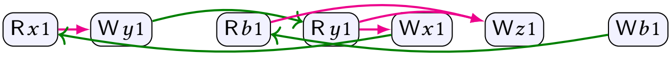
A DISCUSSION

A.1 Comparison to "Promising Semantics" [POPL 2017]

Recently, Cho et al. [2021] showed that certain combinations of compiler optimizations are inconsistent with local DRF guarantees. All of the examples that prove inconsistency have the same shape: they combine read-introduction and case analysis (aka, if-introduction). Effectively, this turns one read into two, where different conditional branches can be taken for the two copies of the read. This is reminiscent of the type of *bait and switch* behavior noted by Jagadeesan et al. [2020]: the promising semantics (PS) [Kang et al. 2017] and related models [Chakraborty and Vafeiadis 2019; Jagadeesan et al. 2010; Manson et al. 2005], fail to validate compositional reasoning of temporal properties. Consider example OOTA4 from [Jagadeesan et al. 2020]:

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

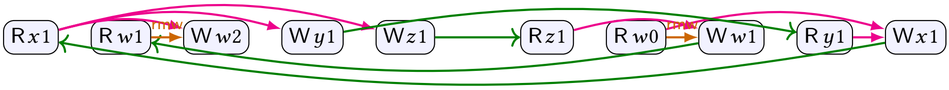


(OOTA4)

Under all variants of PwT, this outcome is disallowed, due to the cycle involving x and y.[1] Under PS, this outcome is allowed by baiting with the `else` branch, then switching to the `then` branch, based on a coin flip (b).

Cho et al. [2021] introduce more complex examples to show that the promising semantics fails LDRF-SC.[2] Here is one, dubbed LDRF-FAIL-PS.

$$\texttt{if}(x)\{\texttt{FADD}(w, 1)\,;\ y := 1\,;\ z := 1\} \parallel \texttt{if}(z)\{\texttt{if}(!\texttt{FADD}(w, 1))\{x := y\}\}\,\texttt{else}\,\{x := 1\}$$



Again, all variants of PwT disallow the outcome due to the cycle involving x and y. It is allowed by PS by baiting the second thread with $x := 1$ in the `else` branch, then switching to the `then` branch. This shows some some structural resemblance to OOTA4, with z replacing b.

Cho et al. argue that the outcome of LDRF-FAIL-PS is inevitable due to compiler optimizations. The examples crucially involve the following sequence of operations:

- read-introduction,
- if-introduction, branching on the read just introduced.

We believe this combination of optimizations is unsound. This is obviously the case in C11: read-introduction may cause undefined behavior (UB), due to the possible introduction of a data race.

The situation is more delicate in LLVM. The short version of the story is that load-hoisting followed by case analysis is unsound in LLVM, without freeze. This happens because:

- read-introduction may result in the undefined value **undef**, due to the possible introduction of a data race [Chakraborty and Vafeiadis 2017], and
- branching on an undefined value in LLVM results in UB.

LLVM delays UB using the undefined value. This allows LLVM to perform optimizations such as load hoisting, where $\texttt{if}(C)\{r := x\}$ is rewritten to $s := x\,;\ r := C?s:r$. Despite this, other optimizations regularly performed by LLVM are unsound [Lee et al. 2017]. An example is loop switching, where $\texttt{while}(C_1)\{\texttt{if}(C_2)\{S_1\}\,\texttt{else}\,\{S_2\}\}$ is rewritten to $\texttt{if}(C_2)\{\texttt{while}(C_1)\{S_1\}\}\,\texttt{else}\,\{\texttt{while}$

[1] All of the reads in OOTA4 are cross-thread, so there is no difference between PwT-MCA$_1$ and PwT-MCA$_2$. For PwT-C11, there is a cycle in $\mathsf{rf} \cup <$.

[2] Cho et al. [2021] show that by restricting RMW-store reorderings, one can establish LDRF-SC for PS. We speculate that no such restriction is required for PwT. (We did not treat RMWs in our proof of LDRF-SC.)

$(C_1)\{S_2\}\}$. Freeze was introduced in LLVM in order to make such optimizations sound by allowing branch on frozen **undef** to give nondeterministic choice rather than UB. In the RFC for freeze, Lopes [2016] says: *"Note that having branch on poison not trigger* UB *has its own problems. We believe this is a good tradeoff."* LDRF-FAIL-PS demonstrates a concrete problem with this tradeoff. Other compilers, such as Compcert, are more conservative [Lee et al. 2017, §9].

Thus, the difference between PS and PwT can be understood in terms of the valid program transformations. PS allows reads to be introduced, with subsequent case analysis on the value read. PwT validates case analysis, but invalidates read-introduction.

Allowing executions such as OOTA4 and LDRF-FAIL-PS also invalidates compositional reasoning for temporal safety properties (see §6).

These differences highlight the subtle tensions between compiler optimizations and program logics that are revealed by relaxed memory models. It is not possible to have everything one wants. Thus, one is forced to choose which optimizations and reasoning principles are most important.[3]

Finally, we note that it is possible that PS is properly weaker than PwT.

A.2 Comparison to "Pomsets with Preconditions" [OOPSLA 2020]

PwT-MCA is closely related to PwP model of [Jagadeesan et al. 2020]. The major difference is that PwT-MCA supports sequential composition. In the remainder of this section, we discuss other differences. We also point out some errors in [Jagadeesan et al. 2020], all of which have been confirmed by the authors.

Substitution. PwP uses substitution rather than Skolemizing. Indeed our use of Skolemization is motivated by disjunction closure for predicate transformers, which do not appear in PwP. In Fig. 1, we gave the semantics of read for nonempty pomsets as:

$$\begin{array}{ll}\text{(R4a)} & \text{if } (E \cap D) \neq \emptyset \text{ then } \tau^D(\psi) \equiv v{=}r \Rightarrow \psi,\\ \text{(R4b)} & \text{if } (E \cap D) = \emptyset \text{ then } \tau^D(\psi) \equiv (v{=}r \vee x{=}r) \Rightarrow \psi.\end{array}$$

In PwP, the definition is roughly as follows:

$$\begin{array}{ll}\text{(R4a}'\text{)} & \text{if } (E \cap D) \neq \emptyset \text{ then } \tau^D(\psi) \equiv \psi[v/r][v/x],\\ \text{(R4b}'\text{)} & \text{if } (E \cap D) = \emptyset \text{ then } \tau^D(\psi) \equiv \psi[v/r][v/x] \wedge \psi[x/r]\end{array}$$

The use of conjunction in R4b′ causes disjunction closure to fail because the predicate transformer $\tau(\psi) = \psi' \wedge \psi''$ does not distribute through disjunction, even assuming that the prime operations do:[4] $\tau(\psi_1 \vee \psi_2) = (\psi_1' \vee \psi_2') \wedge (\psi_1'' \vee \psi_2'') \neq (\psi_1' \wedge \psi_1'') \vee (\psi_2' \wedge \psi_2'') = \tau(\psi_1) \vee \tau(\psi_2)$. See also §4.9.

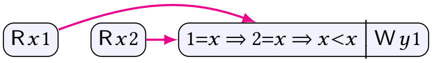
The substitutions collapse x and r, allowing local invariant reasoning (LIR), as required by JMM causality test case 1, discussed in §4.8. Without Skolemizing it is necessary to substitute $[x/r]$, since the reverse substitution $[r/x]$ is useless when r is bound—compare with §A.7. As discussed below (Downset closure), including this substitution affects the interaction of LIR and downset closure.

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

[3] Another example is the tension between load hoisting—forbidden in C11 but allowed by LLVM—and common subexpression elimination over an acquiring lock—allowed by C11 but forbidden by LLVM [Chakraborty and Vafeiadis 2017].
[4] $(\psi_1 \vee \psi_2)' = (\psi_1' \vee \psi_2')$ and $(\psi_1 \vee \psi_2)'' = (\psi_1'' \vee \psi_2'')$.

The substitution $[x/r]$ does not work with Skolemization, even for the dependent case, since we lose the unique marker for each read. In effect, this forces all reads of a location to see the same values. Using this definition, consider the following:

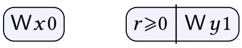
$$r := x\,;\ s := x\,;\ \mathtt{if}(r<s)\{y := 1\}$$



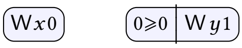
Although the execution seems reasonable, the precondition on the write is not a tautology.

Downset closure. PwP 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 PwP. Consider the final pomset in the last example of §A.8 under the semantics of this paper, which elides the middle read event:

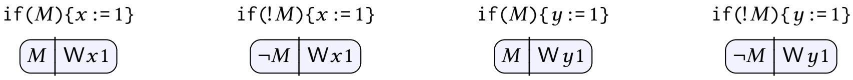
$$x := 0\,;\ r := x\,;\ \mathtt{if}(r\geqslant 0)\{y := 1\}$$



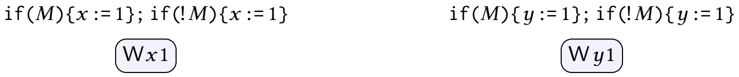
In PwP, the substitution $[x/r]$ is performed by the middle read regardless of whether it is included in the pomset, with the subsequent substitution of $[0/x]$ by the preceding write, we have $[x/r][0/x]$, which is $[0/r][0/x]$, resulting in:



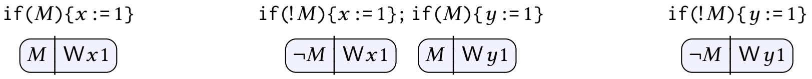
Consistency. PwP imposes *consistency*, which requires that for every pomset P, $\bigwedge_e \kappa(e)$ is satisfiable. Associativity requires that we allow pomsets with inconsistent preconditions. Consider a variant of the example from §9.3.



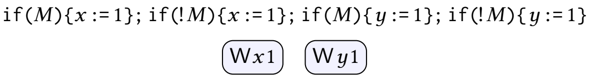
Associating left and right, we have:


if(M){x := 1}; if(!M){x := 1}
Wx1
if(M){y := 1}; if(!M){y := 1}
Wy1

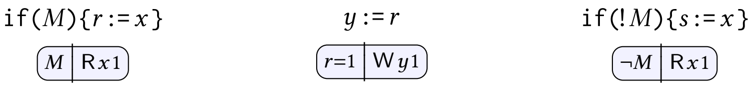
Associating into the middle, instead, we require:



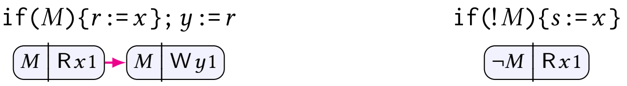
Joining left and right, we have:


if(M){x := 1}; if(!M){x := 1}; if(M){y := 1}; if(!M){y := 1}
Wx1
Wy1

Causal Strengthening. PwP imposes *causal strengthening*, which requires for every pomset P, if $d < e$ then $\kappa(e) \vDash \kappa(d)$. Associativity requires that we allow pomsets without causal strengthening. Consider the following.

