

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

ANONYMOUS AUTHOR(S)

Program logics and semantics tell us that when executing  $((S_1; S_2), \text{state0})$ , we execute  $(S_1, \text{state0})$  to arrive at  $\text{state1}$ , then execute  $(S_2, \text{state1})$  to arrive at the final  $\text{state2}$ . This is, of course, and abstraction. Processors execute instructions out of order, due to pipelines and caches. Compilers reorder programs even more dramatically. All of this reordering is meant to be unobservable in single-threaded code. In multi-threaded code, however, all bets are off. A formal attempt to understand the resulting mess is known as a “relaxed memory model.” The relaxed memory models that have been proposed to date either fail to address sequential composition, or overly restrict processors and compilers.

To support sequential composition, we propose adding families of predicate transformers to the existing model of “Pomsets with Preconditions,” which already supports parallel composition. When composing  $(S_1; S_2)$ , the predicate transformer used to validate the precondition of an event in  $S_2$  is chosen based on the semantic dependencies from  $S_1$  into this event. Our model retains the good properties of the prior work, including efficient implementation on Arm8, support for compiler optimizations, support for logics that prove the absence of thin-air behaviors, and a local data race freedom theorem.

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

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

## ACM Reference Format:

Anonymous Author(s). 2021. Predicate Transformers for Relaxed Memory: Sequential Composition for Concurrency Using Semantic Dependencies. *Proc. ACM Program. Lang.* 0, OOPSLA, Article 0 (October 2021), 33 pages.

## 1 INTRODUCTION

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

### 1.1 Sequential Composition

Introductory programmers are taught *sequential abstraction*: that the program  $S_1; S_2$  executes  $S_1$  before  $S_2$ . Since the late 60s, we’ve been able to explain this using logic [Hoare 1969]. In Dijkstra’s [1975] formulation, we think of programs as *predicate transformers*, where predicates describe the state of memory in the system. In the calculus of weakest preconditions, programs map postconditions to preconditions. We recall the definition of  $wp_S(\psi)$  for loop-free code below.

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

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

© 2021 Copyright held by the owner/author(s).

2475-1421/2021/10-ART0

<https://doi.org/>

For this language, the Hoare triple  $\{\phi\} S \{\psi\}$  holds exactly when  $\phi \Rightarrow wp_S(\psi)$ . We have split Dijkstra's rule for assignment (d2) into three cases. In our notation,  $r$ - $s$  range over thread-local registers, which may be assigned at most once,  $x$ - $z$  range over shared memory references, and  $M$ - $N$  range over thread-local expressions, which do *not* include  $x$ - $z$ .<sup>1</sup>

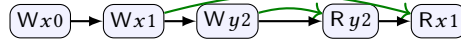
This is quite a pretty explanation of sequential computation in a sequential context. In a concurrent context, however, d2c is unsound! d2c assumes that a read from  $x$  must be fulfilled by a preceding write to  $x$ . In a concurrent context, writes may come from other threads.

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

## 1.2 Memory Models

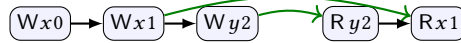
For single-threaded programs, memory can be thought of as you might expect: programs write to, and read from, memory references. This can be thought of as a total order of reads and writes (black arrows), where each read has a matching *fulfilling* write (green arrows), for example:

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



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:

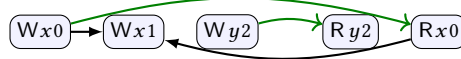
$x := 0; x := 1; y := 2 \parallel r := y; s := x$



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:

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



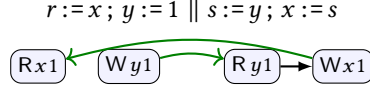
In such models, the causal order between events is important, and includes control and data dependencies, to avoid paradoxical “out of thin air” examples such as:

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

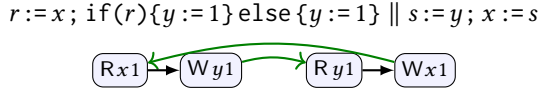


<sup>1</sup>Under these assumptions, (d2c) is equivalent to  $wp_{r:=x}(\psi) = \psi[x/r]$ .

This candidate execution forms a cycle in causal order, so is disallowed, but this depends crucially on the control dependency from (Rx1) to (Wy1), and the data dependency from (Ry1) to (Wx1). If either is missing, then this execution is acyclic and hence allowed. For example dropping the control dependency results in:



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



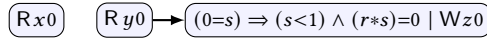
A compiler may lift the assignment  $y := 1$  out of the conditional, thus removing the dependency.

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):

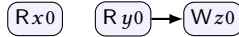
$$\text{if}(s < 1)\{z := r * s\}$$

$$(s < 1) \wedge (r * s) = 0 \mid Wz0$$

Preconditions are discharged by being ordered after a read:

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


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



### 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,

$$\llbracket x := M; y := N; S \rrbracket = \llbracket y := N; x := M; S \rrbracket \text{ if } x \neq y$$

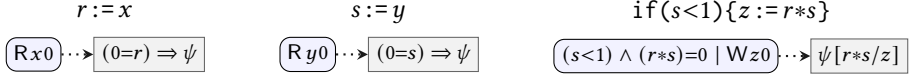
where  $S$  is the entire future computation! By formalizing sequential composition, we can show:

$$\llbracket x := M; y := N \rrbracket = \llbracket y := N; x := M \rrbracket \text{ if } x \neq y$$

Then the equivalence holds in any (sequential) context.

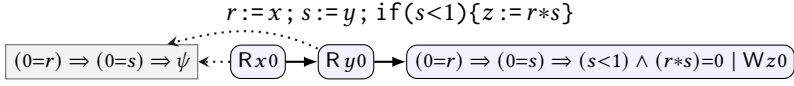
Predicate transformers are a good fit for logical models of dependency calculation, since both are concerned with preconditions and how they are transformed by sequential composition. Our first

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



The predicate transformer from the write matches [Dijkstra](#). For the reads, however, [b2c](#) defines the transformer of  $r := x$  to be  $(x=r) \Rightarrow \psi$ . Instead, we use  $(0=r) \Rightarrow \psi$ , reflecting the fact that 0 may come from a concurrent write. The obligation to find a matching write is moved from the sequential semantics of *substitution* and *implication* to the concurrent semantics of *fulfillment*.

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



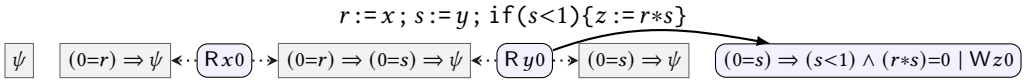
This model works for the sequentially consistent case, but needs to be weakened for the relaxed case. The key observation of this paper is that rather than working with one predicate transformer, we should work with a *family* of predicate transformers, indexed by sets of events.

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

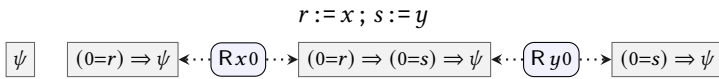


The model of sequential composition then picks which predicate transformer to apply to an event's precondition by picking the one indexed by all the events before it in causal order.

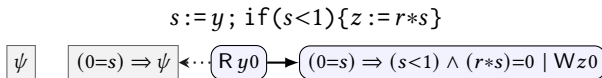
For example, we can recover the expected semantics for  $(\dagger)$  by choosing the predicate transformer which is independent of  $(Rx0)$  but dependent on  $(Ry0)$ , which is the transformer which maps  $\psi$  to  $(0=s) \Rightarrow \psi$ .



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



or to the right, with intermediate step:



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

## 1.4 Contributions

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

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

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

## 2 RELATED WORK

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

There are 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. They use step-indexing to account for loops; we expect that the same approach could be applied here. Kavanagh and Brookes [2018] define a semantics using pomsets without preconditions. Instead, their model uses syntactic dependencies, thus invalidating many compiler optimizations. They also require a fence after every relaxed read on Arm8. Pichon-Pharabod and Sewell [2016] use event structures to calculate dependencies, combined with an operational semantics that incorporates program transformations. This approach seems to require whole-program analysis.

Other studies of relaxed memory can be categorized by their approach to dependency calculation. Hardware models use syntactic dependencies [Alglave et al. 2014]. Many software models do not bother with dependencies at all [Batty et al. 2011; Cox 2016; Watt et al. 2020, 2019]. Others have strong dependencies that disallow compiler optimizations and efficient implementation, typically requiring fences for every relaxed read on Arm [Boehm and Demsky 2014; Dolan et al. 2018; Jeffrey and Riely 2016; Lahav et al. 2017; Lamport 1979]. Many of the most prominent models are 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 and OOTA5 (from [Lochbihler 2013]). The difference with our model can be understood in terms of the valid program transformations. The speculative models allow reads to be introduced, with subsequent case analysis on the value read—effectively, this can turn one read into two, with different conditional branches taken for the two copies of the read. Our model invalidates this transformation. In return, our model enjoys compositionality for temporal safety properties.

## 3 THE BASIC MODEL

After some preliminaries (§3.1–3.2), we define the basic model and establish some basic properties (§3.3 and Figure 1). We then explain the model using examples (§3.4–3.7) and discuss program transformations (§3.8). We encourage readers to skip to the examples, coming back as needed.

### 3.1 Preliminaries

The syntax is built from

- a set of *values*  $\mathcal{V}$ , ranged over by  $v, w, \ell, k$ ,
- 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  $X$  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.

$$\mu ::= \text{rlx} \mid \text{ra} \mid \text{sc} \qquad v ::= \text{acq} \mid \text{rel} \mid \text{ar}$$

$$S ::= r := M \mid r := [L]^\mu \mid [L]^\mu := M \mid F^v \mid \text{skip} \mid S_1; S_2 \mid \text{if}(M)\{S_1\}\text{else}\{S_2\} \mid S_1 \parallel S_2$$

*Memory modes*,  $\mu$ , 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*,  $v$ , 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],  $\parallel$  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 §1–4 we require that

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

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

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

The semantics is built from the following.

