tautology.

Taking the empty set for the read, however, the precondition of (Wy1) is not a tautology:

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

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

B DIFFERENCES WITH "POMSETS WITH PRECONDITIONS"

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 Figure 1, we gave the semantics of read for nonempty pomsets as:

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

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

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

The use of conjunction in R4b′ causes disjunction closure to fail because the predicate transformer $\tau(\psi) = \psi' \wedge \psi''$ does not distribute through disjunction, even assuming that the prime operations do:[6] $\tau(\psi_1 \vee \psi_2) = (\psi_1' \vee \psi_2') \wedge (\psi_1'' \vee \psi_2'') \neq (\psi_1' \wedge \psi_1'') \vee (\psi_2' \wedge \psi_2'') = \tau(\psi_1) \vee \tau(\psi_2)$. See also §2.9.

The substitutions collapse x and r, allowing local invariant reasoning, causality test case 1, discussed at the end of §2.7. Without Skolemizing it is necessary to substitute $[x/r]$, since the reverse substitution $[r/x]$ is useless when r is bound—compare with §2.11. As discussed below (Downset closure), including this substitution affects the interaction of LIR and downset closure.

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

[6] $(\psi_1 \vee \psi_2)' = (\psi_1' \vee \psi_2')$ 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. Using this definition, consider the following:

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

$$\begin{tikzcd} \boxed{\mathsf{R}\,x1} \arrow[rr, bend left] & \boxed{\mathsf{R}\,x2} \arrow[r] & \boxed{1{=}x \Rightarrow 2{=}x \Rightarrow x{<}x \mid \mathsf{W}\,y1} \end{tikzcd}$$

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

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 in the last example of §A under the semantics of this paper, which elides the middle read event:

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

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

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:

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

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 the example from §4.3.

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

Associating left and right, we have:

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

Associating into the middle, instead, we require:

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

Joining left and right, we have:

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

$$\boxed{\mathsf{W}\,x1} \quad \boxed{\mathsf{W}\,y1}$$

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:

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

Associating left, with causal strengthening:

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

Finally, merging:

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

$$\boxed{\mathsf{R}\,x\,1} \rightarrow \boxed{M \mid \mathsf{W}\,y\,1}$$

Instead, associating right:

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

$$\boxed{M \mid \mathsf{R}\,x\,1} \qquad\qquad \boxed{r{=}1 \mid \mathsf{W}\,y\,1} \quad \boxed{\neg M \mid \mathsf{R}\,x\,1}$$

Merging:

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

$$\boxed{\mathsf{R}\,x\,1} \rightarrow \boxed{\mathsf{W}\,y\,1}$$

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 the example in §4.2 is:

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

$$\boxed{r{=}2 \mid \mathsf{R}[2]1} \rightarrow \boxed{(r{=}2 \Rightarrow 1{=}s) \Rightarrow s{=}1 \mid \mathsf{W}\,x\,1}$$

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

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

$$\boxed{r{=}2 \mid \mathsf{R}[2]1} \rightarrow \boxed{r{=}2 \wedge [2]{=}1 \mid \mathsf{W}\,x\,1}$$

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.

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 $\vee$.

Internal Acquiring Reads. The proof of compilation to Arm in [Jagadeesan et al. 2020, a]ssumes that all internal reads can be eliminated. 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.

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

$$\boxed{\mathsf{W}\,x\,2} \rightarrow \boxed{\mathsf{R^{ra}}\,x\,2} \rightarrow \boxed{\mathsf{R}\,y\,0} \rightarrow \boxed{\mathsf{W}\,y\,2} \rightarrow \boxed{\mathsf{W^{ra}}\,x\,1}$$

We discussed our solution in §3.

Redundant Read Elimination. Contrary to the claim, redundant read elimination fails for [Jagadeesan et al. 2020]. We discussed redundant read elimination in §4.1. Consider JMM Causality Test Case 2, which we discussed there.

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

$$\boxed{\mathsf{R}\,x\,1} \quad \boxed{\mathsf{R}\,x\,1} \quad \boxed{\mathsf{W}\,y\,1} \rightarrow \boxed{\mathsf{R}\,y\,1} \rightarrow \boxed{\mathsf{W}\,x\,1}$$

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

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

$$\boxed{\mathsf{R}\,x\,1} \quad \boxed{\mathsf{R}\,x\,1} \quad \boxed{1{=}1 \wedge 1{=}x \wedge x{=}1 \wedge x = x \mid \mathsf{W}\,y\,1}$$

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