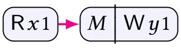


Associating left, with causal strengthening:



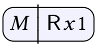
Finally, merging:

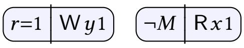
$$\mathtt{if}(M)\{r:=x\};\ y:=r;\ \mathtt{if}(!M)\{s:=x\}$$



Instead, associating right:

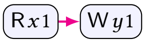
$$\mathtt{if}(M)\{r:=x\} \qquad\qquad y:=r;\ \mathtt{if}(!M)\{s:=x\}$$





Merging:

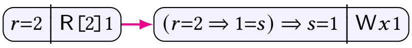
$$\mathtt{if}(M)\{r:=x\};\ y:=r;\ \mathtt{if}(!M)\{s:=x\}$$



With causal strengthening, the precondition of Wy1 depends upon how we associate. This is not an issue in PwP, which always associates to the right.

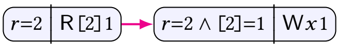
One use of causal strengthening is to ensure that address dependencies do not introduce thin air reads. Associating to the right, the intermediate state of ADDR2 (§9.4) is:

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



In PwP, we have, instead:

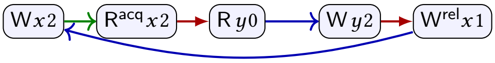
$$s:=[r]\,;\ x:=s$$



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.

Internal Acquiring Reads. The proof of compilation to Arm in PwP assumes that all internal reads can be eliminated. However, this is not the case for acquiring reads. For example, PwP disallows the following execution, where the final values of x is 2 and the final value of y is 2. This execution is allowed by Arm8 and TSO.

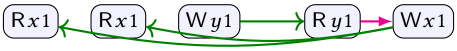
$$x:=2;\ r:=x^{\mathsf{acq}};\ s:=y \parallel y:=2;\ x^{\mathsf{rel}}:=1$$



We discuss two approaches to this problem in §B.

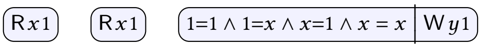
Redundant Read Elimination. Contrary to the claim, redundant read elimination fails for PwP. We discuss redundant read elimination in §9.1. Consider JMM Causality Test Case 2, which we describe there.

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



Under the semantics of PwP, we have

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



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

A.3 Register Consistency

In addition to the three criteria of Def. 4.2 Dijkstra [1975] requires

(x4′) $\tau(\mathsf{ff}) \equiv \mathsf{ff}$.

Unfortunately, our transformer for read actions (R4a) does not obey x4′, since ff is not equivalent to $v{=}r \Rightarrow \mathsf{ff}$.

In this subsection, we refine this requirement to one that does hold. The main insight is to pull values for registers from the actions of pomset itself. Thus, we define θ_λ to capture the *register state* of a pomset.

Definition A.1. Let $\theta_\lambda = \bigwedge_{\{(e,v)\in(E\times\mathcal{V}) \mid \lambda(e)=(\mathsf{R}v)\}} (s_e{=}v)$ where $E = \mathsf{dom}(\lambda)$.
We say that ϕ is λ*-consistent* if $\phi \land \theta_\lambda$ is satisfiable. We say that it is λ*-inconsistent* otherwise.

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

Definition A.2. A λ*-predicate transformer* is a function $\tau : \Phi \to \Phi$ such that

(x1) (x2) (x3) as in Def. 4.2,
(x4) if ψ is λ-inconsistent then $\tau(\psi)$ is λ-inconsistent.

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

(M4) $\tau : 2^{\mathcal{E}} \to \Phi \to \Phi$ is a *family of* λ*-predicate transformers*,

A.4 Comparison with Sequential Predicate Transformers

We compare traditional transformers to the dependent-case transformers of Fig. 1.

All programs in our language are strongly normalizing, so we need not distinguish strong and weak correctness. In this setting, the Hoare triple $\{\phi\}\ S\ \{\psi\}$ holds exactly when $\phi \Rightarrow wp_S(\psi)$.

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

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

Here we have chosen an alternative formulation for the read rule, which is equivalent to the more traditional $\psi[x/r]$, as long as registers are assigned at most once in a program. Our predicate transformers for the dependent case are shown on the right above. Only the read rule differs from the traditional one.

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

For example, let S_i be defined:

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

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

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

$$\boxed{t{=}2 \mid \mathsf{W}x2} \rightarrow \boxed{\mathsf{R}x2} \rightarrow \boxed{2{=}s \Rightarrow (s{+}r){=}7 \mid \mathsf{W}x7} \dashrightarrow \boxed{2{=}s \Rightarrow \psi[s{+}r/x]}$$

The predicate transformers are:

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

A.5 The Need for Respect

In Fig. 1, we choose the weakest precondition. Because of this, associativity requires that s6 is ($<$ respects $<_1$ and $<_2$) rather than ($< \supseteq (<_1 \cup <_2)$). Consider ($r := x$; $y := M$; `skip`). Associating to the left, we might have:

$$P_{12} = \boxed{\mathsf{R}x}^{d}\ \boxed{\phi \mid \mathsf{W}y}^{e} \qquad\qquad P_3 = \emptyset \qquad\qquad P = \boxed{\mathsf{R}x}^{d} \rightarrow \boxed{\phi \mid \mathsf{W}y}^{e}$$

When building P_{12}, the dependent set of e would be the empty set, and thus ϕ must have been constructed using the independent transformer R4b. Attempting to repeat this, associating to the right:

$$P_1 = \boxed{\mathsf{R}x}^{d} \qquad\qquad P_{23} = \boxed{\phi' \mid \mathsf{W}y}^{e} \qquad\qquad P' = \boxed{\mathsf{R}x}^{d} \rightarrow \boxed{\phi' \mid \mathsf{W}y}^{e}$$

In P', however, now the dependent set of e is the singleton $\{d\}$; thus ϕ' must be constructed using the dependent transformer R4a. Since $((v{=}r \vee x{=}r) \Rightarrow \psi) \not\equiv (v{=}r \Rightarrow \psi)$, associativity fails.

If we allow stronger preconditions, as in [Jagadeesan et al. 2020], then we could use inclusion rather than *respects*. To arrive at this semantics, one would replace every occurrence of $\equiv$ in Fig. 1 with $\vDash$. Then ($<$ respects $<_1$ and $<_2$) can be replaced by ($< \supseteq (<_1 \cup <_2)$).

A.6 Write Substitutions

[Todo: Discuss.]

Alan example of why substitute M/x rather than v/x in the write rule:

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

$$\boxed{\mathsf{R}y1} \rightarrow \boxed{\mathsf{W}x1} \quad \boxed{\mathsf{R}x1} \quad \boxed{\mathsf{W}z1}$$

We lost the order from $\mathsf{R}y1$ to $\mathsf{W}z1$.

$$s := x\,;\ z := s$$

$$\boxed{\mathsf{R}x1} \quad \boxed{x{=}1 \mid \mathsf{W}z1}$$

$$x := r\,;\ s := x\,;\ z := s$$

$$\boxed{\mathsf{W}x1} \quad \boxed{\mathsf{R}x1} \quad \boxed{1{=}1 \mid \mathsf{W}z1}$$

$$\boxed{\mathsf{W}x1} \quad \boxed{\mathsf{R}x1} \quad \boxed{r{=}1 \mid \mathsf{W}z1}$$

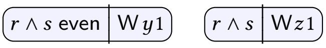
A.7 Read Substitutions

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

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

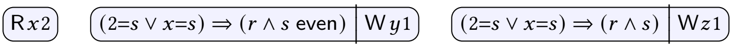
Perhaps surprisingly, this semantics is incomparable with that of Fig. 1. Consider the following:

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



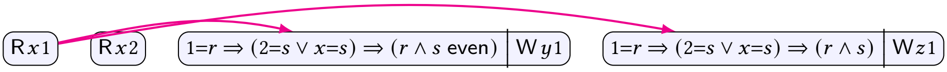
Prepending $(s := x)$, we get the same result regardless of whether we substitute $[s/x]$, since x does not occur in either precondition. Here we show the independent case:

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

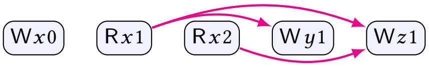

Rx2
$(2{=}s \vee x{=}s) \Rightarrow (r \wedge s$ even$)$ | Wy1
$(2{=}s \vee x{=}s) \Rightarrow (r \wedge s)$ | Wz1

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

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

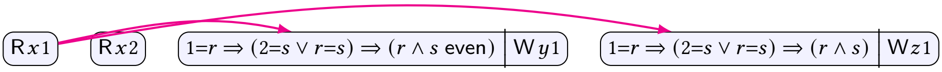

Rx1
Rx2
$1{=}r \Rightarrow (2{=}s \vee x{=}s) \Rightarrow (r \wedge s$ even$)$ | Wy1
$1{=}r \Rightarrow (2{=}s \vee x{=}s) \Rightarrow (r \wedge s)$ | Wz1

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 be dependency ordered after (Rx2); in this case the precondition becomes $(1{=}r \Rightarrow 2{=}s \Rightarrow (r \wedge s))$, which is a tautology.

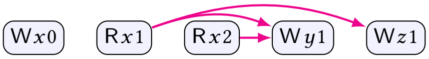


The situation reverses with the substitution $[r/x]$:

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


Rx1
Rx2
$1{=}r \Rightarrow (2{=}s \vee r{=}s) \Rightarrow (r \wedge s$ even$)$ | Wy1
$1{=}r \Rightarrow (2{=}s \vee r{=}s) \Rightarrow (r \wedge s)$ | Wz1

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 read-read dependencies, which is not the case for Arm8.

A.8 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.3. 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) $(\forall e \in E_2)\ \kappa_2(e) \equiv \kappa_1(e)$,
(4) $(\forall e \in E_2)\ \tau_2^D(e) \equiv \tau_1^D(e)$,
(5) $\checkmark_2 \vDash \checkmark_1$,
(6a) $(\forall d \in E_2)\ (\forall e \in E_2)\ d <_2 e$ iff $d <_1 e$,
(6b) $(\forall d \in E_1)\ (\forall e \in E_2)$ if $d <_1 e$ then $d \in E_2$,
(7) $(\forall d \in E_2)\ (\forall e \in E_2)\ d \ \mathsf{rf}_2\ e$ iff $d \ \mathsf{rf}_1\ e$.