- a set of *events*  $\mathcal{E}$ , ranged over by  $e, d, c$ , and subsets ranged over by  $E, D, C$ ,
- a set of *logical formulae*  $\Phi$ , ranged over by  $\phi, \psi, \theta$ ,
- 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  $\models$  between formulae, capturing entailment,
- $\models$  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  $\phi$  is a *tautology* if  $\text{tt} \models \phi$ . We say  $\phi$  is *unsatisfiable* if  $\phi \models \text{ff}$ .



### 3.2 Actions in This Paper

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

$$a, b ::= W^\mu xv \mid R^\mu xv \mid F^\nu$$

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  $rlx \sqsubseteq ra \sqsubseteq sc$  and  $rel \sqsubseteq ar$  and  $acq \sqsubseteq ar$ . We write  $(W^{\sqsupset ra})$  to stand for either  $(W^{ra})$  or  $(W^{sc})$ , and similarly for the other actions and modes.

**Definition 3.1.** Actions  $(R)$  are *read* actions. Actions  $(W^{\sqsupset ra})$  and  $(F^{\sqsupset rel})$  are *release* actions.

We say  $a$  *matches*  $b$  if  $a = (Wxv)$  and  $b = (Rxv)$ .

We say  $a$  *blocks*  $b$  if  $a = (Wx)$  and  $b = (Rx)$ , regardless of value.

We say  $a$  *delays*  $b$  if  $a \bowtie_{co} b$  or  $a \bowtie_{sync} b$  or  $a \bowtie_{sc} b$ .

Let  $\bowtie_{co}$  capture write-write, read-write coherence:  $\bowtie_{co} = \{(Wx, Wx), (Rx, Wx), (Wx, Rx)\}$ .

Let  $\bowtie_{sync}$  capture order due to synchronization:  $\bowtie_{sync} = \{(a, W^{\sqsupset ra}), (a, F^{\sqsupset rel}), (R, F^{\sqsupset acq}), (Rx, R^{\sqsupset ra}x), (R^{\sqsupset ra}, a), (F^{\sqsupset acq}, a), (F^{\sqsupset rel}, W), (W^{\sqsupset ra}x, Wx)\}$ .

Let  $\bowtie_{sc}$  capture order due to sc access:  $\bowtie_{sc} = \{(W^{sc}, W^{sc}), (R^{sc}, W^{sc}), (W^{sc}, R^{sc}), (R^{sc}, R^{sc})\}$ .

### 3.3 Definition: Pomsets with Predicate Transformers

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

**Definition 3.2.** A *predicate transformer* is a function  $\tau : \Phi \rightarrow \Phi$  such that

(x1)  $\tau(\text{ff})$  is  $\text{ff}$ , (x3)  $\tau(\psi_1 \vee \psi_2)$  is  $\tau(\psi_1) \vee \tau(\psi_2)$ ,

(x2)  $\tau(\psi_1 \wedge \psi_2)$  is  $\tau(\psi_1) \wedge \tau(\psi_2)$ , (x4) if  $\phi \models \psi$ , then  $\tau(\phi) \models \tau(\psi)$ .

The definition follows [Dijkstra \[1975\]](#). Note that substitutions ( $\tau(\psi) = \psi[M/r]$  and  $\tau(\psi) = \psi[M/x]$ ) and implications on the right ( $\tau(\psi) = \phi \Rightarrow \psi$ ) are predicate transformers.

As discussed in §1, predicate transformers suffice for sequentially consistent models, but not relaxed models, where dependency calculation is crucial. For dependency calculation, we use a *family* of predicate transformers, indexed by sets of events. We use  $\tau^D$  as the predicate transformer applied to any event  $e$  where if  $d \in D$  then  $d < e$ .

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

We write  $\tau$  as an abbreviation of  $\tau^E$ .

**Definition 3.4.** A *pomset with predicate transformers* over  $\mathcal{A}$  is a tuple  $(E, \lambda, \kappa, \tau, \checkmark, \text{rf}, \leq)$  where

(M1)  $E \subseteq \mathcal{E}$  is a set of events,

(M2)  $\lambda : E \rightarrow \mathcal{A}$  defines a *label* for each event,

(M3)  $\kappa : E \rightarrow \Phi$  defines a *precondition* for each event,

(M4)  $\tau : 2^E \rightarrow \Phi \rightarrow \Phi$  is a *family of predicate transformers* over  $E$ ,

(M5)  $\checkmark : \Phi$  defines a *termination condition*,

(M6)  $\text{rf} : E \rightarrow E$  is an injective relation capturing *reads-from* such that

(M6a) if  $d \xrightarrow{\text{rf}} e$  then  $\lambda(d)$  matches  $\lambda(e)$ ,

(M7)  $\leq : E \times E$ , is a partial order capturing *causality*, such that

(M7a) if  $d \xrightarrow{\text{rf}} e$  and  $\lambda(c)$  blocks  $\lambda(e)$  then either  $c \leq d$  or  $e \leq c$ .

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  $d \xrightarrow{\text{rf}} e$ .

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  $P \in \text{SKIP}$  then  $E = \emptyset$  and  $\tau^D(\psi) \models \psi$ .

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

(p1)  $E = (E_1 \uplus E_2)$ ,

(p2)  $\lambda = (\lambda_1 \cup \lambda_2)$ ,

(p3a) if  $e \in E_1$  then  $\kappa(e) \models \kappa_1(e)$ ,

(p3b) if  $e \in E_2$  then  $\kappa(e) \models \kappa_2(e)$ ,

(p4)  $\tau^D(\psi) \models \tau_1^D(\psi)$ ,

(p5)  $\checkmark \models \checkmark_1 \wedge \checkmark_2$ ,

(p6)  $\text{rf}$  extends  $\text{rf}_1$  and  $\text{rf}_2$ ,

(p7a)  $\leq$  extends  $\leq_1$  and  $\leq_2$ ,

(p7b) if  $d \in E_1$ ,  $e \in E_2$  and  $d \xrightarrow{\text{rf}} e$  then  $d \leq e$ .

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

let  $\kappa'_2(e) = \tau_1^{1^e}(\kappa_2(e))$ , 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  $\kappa(e) \models \kappa_1(e)$ ,

(s3b) if  $e \in E_2 \setminus E_1$  then  $\kappa(e) \models \kappa'_2(e)$ ,

(s3c) if  $e \in E_1 \cap E_2$  then  $\kappa(e) \models \kappa_1(e) \vee \kappa'_2(e)$ ,

(s3d) if  $\lambda_2(e)$  is a release then  $\kappa(e) \models \checkmark_1$ ,

(s4)  $\tau^D(\psi) \models \tau_1^D(\tau_2^D(\psi))$ ,

(s5)  $\checkmark \models \checkmark_1 \wedge \tau_1(\checkmark_2)$ ,

(s6)  $\text{rf}$  extends  $\text{rf}_1$  and  $\text{rf}_2$ ,

(s7a)  $\leq$  extends  $\leq_1$  and  $\leq_2$ ,

(s7b) if  $d \in E_1$ ,  $e \in E_2$  and  $d \xrightarrow{\text{rf}} e$  then  $d \leq e$ ,

(s7c) if  $\lambda_1(d)$  delays  $\lambda_2(e)$  then  $d \leq e$ .

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

(c1)  $E = (E_1 \cup E_2)$ ,

(c2)  $\lambda = (\lambda_1 \cup \lambda_2)$ ,

(c3a) if  $e \in E_1 \setminus E_2$  then  $\kappa(e) \models \phi \wedge \kappa_1(e)$ ,

(c3b) if  $e \in E_2 \setminus E_1$  then  $\kappa(e) \models \neg\phi \wedge \kappa_2(e)$ ,

(c3c) if  $e \in E_1 \cap E_2$

then  $\kappa(e) \models (\phi \wedge \kappa_1(e)) \vee (\neg\phi \wedge \kappa_2(e))$ ,

(c4)  $\tau^D(\psi) \models (\phi \wedge \tau_1^D(\psi)) \vee (\neg\phi \wedge \tau_2^D(\psi))$ ,

(c5)  $\checkmark \models (\phi \wedge \checkmark_1) \vee (\neg\phi \wedge \checkmark_2)$ .

(c6a)  $\text{rf}$  extends  $\text{rf}_1$  and  $\text{rf}_2$ ,

(c6b)  $\text{rf} \subseteq (\text{rf}_1 \cup \text{rf}_2)$ ,

(c7a)  $\leq$  extends  $\leq_1$  and  $\leq_2$ ,

(c7b)  $\leq \subseteq (\leq_1 \cup \leq_2)$ .

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

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

(r1) if  $d, e \in E$  then  $d = e$ ,

(r2)  $\lambda(e) = R^\mu xv$ ,

(r4a) if  $(E \cap D) \neq \emptyset$  then  $\tau^D(\psi) \models v=r \Rightarrow \psi$ ,

(r4b) if  $E \neq \emptyset$  and  $(E \cap D) = \emptyset$  then

$\tau^D(\psi) \models (v=r \vee x=r) \Rightarrow \psi$ .

(r4c) if  $E = \emptyset$  then  $\tau^D(\psi) \models \psi$ .

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

(w1) if  $d, e \in E$  then  $d = e$ ,

(w2)  $\lambda(e) = W^\mu xv$ ,

(w3)  $\kappa(e) \models M=v$ ,

(w4)  $\tau^D(\psi) \models \psi[M/x]$ ,

(w5a) if  $E \neq \emptyset$  then  $\checkmark \models M=v$ ,

(w5b) if  $E = \emptyset$  then  $\checkmark \models \text{ff}$ .

If  $P \in FENCE(\mu)$  then

(f1) if  $d, e \in E$  then  $d = e$ ,

(f2)  $\lambda(e) = F^\mu$ ,

(f4)  $\tau^D(\psi) \models \psi$ ,

(f5) if  $E = \emptyset$  then  $\checkmark \models \text{ff}$ .