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

First, downset closure fails for read-read independency §4.7. Consider

$$\begin{array}{c} r\,{:=}\,x\,;\ \mathtt{if}(!r)\{s\,{:=}\,y\} \\ \boxed{\mathsf{R}\,x0}\quad \boxed{\mathsf{R}\,y0} \end{array}$$

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

$$\begin{array}{c} r\,{:=}\,x \\ \boxed{\mathsf{R}\,x0} \end{array} \qquad\qquad \begin{array}{c} \mathtt{if}(!r)\{s\,{:=}\,y\} \\ \emptyset \end{array}$$

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

$$\begin{array}{c} r\,{:=}\,x \\ \emptyset \end{array} \qquad\qquad \begin{array}{c} \mathtt{if}(!r)\{s\,{:=}\,y\} \\ \boxed{r{=}0 \mid \mathsf{R}\,y0} \end{array}$$

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

Second, the semantics is not downset closed because the independency reasoning of R4b is only applicable for pomsets where the ignored read is present! Revisiting JMM causality test case 1 from the end of §4.6:

$$\begin{array}{c} x\,{:=}\,0 \\ \boxed{\mathsf{W}\,x0} \\ \boxed{\psi[0/x]} \end{array} \qquad\qquad \begin{array}{c} r\,{:=}\,x \\ \boxed{\mathsf{R}\,x1} \\ \boxed{(1{=}r \lor x{=}r) \Rightarrow \psi} \end{array} \qquad\qquad \begin{array}{c} \mathtt{if}(r{\geqslant}0)\{y\,{:=}\,1\};\ z\,{:=}\,r \\ \boxed{r{\geqslant}0 \mid \mathsf{W}\,y1}\quad \boxed{r{=}1 \mid \mathsf{W}\,z1} \end{array}$$

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

$$\begin{tikzcd} \boxed{\mathsf{W}\,x0} \arrow[r] & \boxed{\mathsf{R}\,x1} \arrow[rr, bend left] & \boxed{(1{=}r \lor 0{=}r) \Rightarrow r{\geqslant}0 \mid \mathsf{W}\,y1} & \boxed{1{=}r \Rightarrow r{=}1 \mid \mathsf{W}\,z1} \end{tikzcd}$$

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

Taking the empty set for the read, however, the precondition of $(\mathsf{W}\,y1)$ is not a tautology:

$$\begin{array}{c} x\,{:=}\,0\,;\ r\,{:=}\,x\,;\ \mathtt{if}(r{\geqslant}0)\{y\,{:=}\,1\};\ z\,{:=}\,r \\ \boxed{\mathsf{W}\,x0}\qquad\qquad \boxed{r{\geqslant}0 \mid \mathsf{W}\,y1}\quad \boxed{r{=}1 \mid \mathsf{W}\,z1} \end{array}$$

One way to deal with the second issue would be to allow general access elimination to merge $(\mathsf{W}\,x0)$ and $(\mathsf{R}\,x0)$:

$$\begin{array}{c} x\,{:=}\,0\,;\ r\,{:=}\,x\,;\ \mathtt{if}(r{\geqslant}0)\{y\,{:=}\,1\};\ z\,{:=}\,r \\ \boxed{\mathsf{W}\,x0}\qquad\qquad \boxed{(0{=}r \lor 0{=}r) \Rightarrow r{\geqslant}0 \mid \mathsf{W}\,y1}\quad \boxed{r{=}1 \mid \mathsf{W}\,z1} \end{array}$$

We leave the elaboration of this idea to future work.

A.9 Logical Encoding of Delay for PwT-MCA

[Todo: Remove this section?]

In this subsection, we develop a logical encoding of delay, which can replace I6a in PwT-MCA$_1$. It is not obvious how to repeat this trick for PwT-MCA$_2$, due to thread-local reads-from and thread-local blockers (s6a and s6b in Def. 5.2).

As motivation, recall that we stated Lemma 4.6(g) using inclusions:

(g) $[\![\mathtt{if}(\neg\phi)\{S_2\};\ \mathtt{if}(\phi)\{S_1\}]\!] \subseteq [\![\mathtt{if}(\phi)\{S_1\}\,\mathtt{else}\,\{S_2\}]\!] \supseteq [\![\mathtt{if}(\phi)\{S_1\};\ \mathtt{if}(\neg\phi)\{S_2\}]\!]$.

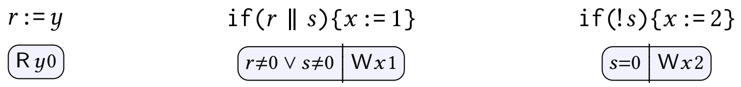
PwT-mca does not satisfy the reverse inclusion. The culprit is delay, which introduces order regardless of whether preconditions are disjoint. As an example, $\llbracket \texttt{if}(r)\{x := 1\}\,\texttt{else}\,\{x := 2\}\rrbracket$ has an execution with $(r{=}0 \mid \mathsf{W}x2) \rightarrow (r{\neq}0 \mid \mathsf{W}x1)$, (using augmentation), whereas $\llbracket \texttt{if}(r)\{x := 1\};\ \texttt{if}(!r)\{x := 2\}\rrbracket$ has no such execution.

[Todo: Put an example for PwT-po.]

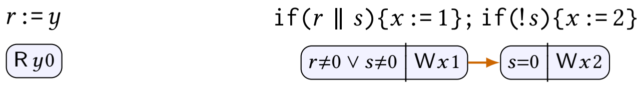
In order to validate the reverse inclusions, we could require that i6a not impose order when $\kappa_1(d) \wedge \kappa_2(e)$ is unsatisfiable. Thus, following on §A.3, we would also like this:

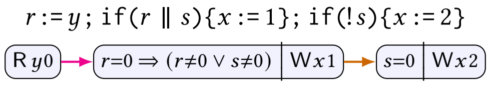
(s6b′) if $\lambda_1(d)$ delays $\lambda_2(e)$ and $\kappa_1(d) \wedge \kappa'_2(e)$ is λ-consistent then $d \leq e$.

However, (s6b′) fails associativity. Example where $\theta_\lambda = (r{=}0)$

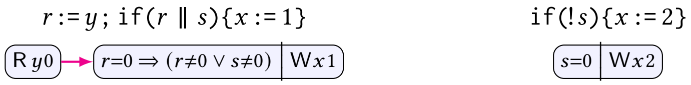


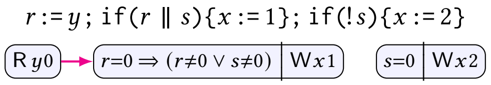
Associating right, order is required since $((r{\neq}0 \vee s{\neq}0) \wedge s{=}0)$ is satisfiable (take $r{=}1$ and $s{=}0$):





Associating left, order is not required between the writes since $(s{\neq}0 \wedge s{=}0)$ is unsatisfiable:





This motivates the logic-based presentation of delay.

In the data model, we require additional symbols: Q_sc, Q^x_ro, and Q^x_wo. We refer to these collectively as *quiescence symbols*.

We update the Def. 4.4 of complete pomset to substitute true for every quiescence symbol (notation $[\mathsf{tt}/\mathsf{Q}]$):

Definition A.4. A PwT is *complete* if
(c3) $\kappa(e)[\mathsf{tt}/\mathsf{Q}]$ is a tautology, (c5) $\checkmark[\mathsf{tt}/\mathsf{Q}]$ is a tautology.

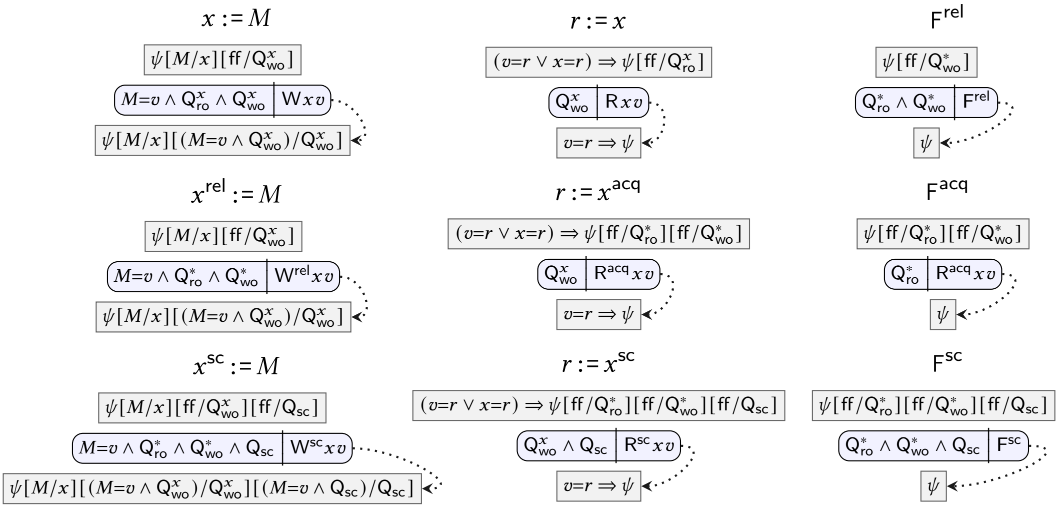
We define some helper notation:

Definition A.5. Let $\mathsf{Q}^*_\mathsf{ro} = \bigwedge_y \mathsf{Q}^y_\mathsf{ro}$, and similarly for Q^*_wo.
Let formulae $\mathsf{Q}^{\mathsf{S}x}_\mu$, $\mathsf{Q}^{\mathsf{L}x}_\mu$, and $\mathsf{Q}^{\mathsf{F}}_\mu$ be defined:

$$\begin{array}{lll}
\mathsf{Q}^{\mathsf{S}x}_\mathsf{rlx} = \mathsf{Q}^x_\mathsf{ro} \wedge \mathsf{Q}^x_\mathsf{wo} & \mathsf{Q}^{\mathsf{L}x}_\mathsf{rlx} = \mathsf{Q}^x_\mathsf{wo} & \mathsf{Q}^{\mathsf{F}}_\mathsf{rel} = \mathsf{Q}^*_\mathsf{ro} \wedge \mathsf{Q}^*_\mathsf{wo} \\
\mathsf{Q}^{\mathsf{S}x}_\mathsf{rel} = \mathsf{Q}^*_\mathsf{ro} \wedge \mathsf{Q}^*_\mathsf{wo} & \mathsf{Q}^{\mathsf{L}x}_\mathsf{acq} = \mathsf{Q}^x_\mathsf{wo} & \mathsf{Q}^{\mathsf{F}}_\mathsf{acq} = \mathsf{Q}^*_\mathsf{ro} \\
\mathsf{Q}^{\mathsf{S}x}_\mathsf{sc} = \mathsf{Q}^*_\mathsf{ro} \wedge \mathsf{Q}^*_\mathsf{wo} \wedge \mathsf{Q}_\mathsf{sc} & \mathsf{Q}^{\mathsf{L}x}_\mathsf{sc} = \mathsf{Q}^x_\mathsf{wo} \wedge \mathsf{Q}_\mathsf{sc} & \mathsf{Q}^{\mathsf{F}}_\mathsf{sc} = \mathsf{Q}^*_\mathsf{ro} \wedge \mathsf{Q}^*_\mathsf{wo} \wedge \mathsf{Q}_\mathsf{sc}
\end{array}$$

Let $[\phi/\mathsf{Q}^*_\mathsf{ro}]$ substitute ϕ for every Q^y_ro, and similarly for Q^*_wo.
Let substitutions $[\phi/\mathsf{Q}^{\mathsf{S}x}_\mu]$, $[\phi/\mathsf{Q}^{\mathsf{L}x}_\mu]$, and $[\phi/\mathsf{Q}^{\mathsf{F}}_\mu]$ be defined:

$$\begin{array}{lll}
[\phi/\mathsf{Q}^{\mathsf{S}x}_\mathsf{rlx}] = [\phi/\mathsf{Q}^x_\mathsf{wo}] & [\phi/\mathsf{Q}^{\mathsf{L}x}_\mathsf{rlx}] = [\phi/\mathsf{Q}^x_\mathsf{ro}] & [\phi/\mathsf{Q}^{\mathsf{F}}_\mathsf{rel}] = [\phi/\mathsf{Q}^*_\mathsf{wo}] \\
[\phi/\mathsf{Q}^{\mathsf{S}x}_\mathsf{rel}] = [\phi/\mathsf{Q}^x_\mathsf{wo}] & [\phi/\mathsf{Q}^{\mathsf{L}x}_\mathsf{acq}] = [\phi/\mathsf{Q}^*_\mathsf{ro}, \phi/\mathsf{Q}^*_\mathsf{wo}] & [\phi/\mathsf{Q}^{\mathsf{F}}_\mathsf{acq}] = [\phi/\mathsf{Q}^*_\mathsf{ro}, \phi/\mathsf{Q}^*_\mathsf{wo}] \\
[\phi/\mathsf{Q}^{\mathsf{S}x}_\mathsf{sc}] = [\phi/\mathsf{Q}^x_\mathsf{wo}, \phi/\mathsf{Q}_\mathsf{sc}] & [\phi/\mathsf{Q}^{\mathsf{L}x}_\mathsf{sc}] = [\phi/\mathsf{Q}^*_\mathsf{ro}, \phi/\mathsf{Q}^*_\mathsf{wo}, \phi/\mathsf{Q}_\mathsf{sc}] & [\phi/\mathsf{Q}^{\mathsf{F}}_\mathsf{sc}] = [\phi/\mathsf{Q}^*_\mathsf{ro}, \phi/\mathsf{Q}^*_\mathsf{wo}, \phi/\mathsf{Q}_\mathsf{sc}]
\end{array}$$



$x := M$
$\psi[M/x][\mathsf{ff}/\mathsf{Q}^x_{\mathsf{wo}}]$
$M{=}v \land \mathsf{Q}^x_{\mathsf{ro}} \land \mathsf{Q}^x_{\mathsf{wo}}$ | $\mathsf{W}xv$
$\psi[M/x][(M{=}v \land \mathsf{Q}^x_{\mathsf{wo}})/\mathsf{Q}^x_{\mathsf{wo}}]$
$r := x$
$(v{=}r \lor x{=}r) \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^x_{\mathsf{ro}}]$
$\mathsf{Q}^x_{\mathsf{wo}}$ | $\mathsf{R}xv$
$v{=}r \Rightarrow \psi$
$\mathsf{F}^{\mathsf{rel}}$
$\psi[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}]$
$\mathsf{Q}^*_{\mathsf{ro}} \land \mathsf{Q}^*_{\mathsf{wo}}$ | $\mathsf{F}^{\mathsf{rel}}$
ψ
$x^{\mathsf{rel}} := M$
$\psi[M/x][\mathsf{ff}/\mathsf{Q}^x_{\mathsf{wo}}]$
$M{=}v \land \mathsf{Q}^*_{\mathsf{ro}} \land \mathsf{Q}^*_{\mathsf{wo}}$ | $\mathsf{W}^{\mathsf{rel}}xv$
$\psi[M/x][(M{=}v \land \mathsf{Q}^x_{\mathsf{wo}})/\mathsf{Q}^x_{\mathsf{wo}}]$
$r := x^{\mathsf{acq}}$
$(v{=}r \lor x{=}r) \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{ro}}][\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}]$
$\mathsf{Q}^x_{\mathsf{wo}}$ | $\mathsf{R}^{\mathsf{acq}}xv$
$v{=}r \Rightarrow \psi$
$\mathsf{F}^{\mathsf{acq}}$
$\psi[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{ro}}][\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}]$
$\mathsf{Q}^*_{\mathsf{ro}}$ | $\mathsf{R}^{\mathsf{acq}}xv$
ψ
$x^{\mathsf{sc}} := M$
$\psi[M/x][\mathsf{ff}/\mathsf{Q}^x_{\mathsf{wo}}][\mathsf{ff}/\mathsf{Q}_{\mathsf{sc}}]$
$M{=}v \land \mathsf{Q}^*_{\mathsf{ro}} \land \mathsf{Q}^*_{\mathsf{wo}} \land \mathsf{Q}_{\mathsf{sc}}$ | $\mathsf{W}^{\mathsf{sc}}xv$
$\psi[M/x][(M{=}v \land \mathsf{Q}^x_{\mathsf{wo}})/\mathsf{Q}^x_{\mathsf{wo}}][(M{=}v \land \mathsf{Q}_{\mathsf{sc}})/\mathsf{Q}_{\mathsf{sc}}]$
$r := x^{\mathsf{sc}}$
$(v{=}r \lor x{=}r) \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{ro}}][\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}][\mathsf{ff}/\mathsf{Q}_{\mathsf{sc}}]$
$\mathsf{Q}^x_{\mathsf{wo}} \land \mathsf{Q}_{\mathsf{sc}}$ | $\mathsf{R}^{\mathsf{sc}}xv$
$v{=}r \Rightarrow \psi$
F^{sc}
$\psi[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{ro}}][\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}][\mathsf{ff}/\mathsf{Q}_{\mathsf{sc}}]$
$\mathsf{Q}^*_{\mathsf{ro}} \land \mathsf{Q}^*_{\mathsf{wo}} \land \mathsf{Q}_{\mathsf{sc}}$ | F^{sc}
ψ

Fig. 1. The Effect of Quiescence for Each Access Mode

Update the following rules from Fig. 1. (The change is similar for address calculation and if-introduction.)

[Todo: This is buggy. Need to enforce order for coherence/synchronization/dependency into a write and coherence/synchronization, but not dependency, into reads. Lack of read-read dependency is bad here. Note that the write rules should mention D—see the agda version of write.]

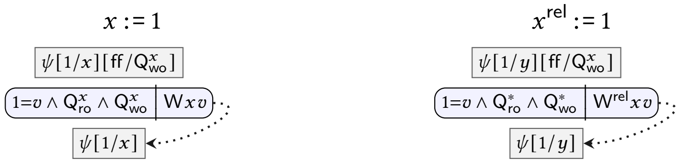
(w3) $\kappa(e) \equiv M{=}v \land \mathsf{Q}^{\mathsf{S}x}_{\mu}$,
(w4a) if $E \neq \emptyset$ then $\tau^D(\psi) \equiv \psi[M/x][(M{=}v \land \mathsf{Q}^{\mathsf{S}x}_{\mu})/\mathsf{Q}^{\mathsf{S}x}_{\mu}]$,
(w4b) if $E = \emptyset$ then $\tau^D(\psi) \equiv \psi[M/x][\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_{\mu}]$,
(r3) $\kappa(e) \equiv \mathsf{Q}^{\mathsf{L}x}_{\mu}$,
(r4a) if $e \in E \cap D$ then $\tau^D(\psi) \equiv v{=}r \Rightarrow \psi$,
(r4b) if $e \in E \setminus D$ then $\tau^D(\psi) \equiv (v{=}r \lor x{=}r) \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_{\mu}]$,
(r4c) if $E = \emptyset$ then $\tau^D(\psi) \equiv \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_{\mu}]$,
(f3) $\kappa(e) \equiv \mathsf{Q}^{\mathsf{F}x}_{\mu}$,
(f4a) if $E \neq \emptyset$ then $\tau^D(\psi) \equiv \psi$,
(f4b) if $E = \emptyset$ then $\tau^D(\psi) \equiv \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{F}x}_{\mu}]$.

The quiescence formulae indicate what must precede an event. For example, all preceding accesses must be ordered before a releasing write, whereas only writes on x must be ordered before a releasing read on x.

The quiescence substitutions update quiescence symbols in subsequent code. For subsequent independent code, w3 and r3 substitute false. In complete pomsets, we substitute true for . For example, we substitute ff for $\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rel}}$ in the independent case for a releasing write; this ensures that subsequent writes to x follow the releasing write in top-level pomsets. Similarly, we substitute ff for $\mathsf{Q}^{\mathsf{L}x}_{\mathsf{acq}}$ in the independent case for an acquiring write; this ensures that all subsequent accesses follow the acquiring read in top-level pomsets.

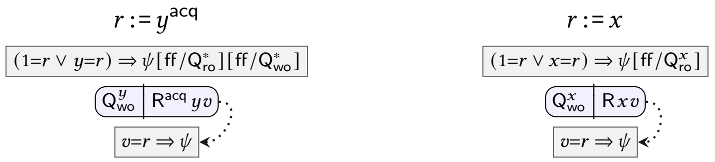
Fig. 1 shows the effect of quiescence for each access mode.

Example A.6. The definition enforces publication. Consider:


$x := 1$
$\psi[1/x][\mathsf{ff}/\mathsf{Q}^x_{\mathsf{wo}}]$
$1{=}v \wedge \mathsf{Q}^x_{\mathsf{ro}} \wedge \mathsf{Q}^x_{\mathsf{wo}}$
$\mathsf{W}xv$
$\psi[1/x]$
$x^{\mathsf{rel}} := 1$
$\psi[1/y][\mathsf{ff}/\mathsf{Q}^x_{\mathsf{wo}}]$
$1{=}v \wedge \mathsf{Q}^*_{\mathsf{ro}} \wedge \mathsf{Q}^*_{\mathsf{wo}}$
$\mathsf{W}^{\mathsf{rel}}xv$
$\psi[1/y]$

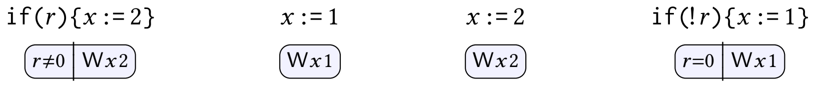
Since $\mathsf{Q}^*_{\mathsf{wo}}[\mathsf{ff}/\mathsf{Q}^x_{\mathsf{wo}}]$ is ff, we must introduce order to get a satisfiable precondition for $(\mathsf{W}y1)$.

Example A.7. The definition enforces subscription. Consider:

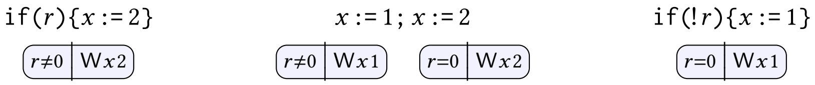

$r := y^{\mathsf{acq}}$
$(1{=}r \vee y{=}r) \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{ro}}][\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}]$
$\mathsf{Q}^y_{\mathsf{wo}}$
$\mathsf{R}^{\mathsf{acq}}yv$
$v{=}r \Rightarrow \psi$
$r := x$
$(1{=}r \vee x{=}r) \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^x_{\mathsf{ro}}]$
$\mathsf{Q}^x_{\mathsf{wo}}$
$\mathsf{R}xv$
$v{=}r \Rightarrow \psi$

Since $\mathsf{Q}^x_{\mathsf{wo}}[\mathsf{ff}/\mathsf{Q}^*_{\mathsf{wo}}]$ is ff, we must introduce order to get a satisfiable precondition for $(\mathsf{W}y1)$.