$\llbracket r := M \rrbracket_1 = LET(r, M)$

$\llbracket \text{skip} \rrbracket_1 = \text{SKIP}$

$\llbracket r := x^\mu \rrbracket_1 = READ(r, x, \mu)$

$\llbracket S_1 \parallel S_2 \rrbracket_1 = \llbracket S_1 \rrbracket_1 \parallel \llbracket S_2 \rrbracket_1$

$\llbracket x^\mu := M \rrbracket_1 = WRITE(x, M, \mu)$

$\llbracket S_1; S_2 \rrbracket_1 = \llbracket S_1 \rrbracket_1; \llbracket S_2 \rrbracket_1$

$\llbracket F^\nu \rrbracket_1 = FENCE(\nu)$

$\llbracket \text{if}(M)\{S_1\}\text{else}\{S_2\} \rrbracket_1 = IF(M \neq 0, \llbracket S_1 \rrbracket_1, \llbracket S_2 \rrbracket_1)$

Fig. 1. Semantics of programs



We give the semantics of programs  $\llbracket \cdot \rrbracket_1$  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  $\kappa'_2$  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 3.5.  $(\mathcal{P}_1; \mathcal{P}_2); \mathcal{P}_3 = \mathcal{P}_1; (\mathcal{P}_2; \mathcal{P}_3)$  and  $\mathcal{P}; \text{skip} = \mathcal{P} = \text{skip}; \mathcal{P}$ .  
 $(\mathcal{P}_1 \parallel \mathcal{P}_2) \parallel \mathcal{P}_3 = \mathcal{P}_1 \parallel (\mathcal{P}_2 \parallel \mathcal{P}_3)$  and  $\mathcal{P} \parallel \text{skip} = \mathcal{P}$ .

PROOF. Straightforward calculation. Associativity of  $;$  requires disjunction closure (x3).  $\square$

Definition 3.6.  $P_2$  is an *augment* of  $P_1$  if

- |                                     |   |   |                                 |
|-------------------------------------|---|---|---------------------------------|
| (1) $E_2 = E_1$ ,                   | (3) $\kappa_2(e) \models \kappa_1(e)$ , | (5) $\checkmark_2 \models \checkmark_1$ , | (7) $\leq_2 \supseteq \leq_1$ . |
| (2) $\lambda_2(e) = \lambda_1(e)$ , | (4) $\tau_2^D(e) \models \tau_1^D(e)$ , | (6) $\text{rf}_2 = \text{rf}_1$ ,         |                                 |

LEMMA 3.7. If  $P_1 \in \llbracket S \rrbracket_1$  and  $P_2$  augments  $P_1$  then  $P_2 \in \llbracket S \rrbracket_1$ .

PROOF. Induction on the definition of  $\llbracket \cdot \rrbracket_1$ .  $\square$

### 3.4 Discussion: Pomsets

We first explain the core of model, ignoring the logic (rules 3–5). We defer discussion of *IF* to §3.6. We defer the discussion of *LET* to §3.7.

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  $\llbracket x := 1 \rrbracket_1$  can write any value  $v$ ; the fact that  $v$  must be 1 is captured in the logic (see §3.5).

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 §3.5), and s7c may require order (see example PUB, below).

Note that reads-from implies order.

LEMMA 3.8. For any  $P$  in the range of  $\llbracket \cdot \rrbracket_1$ ,  $d \xrightarrow{\text{rf}} e$  implies  $d \leq e$ .

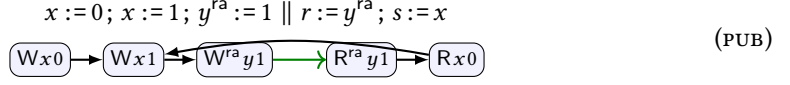
PROOF. Induction on the definition of  $\llbracket \cdot \rrbracket_1$ , using p7b and s7b.  $\square$

In top-level pomsets, every read must have a matching write in *rf* (r2b). Together with m6a and m7a, the lemma guarantees that reads are *fulfilled* at top-level, as in [Jagadeesan et al. 2020, §2.7].<sup>2</sup>

From Definition 3.1, recall that  $a$  *delays*  $b$  if  $a \models_{\text{co}} b$  or  $a \models_{\text{sync}} b$  or  $a \models_{\text{sc}} b$ . s7c guarantees that sequential order is enforced between conflicting accesses of the same location ( $\models_{\text{co}}$ ), into a release and out of an acquire ( $\models_{\text{sync}}$ ) and between SC accesses ( $\models_{\text{sc}}$ ). Combined with the fulfillment

<sup>2</sup>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 §4.3, where we introduce a variant semantics  $\llbracket \cdot \rrbracket_2^f$  for which Lemma 3.8 fails.

requirements (M6a, M7a and Lemma 3.8), these ensure coherence, publication, subscription and other idioms. For example, consider the following:



The execution is allowed due to the evident cycle. All of the order shown is required at top-level: The intra-thread order comes from s7c:  $(Wx0) \rightarrow (Wx1)$  is required by  $\approx_{co}$ .  $(Wx1) \rightarrow (W^{ra}y1)$  and  $(R^{ra}y1) \rightarrow (Rx0)$  are required by  $\approx_{sync}$ . The cross-thread order is required by fulfillment: t2b requires that all top-level reads are in the image of  $\xrightarrow{rf}$ . M6a ensures that  $(W^{ra}y1) \xrightarrow{rf} (R^{ra}y1)$ , and s7b subsequently ensures that  $(W^{ra}y1) \rightarrow (R^{ra}y1)$ . The *antidependency*  $(Rx0) \rightarrow (Wx1)$  is required by M7a. (Alternatively, we could have  $(Wx1) \rightarrow (Wx0)$ , 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, to support common subexpression elimination, and stronger than Java, to support local reasoning about data races. See [Jagadeesan et al. 2020, §3.1] for a discussion.

### 3.5 Discussion: Termination

In top-level pomsets, T1 requires that  $\checkmark$  is a tautology, capturing termination. Roughly, the structural rules take  $\checkmark$  to be  $\checkmark_1 \wedge \checkmark_2$ . The interesting rules are *READ*, *FENCE*, and *WRITE*.

In *READ*, there is no restriction on  $\checkmark$ . From this, it is easy to see that  $\llbracket r := x \rrbracket_1 \supseteq \llbracket \text{skip} \rrbracket_1$  is a valid refinement.

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  $P \in \llbracket x := 1 \rrbracket_1$ . As we noted above,  $P$  can include  $(Wxv)$ , 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  $\llbracket x := 1; x := 1 \rrbracket_1$  includes the singleton pomset  $(Wx1)$ . From this it is easy to see that  $\llbracket x := 1; x := 1 \rrbracket_1 \supseteq \llbracket x := 1 \rrbracket_1$  is a valid refinement. It is equally obvious that  $\llbracket x := 1 \rrbracket \not\supseteq \llbracket x := 1; x := 1 \rrbracket$  is not a valid refinement, since the latter includes a two-element pomset, but the former does not.<sup>3</sup>

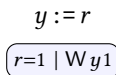
### 3.6 Discussion: Preconditions

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

The definition of *extends* ensures that no new order is introduced between events in  $E_1 \cap E_2$ .<sup>4</sup>

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

*Example 3.9.* A simple example of a data dependency is a pomset  $P \in \llbracket r := x; y := r \rrbracket$ , for which there must be an  $v \in \mathcal{V}$  and  $P' \in \llbracket y := r \rrbracket$  such as the following, where  $v = 1$ :



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

<sup>4</sup>This ensures that  $\llbracket \text{if}(b)\{r := x \parallel x := 1\} \text{ else } \{r := x; x := 1\} \rrbracket_2^{rf}$  does not include  $(Rx1) \xleftarrow{rf} (Wx1)$ , taking  $\xrightarrow{rf}$  from the left and  $\leq$  from the right.

The value chosen for the read may be different from that chosen for the write:

$$r := x; y := r$$

$$\boxed{Rx0} \rightarrow \boxed{r=0 \Rightarrow r=1 \mid Wy1}$$

In this case, the pomset's preconditions depend on a bound register, so cannot contribute to a top-level pomset.

If the values chosen for read and write are compatible, then we have two cases: the independent case, which again cannot be part of a top-level pomset,

$$r := x; y := r$$

$$\boxed{Rx1} \quad \boxed{r=1 \mid Wy1}$$

and the dependent case:

$$\boxed{Rx1} \rightarrow \boxed{r=1 \Rightarrow r=1 \mid Wy1}$$

Since  $r=1 \Rightarrow r=1$  is a tautology, this can be part of a top-level pomset.

*Example 3.10.* Control dependencies are similar, for example for any  $P \in \llbracket r := x; \text{if}(r)\{y := 1\} \rrbracket$ , there must be an  $v \in \mathcal{V}$  and  $P' \in \llbracket \text{if}(r)\{y := 1\} \rrbracket$  such as:

$$\text{if}(r)\{y := 1\}$$

$$\boxed{r \neq 0 \mid Wy1}$$

The rest of the reasoning is the same as Example 3.9.

*Example 3.11.* A simple example of an independency is a pomset  $P \in \llbracket r := x; y := 1 \rrbracket$ , for which there must be:

$$y := 1$$

$$\boxed{1=1 \mid Wy1}$$

In this case it doesn't matter what value the read chooses:

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

$$\boxed{Rx0} \quad \boxed{1=1 \mid Wy1}$$

*Example 3.12.* Consider  $P \in \llbracket \text{if}(r=1)\{y := r\} \text{ else } \{y := 1\} \rrbracket$ , so there must be  $P_1 \in \llbracket y := r \rrbracket$ , and  $P_2 \in \llbracket y := 1 \rrbracket$ , such as:

$$y := r \qquad y := 1$$

$$\boxed{r=1 \mid Wy1} \qquad \boxed{1=1 \mid Wy1}$$

Since there is no requirement for disjointness in the semantics of conditionals, we can consider the case where the event *coalesces* from the two pomsets, in which case:

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

$$\boxed{(r=1 \Rightarrow r=1) \wedge (r \neq 1 \Rightarrow 1=1) \mid Wy1}$$

Here, the precondition is a tautology, independent of  $r$ .