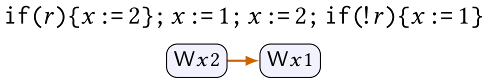
Example A.8. Even in its logical form, s6b′ is incompatible with the ability to strengthen preconditions using augment closure, which is allowed in [Jagadeesan et al. 2020]. Consider the following.


$\mathtt{if}(r)\{x := 2\}$
$r{\neq}0$ | $\mathsf{W}x2$
$x := 1$
$\mathsf{W}x1$
$x := 2$
$\mathsf{W}x2$
$\mathtt{if}(!r)\{x := 1\}$
$r{=}0$ | $\mathsf{W}x1$

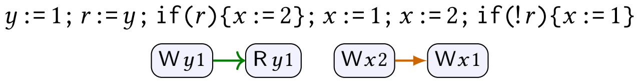
If $r{=}0$ then x is 1, 2, 1. If $r{\neq}0$ then x is 2, 1, 2. Augmenting the middle preconditions and then using sequential composition, we have:


$\mathtt{if}(r)\{x := 2\}$
$r{\neq}0$ | $\mathsf{W}x2$
$x := 1;\ x := 2$
$r{\neq}0$ | $\mathsf{W}x1$
$r{=}0$ | $\mathsf{W}x2$
$\mathtt{if}(!r)\{x := 1\}$
$r{=}0$ | $\mathsf{W}x1$

Note that s6b′ does not require any order between the two writes of the middle pomset. Merging left and right, we have:

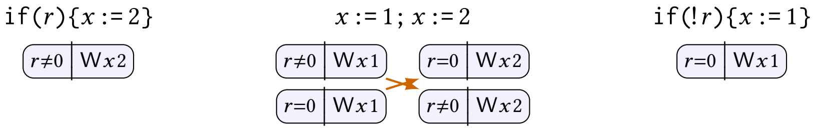

$\mathtt{if}(r)\{x := 2\};\ x := 1;\ x := 2;\ \mathtt{if}(!r)\{x := 1\}$
$\mathsf{W}x2$
$\mathsf{W}x1$

As shown by the following single-threaded code, allowing this outcome would violate DRF-SC.

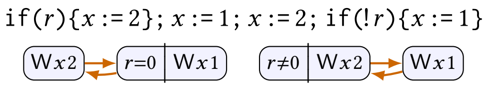

$y := 1;\ r := y;\ \mathtt{if}(r)\{x := 2\};\ x := 1;\ x := 2;\ \mathtt{if}(!r)\{x := 1\}$
$\mathsf{W}y1$
$\mathsf{R}y1$
$\mathsf{W}x2$
$\mathsf{W}x1$

This is one reason that we use *weakest* preconditions, rather than preconditions.

The same problem does not occur due to if-introduction (at least not for complete pomsets, where you need to have termination being a tautology, so you can't arbitrarily choose to partition $\Omega \neq \mathsf{tt}$:


$\mathtt{if}(r)\{x := 2\}$
$r{\neq}0$ | $\mathsf{W}x2$
$x := 1;\ x := 2$
$r{\neq}0$ | $\mathsf{W}x1$
$r{=}0$ | $\mathsf{W}x2$
$r{=}0$ | $\mathsf{W}x1$
$r{\neq}0$ | $\mathsf{W}x2$
$\mathtt{if}(!r)\{x := 1\}$
$r{=}0$ | $\mathsf{W}x1$

Merging left and right, we have


$\mathtt{if}(r)\{x := 2\};\ x := 1;\ x := 2;\ \mathtt{if}(!r)\{x := 1\}$
$\mathsf{W}x2$
$r{=}0$ | $\mathsf{W}x1$
$r{\neq}0$ | $\mathsf{W}x2$
$\mathsf{W}x1$

A.10 Is Coherence/Delay Compatible with If-Introduction and Dead-Write-Removal?

[Todo: Flesh this out.]

With if-introduction, the following equation should hold:

$$\begin{aligned}&[\![\texttt{if}(r)\{x := 2\};\ x := 1;\ x := 2;\ \texttt{if}(!r)\{x := 1\};\ x := 3]\!]\\ =\ &[\![\texttt{if}(!r)\{x := 1\};\ x := 2;\ x := 1;\ \texttt{if}(r)\{x := 2\};\ x := 3]\!]\end{aligned}$$

Using dead write removal, these can be refined, respectively, to:

$$\begin{aligned}&[\![x := 1;\ x := 2;\ x := 3]\!]\\ \neq\ &[\![x := 2;\ x := 1;\ x := 3]\!]\end{aligned}$$

What has become of coherence?

B LOWERING PwT-MCA TO ARM

For simplicity, we restrict to top-level parallel composition.

B.1 Arm executions

Our description of Arm8 follows Alglave et al. [2021], adapting the notation to our setting.

Definition B.1. An *Arm8 execution graph*, G, is tuple $(E, \lambda, \mathsf{poloc}, \mathsf{lob})$ such that

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

The definition of lob is complex. Comparing with our definition of sequential composition, it is sufficient to note that lob includes

(L1) read-write dependencies, required by S3,
(L2) synchronization delay of $\ltimes_{\mathsf{sync}}$, required by I6a,
(L3) sc access delay of $\bowtie_{\mathsf{sc}}$, required by I6a,
(L4) write-write and read-to-write coherence delay of $\bowtie_{\mathsf{co}}$, required by I6a,

and that lob does *not* include

(L5) read-read control dependencies, required by S3,
(L6) write-to-read order of rf, required by M7c,
(L7) write-to-read coherence delay of $\bowtie_{\mathsf{co}}$, required by I6a.

Definition B.2. Execution G is $(\mathsf{co}, \mathsf{rf}, \mathsf{gcb})$*-valid*, under *External Global Consistency* (EGC) if

(A5) $\mathsf{co} \subseteq E \times E$, is a per-location total order on writes, capturing *coherence*,
(A6) $\mathsf{rf} \subseteq E \times E$, is a relation, capturing *reads-from*, such that
  (A6a) rf is surjective and injective relation on $\{e \in E \mid \lambda(e) \text{ is a } \mathsf{read}\}$,
  (A6b) if $d \xrightarrow{\mathsf{rf}} e$ then $\lambda(d)$ matches $\lambda(e)$,
  (A6c) $\mathsf{poloc} \cup \mathsf{co} \cup \mathsf{rf} \cup \mathsf{fr}$ is acyclic, where $e \xrightarrow{\mathsf{fr}} c$ if $e \xleftarrow{\mathsf{rf}} d \xrightarrow{\mathsf{co}} c$, for some d,
(A7) $\mathsf{gcb} \supseteq (\mathsf{co} \cup \mathsf{rf})$ is a linear order such that
  (A7a) if $d \xrightarrow{\mathsf{rf}} e$ and $\lambda(c)$ blocks $\lambda(e)$ then either $c \xrightarrow{\mathsf{gcb}} d$ or $e \xrightarrow{\mathsf{gcb}} c$,
  (A7b) if $e \xrightarrow{\mathsf{lob}} c$ then either $e \xrightarrow{\mathsf{gcb}} c$ or $(\exists d)\ d \xrightarrow{\mathsf{rf}} e$ and $d \xrightarrow{\mathsf{poloc}} e$ but not $d \xrightarrow{\mathsf{lob}} c$.

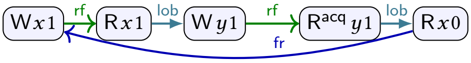
Execution G is $(\mathsf{co}, \mathsf{rf}, \mathsf{cb})$*-valid* under *External Consistency* (EC) if

(A5) and (A6), as for EGC,
(A8) $\mathsf{cb} \supseteq (\mathsf{co} \cup \mathsf{lob})$ is a linear order such that if $d \xrightarrow{\mathsf{rf}} e$ then either

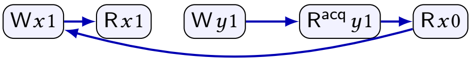
(A8a) $d \xrightarrow{\text{cb}} e$ and if $\lambda(c)$ blocks $\lambda(e)$ then either $c \xrightarrow{\text{cb}} d$ or $e \xrightarrow{\text{cb}} c$, or
(A8b) $d \xleftarrow{\text{cb}} e$ and $d \xrightarrow{\text{poloc}} e$ and $(\nexists c)$ $\lambda(c)$ blocks $\lambda(e)$ and $d \xrightarrow{\text{poloc}} c \xrightarrow{\text{poloc}} e$.

Alglave et al. [2021] show that EGC and EC are both equivalent to the standard definition of Arm8. They explain EGC and EC using the following example, which is allowed by Arm8.[5]

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

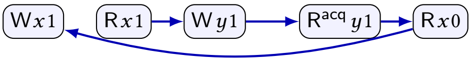


EGC drops lob-order in the first thread using A7b, since $(\mathsf{W}x1)$ is not lob-ordered before $(\mathsf{W}y1)$.



(gcb)

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



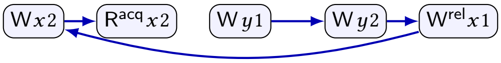
(cb)

B.2 Lowering PwT-MCA1 to Arm

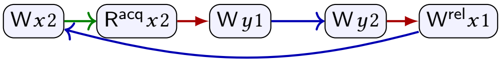
The optimal lowering for Arm8 is unsound for PwT-MCA$_1$. The optimal lowering maps relaxed access to `ldr`/`str` and non-relaxed access to `ldar`/`stlr` [Podkopaev et al. 2019]. In this section, we consider a suboptimal strategy, which lowers non-relaxed reads to (`dmb.sy`; `ldar`). Significantly, we retain the optimal lowering for relaxed access. In the next section we recover the optimal lowering by adopting an alternative semantics for M7c and I6a.

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

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



(gcb)

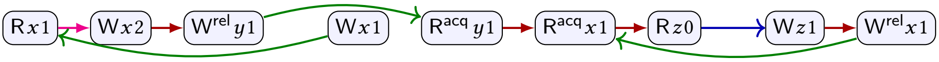


(<)

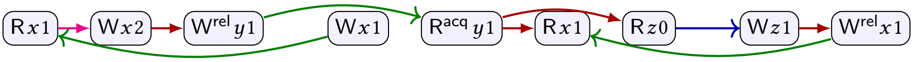
This attempted execution is allowed by Arm8, but disallowed by our semantics.

If the read of x in the execution above is changed from acquiring to relaxed, then our semantics allows the gcb execution, using the independent case for the read and satisfying the precondition of $(\mathsf{W}y1)$ by prepending $(\mathsf{W}x2)$. It may be tempting, therefore, to adopt a strategy of *downgrading* acquires in certain cases. Unfortunately, it is not possible to do this locally without invalidating important idioms such as publication. For example, consider that $(\mathsf{R}^{\mathsf{ra}}x1)$ is *not* possible for the second thread in the following attempted execution, due to publication of $(\mathsf{W}x2)$ via y:

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



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



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

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

Theorem B.3. *Suppose G_1 is $(\mathsf{co}_1, \mathsf{rf}_1, \mathsf{gcb}_1)$-valid for S under the suboptimal lowering that maps non-relaxed reads to* (`dmb.sy; ldar`)*. Then there is a top-level pomset $P_2 \in [\![S]\!]$ such that $E_2 = E_1$, $\lambda_2 = \lambda_1$, $\mathsf{rf}_2 = \mathsf{rf}_1$, and $\leq_2 = \mathsf{gcb}_1$.*

Proof. First, we establish some lemmas about Arm8.

Lemma B.4. *Suppose G is $(\mathsf{co}, \mathsf{rf}, \mathsf{gcb})$-valid. Then $\mathsf{gcb} \supseteq \mathsf{fr}$.*

Proof. Using the definition of fr from A6c, we have $e \xleftarrow{\mathsf{rf}} d \xrightarrow{\mathsf{co}} c$, and therefore $\lambda(c)$ blocks $\lambda(e)$. Applying A7a, we have that either $c \xrightarrow{\mathsf{gcb}} d$ or $e \xrightarrow{\mathsf{gcb}} c$. Since gcb includes co, we have $d \xrightarrow{\mathsf{gcb}} c$, and therefore it must be that $e \xrightarrow{\mathsf{gcb}} c$. □

Lemma B.5. *Suppose G is $(\mathsf{co}, \mathsf{rf}, \mathsf{gcb})$-valid and $c \xrightarrow{\mathsf{poloc}} e$, where $\lambda(c)$ blocks $\lambda(e)$. Then $c \xrightarrow{\mathsf{gcb}} e$.*

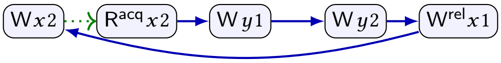
Proof. By way of contradiction, assume $e \xrightarrow{\mathsf{gcb}} c$. If $c \xrightarrow{\mathsf{rf}} e$ then by A7 we must also have $c \xrightarrow{\mathsf{gcb}} e$, contradicting the assumption that gcb is a total order. Otherwise that there is some $d \neq c$ such that $d \xrightarrow{\mathsf{rf}} e$, and therefore $d \xrightarrow{\mathsf{gcb}} e$. By transitivity, $d \xrightarrow{\mathsf{gcb}} c$. By the definition of fr, we have $e \xrightarrow{\mathsf{fr}} c$. But this contradicts A6c, since $c \xrightarrow{\mathsf{poloc}} e$. □

We show that all the order required in the pomset is also required by Arm8. M7b holds since cb_1 is consistent with co_1 and fr_1. As noted above, lob includes the order required by S3 and I6a. We need only show that the order removed from A7b can also be removed from the pomset. In order for A7b to remove order from e to c, we must have $d \xrightarrow{\mathsf{rf}} e$ and $d \xrightarrow{\mathsf{poloc}} e$ but not $d \xrightarrow{\mathsf{lob}} c$. Because of our suboptimal lowering, it must be that e is a relaxed read; otherwise the `dmb.sy` would require $d \xrightarrow{\mathsf{lob}} c$. Thus we know that I6a does not require order from e to c. By chaining R4b and W4, any dependence on the read can by satisfied without introducing order in S3. □

B.3 Lowering PwT-MCA2 to Arm

We can achieve optimal lowering for Arm by weakening the semantics of sequential composition slightly. In particular, we must lose M7c, which states that $d \xrightarrow{\mathsf{rf}} e$ implies $d < e$. Revisiting the example in the last subsection, we essentially mimic the EC characterization:

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



(cb)

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

We first show that EC-validity is unchanged if we assume $\mathsf{cb} \supseteq \mathsf{fr}$:

Lemma B.6. *Suppose G is EC-valid via $(\mathsf{co}, \mathsf{rf}, \mathsf{cb})$. Then there a permutation cb' of cb such that G is EC-valid via $(\mathsf{co}, \mathsf{rf}, \mathsf{cb}')$ and $\mathsf{cb}' \supseteq \mathsf{fr}$, where fr is defined in A6c.*

Proof. Suppose $e \xrightarrow{\mathsf{fr}} c$. By definition of fr, $e \xleftarrow{\mathsf{rf}} d \xrightarrow{\mathsf{co}} c$, for some d. We show that either (1) $e \xrightarrow{\mathsf{cb}} c$, or (2) $c \xrightarrow{\mathsf{cb}} e$ and we can reverse the order in cb' to satisfy the requirements.

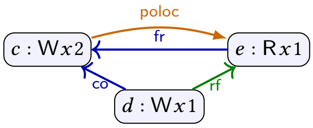
If A8a applies to $d \xrightarrow{\mathsf{rf}} e$, then $e \xrightarrow{\mathsf{cb}} c$, since it cannot be that $c \xrightarrow{\mathsf{co}} d$.

Suppose A8b applies to $d \xrightarrow{\mathsf{rf}} e$ and c is from a different thread than e. Because it is a different thread, we cannot have $e \xrightarrow{\mathsf{lob}} c$, and therefore we can choose $c \xrightarrow{\mathsf{cb}} e$ in cb'.

Suppose A8b applies to $d \xrightarrow{\mathsf{rf}} e$ and c is from the same thread as e. Applying A6c to $e \xrightarrow{\mathsf{fr}} c$, it cannot be that $c \xrightarrow{\mathsf{poloc}} e$. Since poloc is a per-thread-and-per-location total order, it must be that $e \xrightarrow{\mathsf{poloc}} c$. Applying A4a, we cannot have $e \xrightarrow{\mathsf{lob}} c$, and therefore we can choose $c \xrightarrow{\mathsf{cb}} e$ in cb'. □

Here is a contradictory non-example illustrating the last case of the proof:

$x := 2\,;\, r := x \parallel x := 1$



Theorem B.7. *Suppose G_1 is* EC*-valid for S via* $(\mathsf{co}_1, \mathsf{rf}_1, \mathsf{cb}_1)$ *and that* $\mathsf{cb}_1 \supseteq \mathsf{fr}_1$*. Then there is a top-level pomset $P_2 \in [\![S]\!]$ such that $E_2 = E_1$, $\lambda_2 = \lambda_1$, $\mathsf{rf}_2 = \mathsf{rf}_1$, and $\leq_2 = \mathsf{cb}_1$.*

Proof. We show that all the order required in the pomset is also required by Arm8. M7b holds since cb_1 is consistent with co_1 and fr_1. s6b follows from A8b. As noted above, lob includes the order required by s3 and s6a. □

C LDRF-SC FOR PwT-MCA

[Todo: Remove this section?]

In this appendix, we establish a DRF-SC for PwT-MCA$_2$. We prove an *external* result, where the notion of *data-race* is independent of the semantics itself. Since every PwT-MCA$_2$ is also a PwT-MCA$_1$, the result also applies there. Our result is also *local*. Using Dolan et al.'s [2018] notion of *Local Data Race Freedom (LDRF)*.

We do not address PwT-C11. The internal DRF-SC result for C11 [Batty 2015] does not rely on dependencies and thus applies to PwT-C11. In internal DRF-SC, data-races are defined using the semantics of the language itself. Using the notion of dependency defined here, it should be possible to prove an stronger external result for C11, similar to that of [Lahav et al. 2017]—we leave this as future work.

Jagadeesan et al. [2020] prove LDRF-SC for Pomsets with Preconditions (PwP). PwT-MCA generalizes PwP to account for sequential composition. Most of the machinery of LDRF-SC, however, has little to do with sequential semantics. Thus, we have borrowed heavily from the text of [Jagadeesan et al. 2020]; indeed, we have copied directly from the LaTeX source, which is publicly available. We indicate substantial changes or additions using a change-bar on the right.

There are several changes:

- PwP imposes several conditions that we have dropped: *consistency*, *causal strengthening*, *downset closure* (see §A.2).
- PwP allows preconditions that are stronger than the weakest precondition.
- PwP imposes M7c (rf implies $<$) and thus is similar to PwT-MCA$_1$. PwT-MCA$_2$ is a weaker model that is new to this paper.
- PwP did not provide an accurate account of program order for merged actions. We use Lemma 7.2 to correct this deficiency.

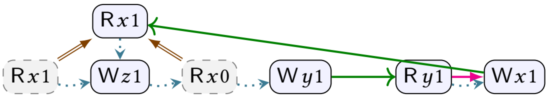
The first two items require us to define gen differently, below.

The result requires that locations are properly initialized. We assume a sufficient condition: that programs have the form "$x_1 := v_1\,;\ \cdots x_n := v_n\,;\, S$" where every location mentioned in S is some x_i. To simplify the definition of *happens-before*, we ban fences and RMWs.

To state the theorem, we require several technical definitions. The reader unfamiliar with [Dolan et al. 2018] may prefer to skip to the examples in the proof sketch, referring back as needed.

Program Order. Let $[\![\cdot]\!]^{\mathsf{po}}_{\mathsf{mca2}}$ be defined by applying the construction of Lemma 7.2 to $[\![\cdot]\!]_{\mathsf{mca2}}$. We consider only *complete* pomsets. For these, we derive program order on compound events as follows. By Lemma 7.4, if there is a compound event e, then there is a phantom event $c \in \pi^{-1}(e)$ such that $\kappa(c)$ is a tautology. If there is exactly one tautology, we identify e with c in program order. If there is more than one tautology, Lemma C.1, below, shows that it suffices to pick an arbitrary one—we identify e with the $c \in \pi^{-1}(e)$ that is minimal in program order. For example, consider JMM causality test case 2, with an added write to z:

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



(‡‡)

Data Race. Data races are defined using *program* order (po), not *pomset* order ($<$).

Because we ban fences and RMWs, we can adopt the simplest definition of *synchronizes-with* (sw): Let $d \xrightarrow{\mathsf{sw}} e$ exactly when d fulfills e, d is a release, e is an acquire, and $\neg(d \xrightarrow{\mathsf{po}} e)$.

Let $\mathsf{hb} = (\mathsf{po} \cup \mathsf{sw})^{+}$ be the *happens-before* relation.

Let $L \subseteq \mathcal{X}$ be a set of locations. We say that d *has an L-race with* e (notation $d \overset{L}{\rightsquigarrow} e$) when (1) at least one is relaxed, (2) at least one is a write, (3) they access the same location in L, and (4) they are unordered by hb: neither $d \xrightarrow{\mathsf{hb}} e$ nor $e \xrightarrow{\mathsf{hb}} d$.