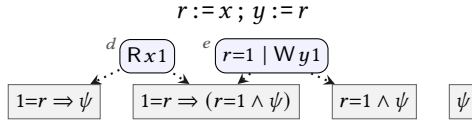
### 3.7 Discussion: Predicate Transformers

the treatment of sequential composition is new. This uses the usual rule for composition of predicate transformers (but preserving the indexing set). For the pomset, we take the union of their events, preserving actions, but crucially in cases [s3b](#) and [s3c](#) we apply a predicate transformer  $\tau_1^C$  from the left-hand side to a precondition  $\kappa_2(e)$  from the right-hand side to build the precondition  $\kappa'_2(e)$ . The indexing set  $C$  for the predicate transformer is  $\{c \mid c < e\}$ , so can depend on the causal order.

*Example 3.13.* For read to write dependency, consider:

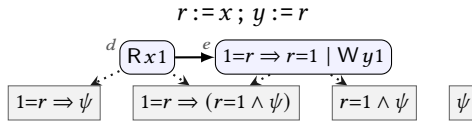


Putting these together without order, we calculate the precondition  $\kappa(e)$  as  $\tau_1^C(\kappa_2(e))$ , where  $C$  is  $\{c \mid c < e\}$ , which is  $\emptyset$ . Since  $\tau_1^\emptyset(\psi)$  is  $\psi$ , this gives that  $\kappa(e)$  is  $\kappa_2(e)$ , which is  $r=1$ . This gives the pomset with predicate transformers:



This pomset's preconditions depend on a bound register, so cannot contribute to a top-level pomset.

Putting them together with order, we calculate the precondition  $\kappa(e)$  as  $\tau_1^C(\kappa_2(e))$ , where  $C$  is  $\{c \mid c < e\}$ , which is  $\{d\}$ . Since  $\tau_1^{\{d\}}(\psi)$  is  $(1=r \Rightarrow \psi)$ , this gives that  $\kappa(e)$  is  $(1=r \Rightarrow \kappa_2(e))$ , which is  $(1=r \Rightarrow r=1)$ . This gives the pomset with predicate transformers:

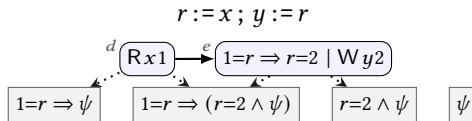


This pomset's preconditions do not depend on a bound register, so can contribute to a top-level pomset.

*Example 3.14.* If the read and write choose different values:



Putting these together with order, we have the following, which cannot be part of a top-level pomset:



*Example 3.15.* ?? includes  $M=v$  to ensure that spurious merges do not go undetected. Consider the following.



Merging the actions, since  $2=1$  is unsatisfiable, we have:

$$x := 1; x := 2$$

$$\boxed{Wx1} \cdots \rightarrow \boxed{\text{ff}} \quad \boxed{\psi}$$

This pomset cannot be part of a top-level pomset, since  $\tau^E(s) = \text{ff}$  for every quiescence symbol  $s$ . This is what we would hope: that the program  $x := 1; x := 2$  should only be top-level if there is a  $(Wx2)$  event.

*Example 3.16.* The predicate transformer we have chosen for  $??$  is different from the one used traditionally, which is written using substitution. Substitution is also used in [Jagadeesan et al. 2020]. Attempting to write the predicate transformers in this style we have:

$$??) \tau^D(\psi) \models \psi[v/r],$$

$$??) \tau^C(\psi) \models (\forall r)\psi.$$

This phrasing of  $??$  says that  $\psi$  must be independent of  $r$  in order to appear in a top-level pomset. This choice for  $??$  is forced by Definition 3.3, 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:

$$\begin{array}{cccc} r := y & x := !r & x := !!r & x := 0 \\ \boxed{Ry1} & \boxed{r=0 \mid Wx1} & \boxed{r \neq 0 \mid Wx1} & \boxed{Wx0} \end{array}$$

Associating to the right and merging:

$$\begin{array}{cc} r := y & x := !r; x := !!r; x := 0 \\ \boxed{Ry1} & \boxed{r=0 \vee r \neq 0 \mid Wx1} \rightarrow \boxed{Wx0} \end{array}$$

The precondition of  $(Wx1)$  is a tautology, thus we have:

$$\begin{array}{c} r := y; x := !r; x := !!r; x := 0 \\ \boxed{Ry1} \quad \boxed{Wx1} \rightarrow \boxed{Wx0} \end{array}$$

If, instead, we associate to the left:

$$\begin{array}{cc} r := y; x := !r & x := !!r; x := 0 \\ \boxed{Ry1} \quad \boxed{1=0 \mid Wx1} & \boxed{r \neq 0 \mid Wx1} \rightarrow \boxed{Wx0} \end{array}$$

Sequencing and merging:

$$\begin{array}{c} r := y; x := !r; x := !!r; x := 0 \\ \boxed{Ry1} \quad \boxed{1=0 \vee r \neq 0 \mid Wx1} \rightarrow \boxed{Wx0} \end{array}$$

In this case, the precondition of  $(Wx1)$  is not a tautology, forcing a dependency  $(Ry1) \rightarrow (Wx1)$ .

Our solution is to Skolemize. We have proven associativity of Definition ?? in Agda. The proof requires that predicate transformers distribute through disjunction (Definition 3.2). The attempt to define predicate transformers using substitution fails for  $??$  because the predicate transformer  $\tau(\psi) = (\forall r)\psi$  does not distribute through disjunction:  $\tau(\psi_1 \vee \psi_2) = (\forall r)(\psi_1 \vee \psi_2) \neq ((\forall r)(\psi_1)) \vee ((\forall r)(\psi_2)) = \tau(\psi_1) \vee \tau(\psi_2)$ .

### 3.8 Discussion: Valid Transformations

When  $\llbracket S \rrbracket \supseteq \llbracket S' \rrbracket$ , we say that  $S'$  is a *valid transformation* of  $S$ . In this subsection, we show the validity of specific optimizations. Let  $\text{id}(S)$  be the set of locations and registers that occur in  $S$ .

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

$$\llbracket r := x; s := y; S \rrbracket = \llbracket s := y; r := x; S \rrbracket \quad \text{if } r \neq s \quad (\text{RR})$$

$$\llbracket x := M; y := N; S \rrbracket = \llbracket y := N; x := M; S \rrbracket \quad \text{if } x \neq y \quad (\text{WW})$$

$$\llbracket x := M; s := y; S \rrbracket = \llbracket s := y; x := M; S \rrbracket \quad \text{if } x \neq y \text{ and } s \notin \text{id}(M) \quad (\text{RW})$$

?? imposes no order between events in **RR**–**RW**. Using augmentation closure, ?? also validates roach-motel reorderings [Sevčík 2008]. For example, on read/write pairs:

$$\llbracket x^\mu := M; s := y; S \rrbracket \supseteq \llbracket s := y; x^\mu := M; S \rrbracket \quad \text{if } x \neq y \text{ and } s \notin \text{id}(M) \quad (\text{ROACH1})$$

$$\llbracket x := M; s := y^\mu; S \rrbracket \supseteq \llbracket s := y^\mu; x := M; S \rrbracket \quad \text{if } x \neq y \text{ and } s \notin \text{id}(M) \quad (\text{ROACH2})$$

Redundant load elimination (**RL**) follows from ??, taking  $d \in E$ , regardless of the access mode:

$$\llbracket r := x^\mu; s := x^\mu; S \rrbracket \supseteq \llbracket r := x^\mu; s := r; S \rrbracket \quad (\text{RL})$$

Many laws hold for the conditional, such as dead code elimination (**DC**) and code lifting (**CL**):

$$\llbracket \text{if}(M)\{S_1\}\text{else}\{S_2\} \rrbracket = \llbracket S_1 \rrbracket \quad \text{if } M \text{ is a tautology} \quad (\text{DC})$$

$$\llbracket \text{if}(M)\{S\}\text{else}\{S\} \rrbracket \supseteq \llbracket S \rrbracket \quad (\text{CL})$$

Code lifting also applies to program fragments inside a conditional. For example:

$$\llbracket \text{if}(M)\{x := N; S_1\}\text{else}\{x := N; S_2\} \rrbracket \supseteq \llbracket x := N; \text{if}(M)\{S_1\}\text{else}\{S_2\} \rrbracket$$

We discuss the inverse of **CL** in §??.

## 4 ARM

For simplicity, we restrict to top level parallel composition and ignore fences<sup>5</sup>.

### 4.1 Arm executions

*Definition 4.1.* An *Arm8 execution graph*,  $G$ , is tuple  $(E, \lambda, \text{poloc}, \text{lob})$  such that

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

The definition of  $\text{lob}$  is complex. Comparing with our definition of sequential composition, it is sufficient to note that  $\text{lob}$  includes

- (L1) read-write dependencies, required by **s3**,
- (L2) synchronization delay of  $\preceq_{\text{sync}}$ , required by **s7c**,
- (L3) sc access delay of  $\preceq_{\text{sc}}$ , required by **s7c**,
- (L4) write-write and read-to-write coherence delay of  $\preceq_{\text{co}}$ , required by **s7c**,

and that  $\text{lob}$  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  $\preceq_{\text{co}}$ , required by **s7c**.

<sup>5</sup>Fences are not actions in Arm8, which complicates the theorem statements.