Generators. We say that $P' \in \nabla(\mathcal{P})$ if there is some $P \in \mathcal{P}$ such that P is *complete* (Def. 5.1) and P' is a *downset* of P (Def. A.3).

Let P be *augmentation-minimal* in $\mathcal{P}$ if $P \in \mathcal{P}$ and there is no $P{\neq}P'{\in}\mathcal{P}$ such that P augments P'.

Let $\mathsf{gen}[\![S]\!] = \{P \in \nabla[\![S]\!]^{\mathsf{po}}_{\mathsf{mca2}} \mid P \text{ is augmentation-minimal in } \nabla[\![S]\!]^{\mathsf{po}}_{\mathsf{mca2}}\}$.

Extensions. We say that P' *S-extends* P if $P \neq P' \in \mathsf{gen}[\![S]\!]$ and P is a downset of P'.

Similarity. We say that P' *is e-similar to* P if they differ at most in (1) pomset order adjacent to e, (2) the value associated with event e, if it is a read, and (3) the addition and removal of read events po-after e.

Stability. We say that P is *L-stable in S* if (1) $P \in \mathsf{gen}[\![S]\!]$, (2) P is po-convex (nothing missing in program order), (3) there is no S-extension of P with a *crossing* L-race: that is, there is no $d \in E$, no P' S-extending P, and no $e \in E' \setminus E$ such that $d \overset{L}{\rightsquigarrow} e$. The empty pomset is L-stable.

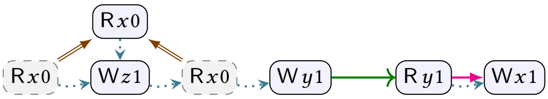
Sequentiality. Let $\lessdot_L = <_L \cup \mathsf{po}$, where $<_L$ is the restriction of $<$ to events that access locations in L. We say that P' is *L-sequential after* P if (1) P' is po-convex, (2) $\lessdot_L$ is acyclic in $E' \setminus E$.

Simplicity. We say that P' is *L-simple after* P if all of the events in $E' \setminus E$ that access locations in L are *simple* (Def. 7.1).

Lemma C.1. *Suppose $P' \in \mathsf{gen}[\![S]\!]$ and P is L-sequential after P. Let P'' be the restriction of P' that is L-simple after P (throwing out compound L-events after P). Then $P'' \in \mathsf{gen}[\![S]\!]$.*

As a negative example, note that (‡‡) is not L-sequential—in fact there is no execution of the program that results in the simple events of (‡‡): without merging the reads, there would be a dependency $(\mathsf{R}x1) \rightarrow (\mathsf{W}y1)$. L-sequential executions of this code must read 0 for x:

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



Theorem C.2. *Let P be L-stable in S. Let P' be a S-extension of P that is L-sequential after P. Let P'' be a S-extension of P' that is po-convex, such that no subset of E'' satisfies these criteria. Then either (1) P'' is L-sequential and L-simple after P or (2) there is some S-extension P''' of P' and some $e \in (E'' \setminus E')$ such that (a) P''' is e-similar to P'', (b) P''' is L-sequential and L-simple after P, and (c) $d \overset{L}{\rightsquigarrow} e$, for some $d \in (E'' \setminus E)$.*

The theorem provides an inductive characterization of *Sequential Consistency for Local Data-Race Freedom (SC-LDRF)*: Any extension of a L-stable pomset is either L-sequential, or is e-similar to a L-sequential extension that includes a race involving e.

Proof Sketch. We show L-sequentiality. L-simplicity then follows from Lemma C.1.

In order to develop a technique to find P''' from P'', we analyze pomset order in generation-minimal top-level pomsets. First, we note that $<_*$ (the transitive reduction $<$) can be decomposed into three disjoint relations. Let $\mathsf{ppo} = (<_* \cap \mathsf{po})$ denote *preserved* program order, as required by sequential composition and conditional. The other two relations are cross-thread subsets of $(<_* \setminus \mathsf{po})$: rfe (reads-from-external) orders writes before reads, satisfying P6a and P6b; cae (coherence-after-external) orders read and write accesses before writes, satisfying M7b. (Within a thread, s6a and s6b induce order that is included in ppo.)

Using this decomposition, we can show the following.

Lemma C.3. *Suppose $P'' \in \mathit{gen}\llbracket S \rrbracket$ has an external read $d \xrightarrow{\mathsf{rf}''} e$ that is maximal in $(\mathsf{ppo} \cup \mathsf{rfe})$. Further suppose that there another write d' that could fulfill e. Then there exists an e-similar P''' with $d' \xrightarrow{\mathsf{rf}'''} e$ such that $P''' \in \mathit{gen}\llbracket S \rrbracket$.*

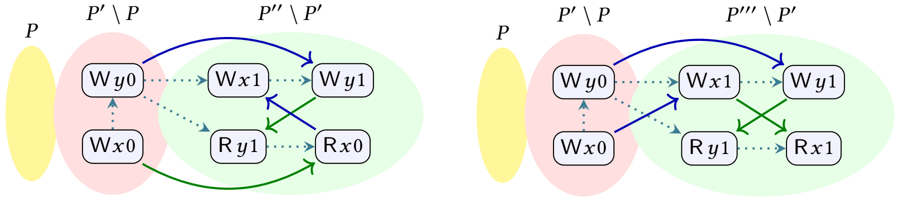
The proof of the lemma follows an inductive construction of $\mathrm{gen}\llbracket S \rrbracket$, starting from a large set with little order, and pruning the set as order is added: We begin with all pomsets generated by the semantics without imposing the requirements of fulfillment (including only ppo). We then prune reads which cannot be fulfilled, starting with those that are minimally ordered.

We can prove a similar result for $(\mathsf{po} \cup \mathsf{rfe})$-maximal read and write accesses.

Turning to the proof of the theorem, if P'' is L-sequential after P, then the result follows from (1). Otherwise, there must be a $\lessdot_L$ cycle in P'' involving all of the actions in $(E'' \setminus E')$: If there were no such cycle, then P'' would be L-sequential; if there were elements outside the cycle, then there would be a subset of E'' that satisfies these criteria.

If there is a $(\mathsf{po} \cup \mathsf{rfe})$-maximal access, we select one of these as e. If e is a write, we reverse the outgoing order in cae; the ability to reverse this order witnesses the race. If e is a read, we switch its fulfilling write to a "newer" one, updating cae; the ability to switch witnesses the race. For example, for P'' on the left below, we choose the P''' on the right; e is the read of x, which races with $(\mathsf{W}x1)$.

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

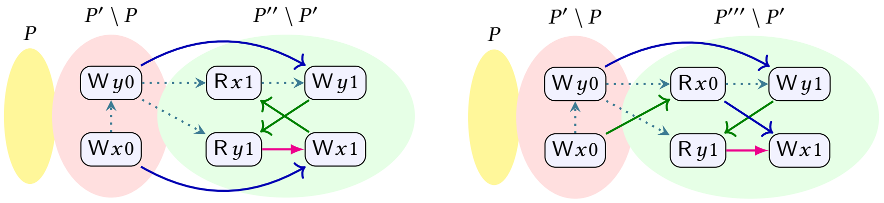


It is important that e be $(\mathsf{po} \cup \mathsf{rfe})$-maximal, not just $(\mathsf{ppo} \cup \mathsf{rfe})$-maximal. The latter criterion would allow us to choose e to be the read of y, but then there would be no e-similar pomset: if an execution reads 0 for y then there is no read of x, due to the conditional.

In the above argument, it is unimportant whether e reads-from an internal or an external write; thus the argument applies to PwT-MCA$_2$ and PwT-MCA$_1$ as it does for PwT-MCA$_1$.

If there is no $(\mathsf{po} \cup \mathsf{rfe})$-maximal access, then all cross-thread order must be from rfe. In this case, we select a $(\mathsf{ppo} \cup \mathsf{rfe})$-maximal read, switching its fulfilling write to an "older" one. If there are several of these, we choose one that is po-minimal. As an example, consider the following; once again, e is the read of x, which races with $(\mathsf{W}x1)$.

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



This example requires $(\mathsf{W}x0)$. Proper initialization ensures the existence of such "older" writes. □

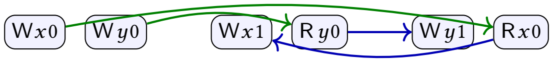
D PwT-MCA: ADDITIONAL EXAMPLES

This appendix includes additional examples. They all apply equally to PwT-MCA$_1$ and PwT-MCA$_2$. Many of these are taken directly from [Jagadeesan et al. 2020]; see there for further discussion.

D.1 Buffering

Store buffering is allowed, as required by TSO.

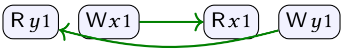
$$x := 0;\ y := 0;\ (x := 1;\ r := y \parallel y := 1;\ r := x)$$



(SB)

Load buffering is allowed, as required by Arm8.

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

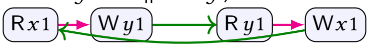


(LB)

D.2 Thin-Air

Thin air is disallowed. [Pugh 2004, TC4]:

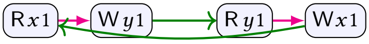
$$y := x \parallel r := y;\ x := r$$



(OOTA1)

The control variant ([Pugh 2004, TC13]) is also disallowed:

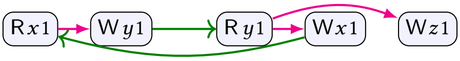
$$\mathtt{if}(x)\{y := 1\} \parallel \mathtt{if}(y)\{x := 1\}$$



(OOTA2)

[Jagadeesan et al. 2020, §2]

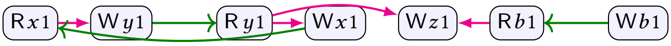
$$y := x \parallel r := y;\ \mathtt{if}(r)\{x := r;\ z := r\}\,\mathtt{else}\,\{x := 2\}$$



(OOTA3)

[Jeffrey and Riely 2019, §8] and [Jagadeesan et al. 2020, §6]:

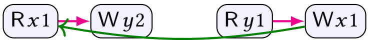
$$y := x \parallel r := y;\ \mathtt{if}(b)\{x := r;\ z := r\}\,\mathtt{else}\,\{x := 1\} \parallel b := 1$$



(OOTA4)

[Svendsen et al. 2018, RNG] is disallowed since there is no write to fulfill $(\mathsf{R}y1)$.

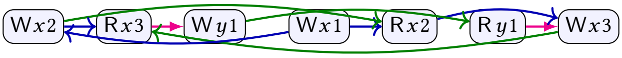
$$(y := x{+}1 \parallel x := y)$$



(OOTA6)

OOTA7 is allowed by PS, but not WEAKESTMO [Chakraborty and Vafeiadis 2019, Fig. 3]:

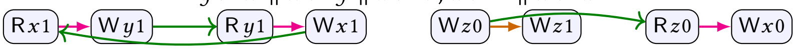
$$x := 2;\ \texttt{if}(x{\neq}2)\{y := 1\} \parallel x := 1;\ r := x;\ \texttt{if}(y)\{x := 3\}$$



(OOTA7)

OOTA4 is similar to TC5 [Pugh 2004]:

$$y := x \parallel x := y \parallel z := 0;\ z := 1 \parallel x := z$$



(TC5)

The justification for forbidding this execution states:

> values are not allowed to come out of thin air, even if there are other executions in which the thin-air value would have been written to that variable by some not out-of-thin air means.