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

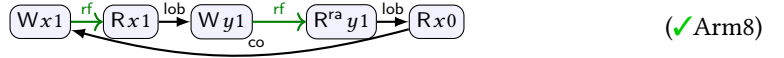
- (A5)  $\text{co} : E \times E$ , is a per-location total order on writes, capturing *coherence*,
- (A6)  $\text{rf} : E \times E$ , is a surjective and injective relation on reads, capturing *reads-from*, such that
  - (A6a) if  $d \xrightarrow{\text{rf}} e$  then  $\lambda(d)$  matches  $\lambda(e)$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

Execution  $G$  is  $(\text{co}, \text{rf}, \text{cb})$ -valid under *External Consistency* (EC) if

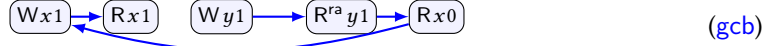
- (A5) and (A6), as for EGC,
- (A8)  $\text{cb} \supseteq (\text{co} \cup \text{lob})$  is a linear order such that if  $d \xrightarrow{\text{rf}} e$  then either
  - (A8a)  $d \xrightarrow{\text{cb}} e$  and if  $\lambda(c)$  blocks  $\lambda(e)$  then either  $c \xrightarrow{\text{cb}} d$  or  $e \xrightarrow{\text{cb}}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

[Alglave et al. 2021] explain EGC and EC using the following example.<sup>6</sup>

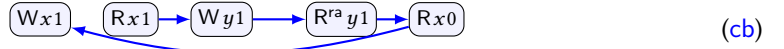
$x := 1; r := x; y := r \parallel 1 := y^{\text{ra}}; s := x$



EGC drops lob-order in the first thread using 4.4, since  $(Wx1)$  is not lob-ordered before  $(Wy1)$ .



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



## 4.2 Arm Compilation 1

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

We do not distinguish control dependencies from address dependencies, and therefore L5 forces us to drop all dependencies between reads. To achieve this, we modify the definition of  $\kappa'_2$  in Figure 1.

**Definition 4.3.** Let  $\llbracket \cdot \rrbracket_2$  be defined as in Figure 1, replacing the definition of  $\kappa'_2$  with:

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

**THEOREM 4.4.** Suppose  $G_1$  is  $(\text{co}_1, \text{rf}_1, \text{gcb}_1)$ -valid for  $S$  under the suboptimal lowering that maps non-relaxed reads to (dmb.sy; ldar). Then there is a top-level pomset  $P_2 \in \llbracket S \rrbracket_2$  such that  $E_2 = E_1$ ,  $\lambda_2 = \lambda_1$ ,  $\text{rf}_2 = \text{rf}_1$ , and  $\leq_2 = \text{gcb}_1$ .

**PROOF.** First, we establish some lemmas about Arm8.

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

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

<sup>6</sup>We have changed an address dependency in the first thread to a data dependency.

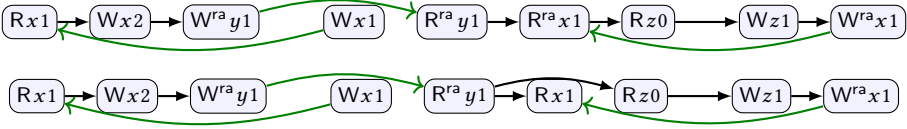


LEMMA 4.6. Suppose  $G$  is  $(\text{co}, \text{rf}, \text{gcb})$ -valid and  $c \xrightarrow{\text{poloc}} e$ , where  $\lambda(c)$  blocks  $\lambda(e)$ . Then  $c \xrightarrow{\text{gcb}} e$ .

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

We show that all the order required in the pomset is also required by Arm8. m7a holds since  $\text{cb}_1$  is consistent with  $\text{co}_1$  and  $\text{fr}_1$ . As noted above,  $\text{lob}$  includes the order required by s3 and s7c. We need only show that the order removed from 4.4 can also be removed from the pomset. In order for to remove order from  $e$  to  $c$ , we must have  $d \xrightarrow{\text{rf}} e$  and  $d \xrightarrow{\text{poloc}} e$  but not  $d \xrightarrow{\text{lob}} c$ . Because of our suboptimal lowering, it must be that  $e$  is a relaxed read; otherwise the dmb.sy would require  $d \xrightarrow{\text{lob}} c$ . Thus we know that s7c does not require order from  $e$  to  $c$ . By chaining r4b and w4, any dependence on the read can be satisfied without introducing order in s3.  $\square$

Downgrading messes up publication:

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


### 4.3 Arm Compilation 2

Definition 4.7. Let  $\llbracket \cdot \rrbracket_2^{\text{rf}}$  be defined as for  $\llbracket \cdot \rrbracket_2$  in Definition 4.3 and Figure 1, changing s7b and s7c:

(s7b<sup>rf</sup>) if  $\lambda_1(c)$  blocks  $\lambda_2(e)$  then  $d \xrightarrow{\text{rf}} e$  implies  $c \leq d$ ,

(s7c<sup>rf</sup>) if  $\lambda_1(d)$  delays'  $\lambda_2(e)$  then  $d \leq e$ ,

where  $\text{delays}'$  replaces  $\preceq_{\text{co}}$  in Definition 3.1 of  $\text{delays}$  by  $\preceq_{\text{lws}} = \{(Wx, Wx), (Rx, Wx)\}$ .

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

Note that Lemma 3.8 fails for  $\llbracket \cdot \rrbracket_2^{\text{rf}}$ , since  $d \xrightarrow{\text{rf}} e$  may not imply  $d \leq e$  when  $d$  and  $e$  come from different sides of a sequential composition. This means that  $\text{rf}$  must be verified during pomset construction, rather than post-hoc. If one wants a post-hoc verification technique for  $\text{rf}$ , it is possible to include program order (po) in the pomset.

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

LEMMA 4.9.  $P$  is top-level iff  $d \xrightarrow{\text{rf}} e$  implies either

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

THEOREM 4.10. Suppose  $G_1$  is EC-valid for  $S$  via  $(\text{co}_1, \text{rf}_1, \text{cb}_1)$  and that  $\text{cb}_1 \supseteq \text{fr}_1$ . Then there is a top-level pomset  $P_2 \in \llbracket S \rrbracket_2^{\text{rf}}$  such that  $E_2 = E_1$ ,  $\lambda_2 = \lambda_1$ ,  $\text{rf}_2 = \text{rf}_1$ , and  $\leq_2 = \text{cb}_1$ .

PROOF. We show that all the order required in the pomset is also required by Arm8. m7a holds since  $\text{cb}_1$  is consistent with  $\text{co}_1$  and  $\text{fr}_1$ . s7b<sup>rf</sup> follows from A8b. As noted above,  $\text{lob}$  includes the order required by s3 and s7c<sup>rf</sup>.  $\square$

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

LEMMA 4.11. Suppose  $G$  is EC-valid via  $(\text{co}, \text{rf}, \text{cb})$ . Then there a permutation  $\text{cb}'$  of  $\text{cb}$  such that  $G$  is EC-valid via  $(\text{co}, \text{rf}, \text{cb}')$  and  $\text{cb}' \supseteq \text{fr}$ , where  $\text{fr}$  is defined in A6b.

PROOF. We show that any **cb** order that contradicts **fr** is incidental.

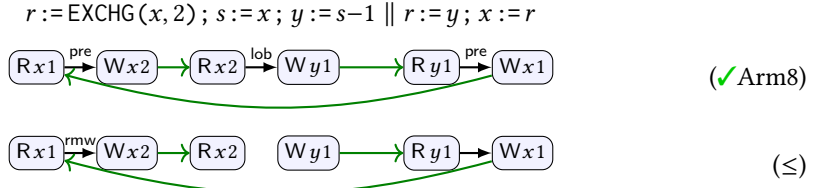
By definition of **fr**,  $e \xrightarrow{\text{rf}} d \xrightarrow{\text{co}} c$ , for some  $d$ . Since **cb**  $\supseteq$  **co**, we know that  $d \xrightarrow{\text{co}} c$ .

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

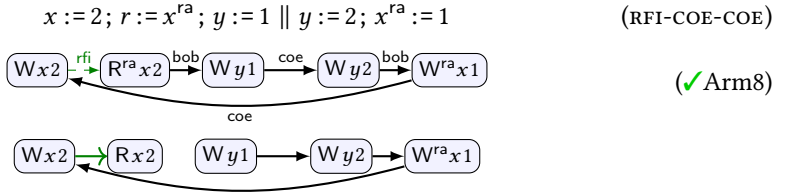
Suppose **A8b** applies to  $d \xrightarrow{\text{rf}} e$  and  $c$  is from a different thread. Because it is a different thread, we cannot have  $e \xrightarrow{\text{lob}} c$ , and thus the order in **cb** is incidental.

Suppose **A8b** applies to  $d \xrightarrow{\text{rf}} e$  and  $c$  is from the same thread. Since  $c \xrightarrow{\text{co}} d$ , it cannot be that  $c \xrightarrow{\text{poloc}} d$ , using **A6b**. It also cannot be that  $d \xrightarrow{\text{poloc}} c \xrightarrow{\text{poloc}} e$ . It must be that  $e \xrightarrow{\text{poloc}} c$ . By **A4a**, we cannot have  $e \xrightarrow{\text{lob}} c$ , and thus the order in **cb** is incidental.  $\square$

Bad example:



Anton example 1 [rfi-coe-coe]



## 5 ADDITIONAL FEATURES

### 5.1 Register Recycling

*Definition 5.1.* Let  $\llbracket \cdot \rrbracket_3$  be defined as for  $\llbracket \cdot \rrbracket_2$  in Definition 4.3 and Figure 1, changing **r4** of *READ*:

(**r4a**) if  $(E \cap D) \neq \emptyset$  then  $\tau^D(\psi) \models v = s_e \Rightarrow \psi[s_e/r]$ ,

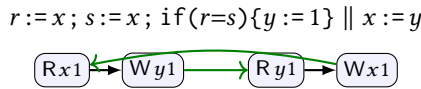
(**r4b**) if  $E \neq \emptyset$  and  $(E \cap D) = \emptyset$  then  $\tau^D(\psi) \models (v = s_e \vee x = s_e) \Rightarrow \psi[s_e/r]$ .

(**r4c**) ( $\forall s$ ) if  $E = \emptyset$  then  $\tau^D(\psi) \models \psi[s/r]$ .

Similarly, let  $\llbracket \cdot \rrbracket_3^{\text{rf}}$  be defined as for  $\llbracket \cdot \rrbracket_2^{\text{rf}}$  in Definition 4.7, with this definition of *READ*.

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

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



This execution is not allowed under Definition ??, since the precondition of ( $Wy1$ ) in the independent case is

$$(r=1 \vee r=x) \Rightarrow (s=1 \vee s=r) \Rightarrow (r=s),$$

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



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

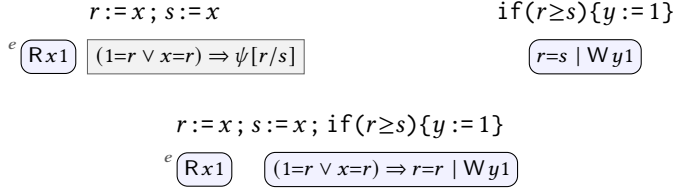
??)  $\tau^D(\psi)$  implies  $v=s_e \Rightarrow \psi[s_e/r]$ ,

??)  $(\forall s) \tau^C(\psi)$  implies  $\psi[s/r]$ .

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



Coalescing and composing:



The precondition of (Wy1) is a tautology, as required.

## 5.2 If-Closure

**Definition 5.5.** Let  $\llbracket \cdot \rrbracket_4$  be defined as for  $\llbracket \cdot \rrbracket_2$  in Definition 4.3 and Figure 1, changing *WRITE* and *READ*.

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

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

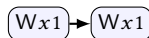
If  $P \in \text{READ}(r, x, \mu)$  then  $(\exists v : E \rightarrow \mathcal{V}) (\exists \theta : E \rightarrow \Phi)$

- (r1) if  $\theta_d \wedge \theta_e$  is satisfiable then  $d = e$ ,
- (r2)  $\lambda(e) = R^\mu x v_e$ ,
- (r3)  $\kappa(e) \models \theta_e$ ,
- (r4a)  $(\forall e \in E \cap D) \tau^D(\psi) \models \theta_e \Rightarrow v_e = s_e \Rightarrow \psi[s_e/r]$ ,
- (r4b)  $(\forall e \in E \setminus D) \tau^D(\psi) \models \theta_e \Rightarrow (v_e = s_e \vee x = s_e) \Rightarrow \psi[s_e/r]$ ,
- (r4c)  $(\forall s) \tau^D(\psi) \models (\bigwedge_{e \in E} \neg \theta_e) \Rightarrow \psi[s/r]$ .

Similarly, let  $\llbracket \cdot \rrbracket_4^{\text{rf}}$  be defined as for  $\llbracket \cdot \rrbracket_2^{\text{rf}}$  in Definition 4.7, with these definitions of *WRITE* and *READ*.

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

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



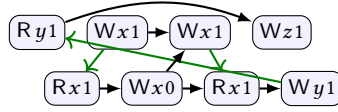
However, if  $S = (\text{if}(\neg M)\{x := 1\}; \text{if}(M)\{x := 1\})$ , then it *does* allow the execution. Looking at the initial program:

$\text{if}(M)\{x := 1\}$	$x := 1$	$\text{if}(\neg M)\{x := 1\}$
$\boxed{M \mid Wx1}$	$\boxed{Wx1}$	$\boxed{\neg M \mid Wx1}$

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

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

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

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


**Definition 5.7 (ALPHA/IF).** Update Definition ?? to:

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

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

??)  $\lambda(e) = Wxv_e$ ,

??)  $\kappa(e)$  implies  $\theta_e \wedge M=v$ ,

??)  $(\forall e \in E \cap D) \tau^D(\psi)$  implies  $\theta_e \Rightarrow (\psi \wedge M=v)$ ,

??)  $\tau^C(\psi)$  implies  $(\nexists e \in E \cap C \mid \theta_e) \Rightarrow \psi$ ,

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

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

??)  $\lambda(e) = Rxv_e$ ,

??)  $\kappa(e)$  implies  $\theta_e$ .

??)  $(\forall e \in E \cap D) \tau^D(\psi)$  implies  $\theta_e \Rightarrow v_e=s_e \Rightarrow \psi[s_e/r]$ ,

??)  $(\forall s) \tau^C(\psi)$  implies  $(\nexists e \in E \mid \theta_e) \Rightarrow \psi[s/r]$ .

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

$\text{if}(M)\{x := 1\}$	$x := 1$	$\text{if}(\neg M)\{x := 1\}$
$\overset{d}{\boxed{M \mid Wx1}}$	$\overset{d}{\boxed{\neg M \mid Wx1}} \overset{e}{\boxed{M \mid Wx1}}$	$\overset{e}{\boxed{\neg M \mid Wx1}}$

Coalescing events gives the desired result.

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

### 5.3 Address Calculation

**Definition 5.9.** Let  $\llbracket \cdot \rrbracket_5$  be defined as for  $\llbracket \cdot \rrbracket_2$  in Definition 4.3 and Figure 1, changing *WRITE* and *READ*.

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

(w1) if  $\theta_d \wedge \theta_e$  is satisfiable then  $d = e$ ,

(w4b)  $(\forall k)$

(w2)  $\lambda(e) = W^\mu[\ell]v_e$ ,

$\tau^D(\psi) \models (\bigwedge_{e \in E} \neg \theta_e) \Rightarrow (L=k) \Rightarrow \psi[M/[k]]$

(w3)  $\kappa(e) \models \theta_e \wedge L=\ell_e \wedge M=v_e$ ,

(w5a)  $\checkmark \models \theta_e \Rightarrow L=\ell_e \wedge M=v_e$ ,

(w4a)  $\tau^D(\psi) \models \theta_e \Rightarrow \psi[M/[\ell]]$ ,

(w5b)  $\checkmark \models \bigvee_{e \in E} \theta_e$ .

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

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

(R2)  $\lambda(e) = R^\mu[\ell]v_e$

(R3)  $\kappa(e) \models \theta_e \wedge L = \ell_e$ ,

(R4a)  $(\forall e \in E \cap D) \tau^D(\psi) \models \theta_e \Rightarrow (L = \ell_e \Rightarrow v_e = s_e) \Rightarrow \psi[s_e/r]$ ,

(R4b)  $(\forall e \in E \setminus D) \tau^D(\psi) \models \theta_e \Rightarrow ((L = \ell_e \Rightarrow v_e = s_e) \vee (L = \ell_e \Rightarrow [\ell] = s_e)) \Rightarrow \psi[s_e/r]$ ,

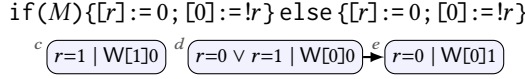
(R4c)  $(\forall s) \tau^D(\psi) \models (\bigwedge_{e \in E} \neg \theta_e) \Rightarrow \psi[s/r]$ .

Similarly, let  $\llbracket \cdot \rrbracket_5^{\text{rf}}$  be defined as for  $\llbracket \cdot \rrbracket_2^{\text{rf}}$  in Definition 4.7, with these definitions of *WRITE* and *READ*.

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



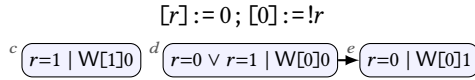
Merging, we have:



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



Sequencing and merging, we have:



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

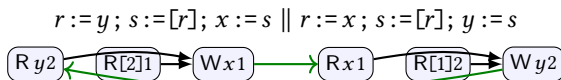
*Example 5.11.* The combination of read-read independency and address calculation (ADDR/RRD) is somewhat delicate. Combining Definition ?? and Definition ?? we have:

??)  $\tau^D(\psi) \models (L = \ell \Rightarrow v = r) \Rightarrow \psi$ ,

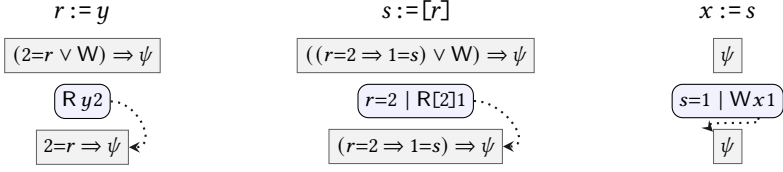
??)  $\tau^C(\psi) \models ((L = \ell \Rightarrow v = r) \vee W) \Rightarrow \psi$ .

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

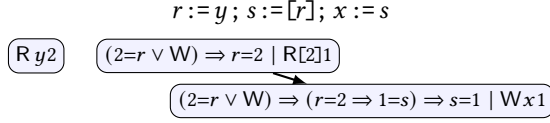
Consider the following program, from [Jagadeesan et al. 2020, §5], where initially  $x = 0, y = 0, [0] = 0, [1] = 2$ , and  $[2] = 1$ . It should only be possible to read 0, disallowing the attempted execution below:



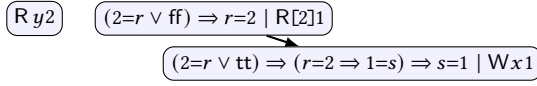
Looking at the left thread:



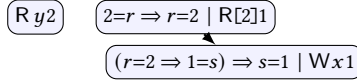
Composing, we have:



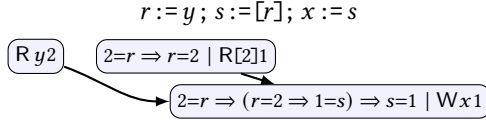
Substituting for W:



Which is:



The precondition of (R[2]1) is a tautology, but the precondition of (W x1) is not. This forces a dependency:



All the preconditions are now tautologies.

## 5.4 Access Elimination

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

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

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

*Definition 5.12.*

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

## 5.5 Merging Different labels

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

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

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

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

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

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

## 5.6 Read-Modify-Write Operations (RMW)

Extend the syntax

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

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

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

**Definition 5.14.** Let  $\text{READ}'$  be defined as for  $\text{READ}$ , adding the constraint:

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

If  $P \in \text{FADD}(r, x, M, \mu_1, \mu_2)$  then  $(\exists P_1 \in \text{READ}'(r, x, \mu_1); \text{WRITE}(x, r+M, \mu_2))$

(U1) if  $\lambda_1(e)$  is a write then there is a read  $\lambda_1(d)$  such that  $\kappa(e) \models \kappa(d)$  and  $d \xrightarrow{\text{rmw}} e$ .

If  $P \in \text{EXCHG}(r, x, M, \mu_1, \mu_2)$  then  $(\exists P_1 \in \text{READ}'(r, x, \mu_1); \text{WRITE}(x, M, \mu_2))$

(U1) if  $\lambda_1(e)$  is a write then there is a read  $\lambda_1(d)$  such that  $\kappa(e) \models \kappa(d)$  and  $d \xrightarrow{\text{rmw}} e$ .

If  $P \in \text{CAS}(r, x, M, N, \mu_1, \mu_2)$  then  $(\exists P_1 \in \text{READ}'(r, x, \mu_1); \text{IF}(r=M, \text{WRITE}(x, N, \mu_2), \text{SKIP}))$

(U1) if  $\lambda_1(e)$  is a write then there is a read  $\lambda_1(d)$  such that  $\kappa(e) \models \kappa(d)$  and  $d \xrightarrow{\text{rmw}} e$ .

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

(M8)  $\text{rmw} : E \rightarrow E$  is a partial function capturing read-modify-write *atomicity*, such that

(M8a) if  $d \xrightarrow{\text{rmw}} e$  then  $\lambda(e)$  blocks  $\lambda(d)$ ,

(M8b) if  $d \xrightarrow{\text{rmw}} e$  then  $d \leq e$ ,

(M8c) if  $\lambda(c)$  overlaps  $\lambda(d)$  then

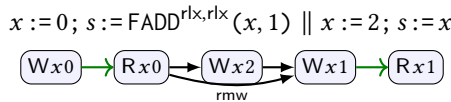
(i) if  $d \xrightarrow{\text{rmw}} e$  then  $c \leq e$  implies  $c \leq d$ ,

(ii) if  $d \xrightarrow{\text{rmw}} e$  then  $d \leq c$  implies  $e \leq c$ .

Extend the definition of  $\text{par}$ ,  $\text{if}$ ,  $\text{seq}$  to include:

(p0) (s0) (c0)  $\text{rmw} = (\text{rmw}_1 \cup \text{rmw}_2)$ ,

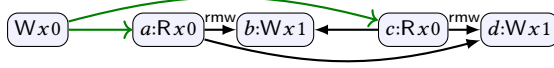
**Example 5.15.** This definition ensures atomicity, disallowing executions such as [Podkopaev et al. 2019, Ex. 3.2]:



By M8c(i), since  $(Wx2) \rightarrow (Wx1)$ , it must be that  $(Wx2) \rightarrow (Rx0)$ , creating a cycle.

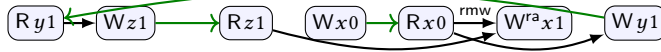


*Example 5.16.* Two successful rmws cannot see the same write:

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


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

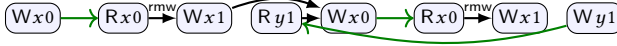
*Example 5.17.* By using two actions rather than one, the definition allows examples such as the following, which is allowed by Arm8 [Podkopaev et al. 2019, Ex. 3.10]:

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


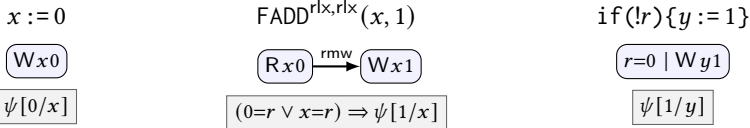
*Example 5.18.* For rmw operations, the independent case for a read should be the same as the empty case. To see why, consider the semantics of local invariant reasoning (LIR) from Definition ??:

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

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

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


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

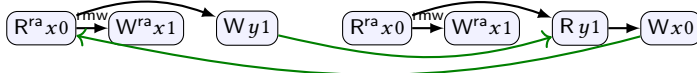


After sequencing, the precondition of  $(Wy1)$  is a tautology:  $(0=r \vee 0=r) \Rightarrow r=0$ .

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

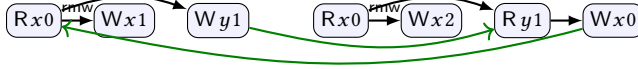
As a result, we disallow the use of ?? when treating the read event in an rmw.  
[Todo: write out the rules.]

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

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


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

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

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


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

If  $P \in \text{EXPAND}(\mathcal{P})$  then  $(\exists P_1, \dots, P_n \in \mathcal{P}) (\exists \theta_1, \dots, \theta_n \in \Phi)$

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

## REFERENCES

- Jade Alglave. 2020. This commit adds three alternative formulations of the Arm model, both for non-mixed and mixed size accesses. <https://github.com/herd/herdtools7/commit/685ee4>.
- Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. 2021. Armed Cats: Formal Concurrency Modelling at Arm. *TOPLAS* (2021). To Appear.
- Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. <https://doi.org/10.1145/2627752>
- Arm Limited. 2020. Arm Architecture Reference Manual: Armv8, for Armv8-A Architecture Profile (Issue F.c). <https://developer.arm.com/documentation/ddi0487/latest>.
- Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) (POPL '11). ACM, New York, NY, USA, 55–66. <https://doi.org/10.1145/1926385.1926394>
- Hans-J. Boehm and Brian Demsky. 2014. Outlawing Ghosts: Avoiding Out-of-thin-air Results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness* (Edinburgh, United Kingdom) (MSPC '14). ACM, New York, NY, USA, Article 7, 6 pages. <https://doi.org/10.1145/2618128.2618134>
- 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>
- Minkim 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](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 Bodik, 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](https://doi.org/10.1007/978-3-030-44914-8_22)
- Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA) (POPL '16)*. ACM, New York, NY, USA, 622–633. <https://doi.org/10.1145/2837614.2837616>
- Anton Podkopaev. 2020. Private correspondence.
- Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. 2019. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.* 3, POPL (2019), 69:1–69:31. <https://doi.org/10.1145/3290382>
- William Pugh. 2004. Causality Test Cases. <https://perma.cc/PJT9-XS8Z>
- Jaroslav Sevcík. 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 DISCUSSION

### A.1 Downset Closure

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

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

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

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

*Example A.2.* For RRD, Definition ?? states:

- $$\begin{aligned} ??) \quad \tau^D(\psi) \models v=r &\Rightarrow \psi, \\ ??) \quad \tau^C(\psi) \models (v=r \vee W) &\Rightarrow \psi, \\ ??) \quad \tau^B(\psi) \models \psi, &\text{ when } E = \emptyset. \end{aligned}$$

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

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

Rx0
Ry0

The semantics of this program includes the singleton pomset (Rx0), but not the singleton pomset (Ry0). To get (Rx0), we combine:

$$\begin{array}{cc} r := x & \text{if } (!r) \{ s := y \} \\ \boxed{\text{Rx0}} & \emptyset \end{array}$$

Attempting to get (Ry0), we instead get:

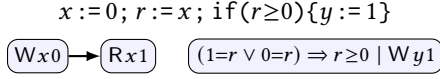
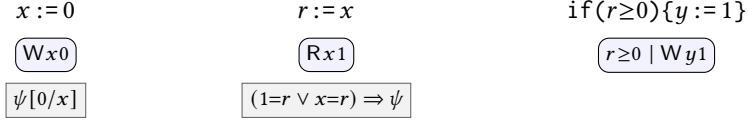
$$\begin{array}{cc} r := x & \text{if } (!r) \{ s := y \} \\ \emptyset & \boxed{r=0 \mid \text{Ry0}} \end{array}$$

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

*Example A.3.* For LIR, Definition ?? states:

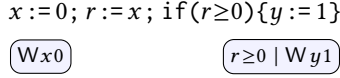
- $$\begin{aligned} ??) \quad \tau^D(\psi) \models v=r &\Rightarrow \psi, \\ ??) \quad \tau^C(\psi) \models (v=r \vee x=r) &\Rightarrow \psi, \text{ when } E \neq \emptyset, \\ ??) \quad \tau^B(\psi) \models \psi, &\text{ when } E = \emptyset. \end{aligned}$$

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



The precondition of  $(Wy1)$  is a tautology.

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



The precondition of  $(Wy1)$  is not a tautology.

## A.2 Comparison with Weakest Preconditions

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

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

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

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

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. In Definition ??, the transformers for the dependent case are as follows:

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

Only the read rule differs from the traditional one.

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

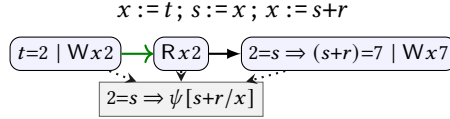
For example, let  $S_i$  be defined:

$$S_1 = s := x; x := s+r$$

$$S_2 = x := t; S_1$$

$$S_3 = t := 2; r := 5; S_2$$

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



The predicate transformers are:

$$wp_{S_1}(\psi) = x=s \Rightarrow \psi[s+r/x]$$

$$\tau_{S_1}(\psi) = 2=s \Rightarrow \psi[s+r/x]$$

$$wp_{S_2}(\psi) = t=s \Rightarrow \psi[s+r/x]$$

$$\tau_{S_2}(\psi) = 2=s \Rightarrow \psi[s+r/x]$$

$$wp_{S_3}(\psi) = 2=s \Rightarrow \psi[s+5/x]$$

$$\tau_{S_3}(\psi) = 2=s \Rightarrow \psi[s+5/x]$$

### A.3 Substitutions

Recall the load rules from §??:

$$??) \tau^D(\psi) \models v=r \Rightarrow \psi,$$

$$??) \tau^C(\psi) \models (v=r \vee x=r) \Rightarrow \psi, \text{ when } E \neq \emptyset,$$

$$??) \tau^B(\psi) \models \psi, \text{ when } E = \emptyset.$$

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

$$??) \tau^D(\psi) \models v=r \Rightarrow \psi[r/x],$$

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

$$??) \tau^B(\psi) \models \psi[r/x], \text{ when } E = \emptyset.$$

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

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

$$(r \wedge s \text{ even} \mid Wy1)$$

$$(r \wedge s \mid Wz1)$$

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:

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

$$(2=s \vee x=s) \Rightarrow (r \wedge s \text{ even}) \mid Wy1$$

$$(Rx2)$$

$$(2=s \vee x=s) \Rightarrow (r \wedge s) \mid Wz1$$

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

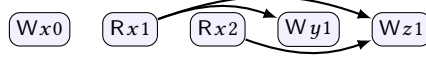
$$r := x; s := x; \text{if}(r \wedge s \text{ even})\{y := 1\}; \text{if}(r \wedge s)\{z := 1\}$$

$$(Rx1) \rightarrow (1=r \Rightarrow (2=s \vee x=s) \Rightarrow (r \wedge s \text{ even}) \mid Wy1)$$

$$(Rx2) \rightarrow (1=r \Rightarrow (2=s \vee x=s) \Rightarrow (r \wedge s) \mid Wz1)$$

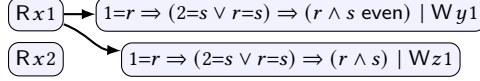
Prepending  $(x := 0)$ , which substitutes  $[0/x]$ , the precondition of  $(Wy1)$  becomes  $(1=r \Rightarrow (2=s \vee 0=s) \Rightarrow (r \wedge s \text{ even}))$ , which is a tautology, whereas the precondition of  $(Wz1)$  becomes  $(1=r \Rightarrow$

$(2=s \vee 0=s) \Rightarrow (r \wedge s)$ ), which is not. In order to be top-level,  $Wz1$  must depend on  $Rx2$ ; in this case the precondition becomes  $(1=r \Rightarrow 2=s \Rightarrow (r \wedge s))$ , which is a tautology.

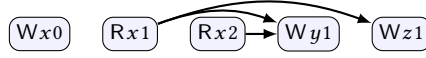


The situation reverses with the substitution  $[r/x]$ :

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



Prepending  $(x := 0)$ :



The dependency has changed from  $(Rx2) \rightarrow (Wz1)$  to  $(Rx2) \rightarrow (Wy1)$ . The resulting sets of pomsets are incomparable.

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

## B DIFFERENCES WITH OOPSLA

### B.1 Substitution

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

$$??) \tau^D(\psi) \models v=r \Rightarrow \psi,$$

$$??) \tau^C(\psi) \models (v=r \vee x=r) \Rightarrow \psi.$$

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

$$??) \tau^D(\psi) \models \psi[v/r][v/x],$$

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

The use of conjunction in ?? causes disjunction closure to fail because the predicate transformer  $\tau(\psi) = \psi' \wedge \psi''$  does not distribute through disjunction, even assuming that the prime operations do:<sup>7</sup>  $\tau(\psi_1 \vee \psi_2) = (\psi'_1 \vee \psi'_2) \wedge (\psi''_1 \vee \psi''_2) \neq (\psi'_1 \wedge \psi'_2) \vee (\psi'_2 \wedge \psi'_2) = \tau(\psi_1) \vee \tau(\psi_2)$ . See also Example 3.16.

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

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

<sup>7</sup>  $(\psi_1 \vee \psi_2)' = (\psi'_1 \vee \psi'_2)$  and  $(\psi_1 \vee \psi_2)'' = (\psi''_1 \vee \psi''_2)$ .

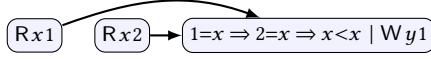


The substitution  $[x/r]$  does not work with Skolemization, even for the dependent case, since we lose the unique marker for each read. In effect, this forces the reads to the same values. To be concrete, the candidate definition would modify ?? to be:

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

Using this definition, consider the following:

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



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

## B.2 Downset closure

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

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



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:



## B.3 Consistency

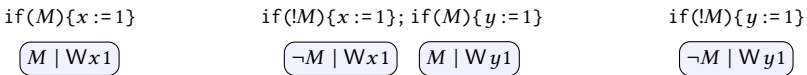
[Jagadeesan et al. 2020] imposes *consistency*, which requires that for every pomset  $P$ ,  $\bigwedge_e \kappa(e)$  is satisfiable. Associativity requires that we allow pomsets with inconsistent preconditions. Consider a variant of Example 5.6 from §??.



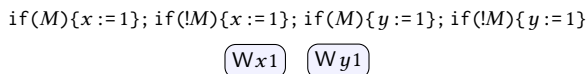
Associating left and right, we have:



Associating into the middle, instead, we require:



Joining left and right, we have:



## B.4 Causal Strengthening

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

$$\begin{array}{ccc} \text{if}(M)\{r:=x\} & y:=r & \text{if}(!M)\{s:=x\} \\ \boxed{M \mid Rx1} & \boxed{r=1 \mid Wy1} & \boxed{\neg M \mid Rx1} \end{array}$$

Associating left, with causal strengthening:

$$\begin{array}{ccc} \text{if}(M)\{r:=x\}; y:=r & & \text{if}(!M)\{s:=x\} \\ \boxed{M \mid Rx1} \rightarrow \boxed{M \mid Wy1} & & \boxed{\neg M \mid Rx1} \end{array}$$

Finally, merging:

$$\begin{array}{c} \text{if}(M)\{r:=x\}; y:=r; \text{if}(!M)\{s:=x\} \\ \boxed{Rx1} \rightarrow \boxed{M \mid Wy1} \end{array}$$

Instead, associating right:

$$\begin{array}{ccc} \text{if}(M)\{r:=x\} & y:=r; \text{if}(!M)\{s:=x\} & \\ \boxed{M \mid Rx1} & \boxed{r=1 \mid Wy1} \quad \boxed{\neg M \mid Rx1} & \end{array}$$

Merging:

$$\begin{array}{c} \text{if}(M)\{r:=x\}; y:=r; \text{if}(!M)\{s:=x\} \\ \boxed{Rx1} \rightarrow \boxed{Wy1} \end{array}$$

With causal strengthening, the precondition of  $Wy1$  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 Example 5.11 is:

$$\begin{array}{c} s := [r]; x := s \\ \boxed{r=2 \mid R[2]1} \rightarrow \boxed{(r=2 \Rightarrow 1=s) \Rightarrow s=1 \mid Wx1} \end{array}$$

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

$$\begin{array}{c} s := [r]; x := s \\ \boxed{r=2 \mid R[2]1} \rightarrow \boxed{r=2 \wedge [2]=1 \mid Wx1} \end{array}$$

Without causal strengthening, the precondition of  $(Wx1)$  would be simply  $[2]=1$ . The treatment in this paper, using implication rather than conjunction, is more precise.

## B.5 Parallel Composition

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

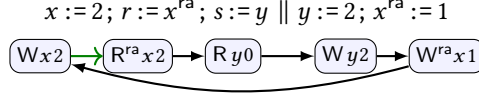
## B.6 Read-Modify-Write Actions

In [Jagadeesan et al. 2020], the atomicity axioms  $m8c$  erroneously applies only to overlapping writes, not overlapping reads. The difficulty can be seen in Example 5.16.

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

## B.7 Downgrading Internal Acquiring Reads

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

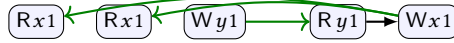


The solution we have adopted is to allow an acquiring read to be downgraded to a relaxed read when it is preceded (sequentially) by a relaxed write that could fulfill it. This solution allows executions that are not allowed under Arm8 since we do not insist that the local relaxed write is actually read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

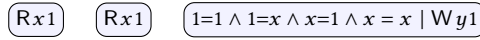
As a result, we use a different proof strategy for Arm8 implementation, which does not rely on read elimination. The proof idea uses a recent alternative characterization of Arm8 [Alglave 2020; Arm Limited 2020].

## B.8 Redundant Read Elimination

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

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


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

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


The precondition of (Wy1) is *not* a tautology, and therefore redundant read elimination fails. (It is a tautology in  $r := x; s := r; \text{if}(r=s)\{y := 1\}$ .) In [Jagadeesan et al. 2020, §3.1], we incorrectly stated that the precondition of (Wy1) was  $1=1 \wedge x=x$ .

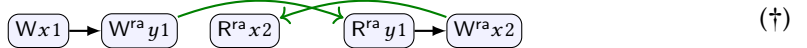
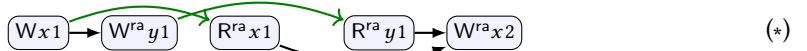
## B.9 Stupid Bug in LDRF

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

## C MORE STUFF

### C.1 A Note on Mixed-Mode Data Races

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

$$x := 1; y^{ra} := 1; r := x^{ra} \parallel \text{if}(y^{ra})\{x^{ra} := 2\}$$


The program is data-race free. The two executions shown are the only top-level executions that include (W<sup>ra</sup>x2).

Theorem 4.1 of [Dongol et al. 2019] is stated by extending execution sequences. In the terminology of [Dongol et al. 2019], a read is *L-weak* if it is sequentially stale. Let  $\rho = (Wx1)(W^ra y1)(R^ra y1)(W^ra x2)$  be a sequence and  $\alpha = (R^ra x1)$ .  $\rho$  is *L-sequential* and  $\alpha$  is *L-weak* in  $\rho\alpha$ . But there is no execution of this program that includes a data race, contradicting the theorem. The error seems to be in Lemma A.4 of [Dongol et al. 2019], which states that if  $\alpha$  is *L-weak* after an *L-sequential*  $\rho$ , then  $\alpha$  must be in a data race. That is clearly false here, since  $(R^ra x1)$  is stale, but the program is data race free.

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



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



is a downset of  $(*)$ .

## C.2 If Closure and Address Dependencies

An optimization ( $p/q$  are registers):

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

vs

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

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

If closure is at odds with Java Final field semantics.

Do sequencing and if commute?

## C.3 About Arm

Hypothesis: gcb cannot contradict (poloc minus RxR).