OOTA4 is an interesting border case, since it is allowed by speculative models (§A.1).

[Todo: What's the point?] We presented two examples of thin-air behavior involving address calculation in §9.4. The justification for TC12 states:

> Since no other thread accesses [either [0] or [1]], the code for [the second] thread should be equivalent to:
>
> $$r := y;\ [r] := 0;\ (s := \texttt{if}(r{=}0)\{0\}\,\texttt{else}\,\{1\});\ x := s;$$
>
> With this code, it is clear that this is the same situation as test 4.

[Jagadeesan et al. 2020, §6]:

> Boehm's [2019] RFUB example presents another potential form of OOTA behavior. Our analysis shows that there is no OOTA behavior in RFUB, only a false dependency:
>
> $$[\![r := y;\ x := r]\!] \not\supseteq [\![r := y;\ \texttt{if}(r{\neq}1)\{z := 1;\ r := 1\};\ x := r]\!] \qquad \text{(RFUB)}$$
>
> The left command is half of OOTA3 ($y := x$). The right command is dubbed RFUB, for *Register assignment From an Unexecuted Branch*. Boehm observes that in the context $x := y \parallel [-]$, these programs have different behaviors. Yet the OOTA example on the left never writes 1. Why should the unexecuted branch change that? Because of the conditional, the write to x in RFUB is independent of the read from y. It useful to considering the Hoare logic formulas satisfied by the two threads above: we have $\{\mathrm{tt}\}$ RFUB $\{x = 1\}$ for the right thread of RFUB, but not $\{\mathrm{tt}\}$ OOTA3 $\{x = 1\}$ for the right thread of OOTA3. The change in the thread from OOTA3 to RFUB is not a valid refinement under Hoare logic; thus, it is expected that RFUB may have additional behaviors.

RFUB New Constructor:

$$y := x \parallel r := y;\ \texttt{if}(r{=}\texttt{null})\{r := \texttt{new}\,\texttt{C}()\};\ x := r;\ r.\texttt{f}() \qquad \text{(RFUB-NC)}$$

This is similar to:

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

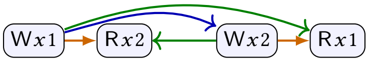
And different from the following, which is similar to TC18:

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

D.3 Coherence

The following execution is disallowed by fulfillment (M7a and M7b). It is also disallowed by C11 and Java.

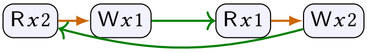
$$x := 1;\ r := x \parallel x := 2;\ s := x \tag{COH}$$



M7b requires that we order one write with respect to the other, either before the write or after the read (and therefore after the write). Suppose we pick 1 before 2, as shown. This satisfies M7b for (Rx2). But to satisfy the requirement for (Rx1) we must have either (Wx2) $<$ (Wx1) or (Rx1) $<$ (Wx2). Either way, we have a cycle.

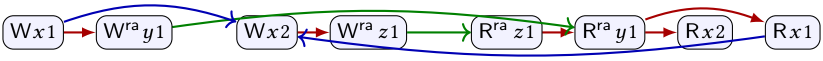
Our model is more coherent than Java, which permits the following:

$$r := x;\ x := 1 \parallel s := x;\ x := 2 \tag{TC16}$$



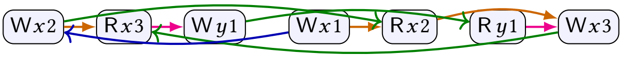
We also forbid the following, which Java allows:

$$x := 1;\ y^{\mathsf{rel}} := 1 \parallel x := 2;\ z^{\mathsf{rel}} := 1 \parallel r := z^{\mathsf{ra}};\ r := y^{\mathsf{ra}};\ r := x;\ r := x \tag{CO3}$$



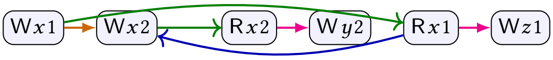
The following outcome is allowed by the promising semantics [Kang et al. 2017], but not in WEAKESTMO [Chakraborty and Vafeiadis 2019, Fig. 3]. We disallow it:

$$x := 2;\ \mathtt{if}(x{\neq}2)\{y := 1\} \parallel x := 1;\ r := x;\ \mathtt{if}(y)\{x := 3\} \tag{COH-CYC}$$



C11 includes read-read coherence between relaxed atomics in order to forbid the following. We do not order reads by intra-thread coherence, and this allow the following:

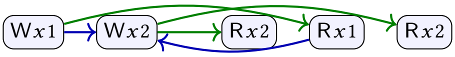
$$x := 1;\ x := 2 \parallel y := x;\ z := x \tag{CO2}$$



Here, the reader sees 2 then 1, although they are written in the reverse order.

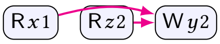
We also allow the following, similar execution:

$$x := 1;\ x := 2 \parallel r_1 := x;\ r_2 := x;\ r_3 := x;$$



Pugh [1999, §2.3] presented the following example to show that Java's original memory model required alias analysis to validate common subexpression elimination (CSE).

$$r_1 := x;\ r_2 := z;\ r_3 := x;\ \mathtt{if}(r_3{\leqslant}1)\{y = r_2\}$$



Coalescing the two read of x is obviously allowed if $z{\neq}x$. But if $z{=}x$, coalescing is only permitted because we do not include read-read pairs in $\bowtie_{\mathsf{co}}$ (§4.2):

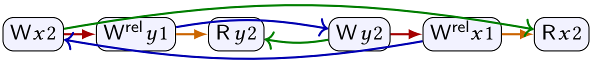
$$\bowtie_{\mathsf{co}} = \{(\mathsf{W}x, \mathsf{W}x),\ (\mathsf{R}x, \mathsf{W}x),\ (\mathsf{W}x, \mathsf{R}x)\}$$

C11 has read-read coherence, and therefore CSE is only valid up to alias analysis in C11.

D.4 RA

Is RC11 RA or SRA? Our model of RA is closer to strong RA (SRA) [Lahav and Boker 2020; Lahav et al. 2016], than C11. For example, C11 allows the following, which we disallow:

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

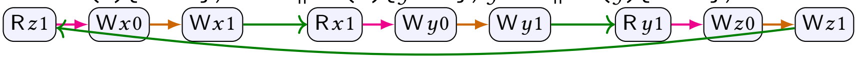


(SRA)

D.5 MCA

Here are a few litmus tests that distinguish MCA architectures from non-MCA architectures. MCA1 is an example of *write subsumption* [Pulte et al. 2018, §3]:

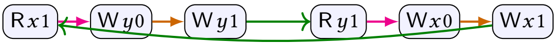
$$\texttt{if}(z)\{x := 0\};\ x := 1 \parallel \texttt{if}(x)\{y := 0\};\ y := 1 \parallel \texttt{if}(y)\{z := 0\};\ z := 1$$



(MCA1)

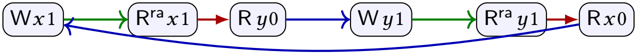
Two thread variant:

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



IRIW is allowed if all accesses are relaxed, but not if the initial reads are acquiring:

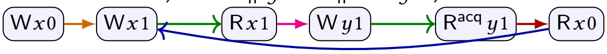
$$x := 1 \parallel r := x^{\mathsf{ra}};\ s := y \parallel y := 1 \parallel s := y^{\mathsf{ra}};\ r := x$$



(IRIW)

MCA2 is a simplified version of IRIW

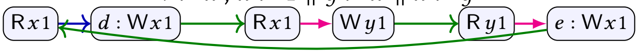
$$x := 0;\ x := 1 \parallel y := x \parallel r := y^{\mathsf{ra}};\ s := x$$



(MCA2)

[Flur et al. 2016] and [Lahav and Vafeiadis 2016, Fig. 4] discuss the following, which is not valid in Arm8, although it was valid under some earlier sketches of the model:

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



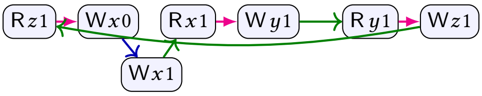
(MCA3)

These candidate executions are invalid, due to cycles.

D.6 Detour

The following example [Podkopaev et al. 2019, Ex. 3.7] is disallowed by IMM by including a detour relation. It is also disallowed by PS.

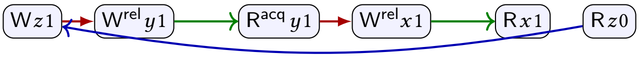
$$x := z{-}1;\ y := x \parallel x := 1 \parallel z := y$$



D.7 Read-Read Dependencies and Java Final Field Semantics Versus If-Closure

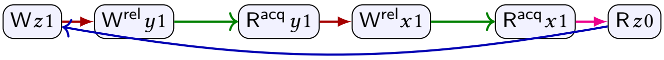
One might worry that the lack of read-read dependencies could cause DRF-SC to fail. For example, the following execution has a control dependency between the reads of the last thread, but this order is not enforced, neither by our model, nor Arm8.

$$z := 1;\ y^{\mathsf{rel}} := 1 \parallel r := y^{\mathsf{acq}};\ x^{\mathsf{rel}} := 1 \parallel \mathtt{if}(x)\{s := z\}$$



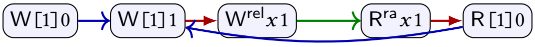
If the first read of the last thread is acquiring, then the execution is disallowed, since acquiring reads are ordered with respect to the reads that follow.

$$z := 1;\ y^{\mathsf{rel}} := 1 \parallel r := y^{\mathsf{acq}};\ x^{\mathsf{rel}} := 1 \parallel \mathtt{if}(x^{\mathsf{acq}})\{s := z\}$$



Arm8 enforces address dependencies between reads, but not control dependencies. To support case-analysis (AKA if-closure), we drop all dependencies between reads. This, in turn, invalidates Java's final field semantics.

$$(r := 1;\ [r] := 0;\ [r] := 1;\ x^{\mathsf{rel}} := r) \parallel (r := x^{\mathsf{acq}};\ s := [r]) \qquad \text{(ADDR2)}$$

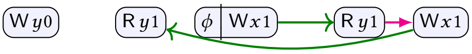


The acquire annotation is required to ensure publication. If address dependencies were enforced between reads then the acquire annotation could be dropped. However, the compiler would need to track address dependencies in order to ensure that case analysis did not convert them to control dependencies.

D.8 Local Invariant Reasoning and Value Range Analysis

We have already seen TC1 in §4.8, TC2 in §9.1 and TC6 in §7. Here is the complete program for TC6:

$$y := 0;\ (r := y;\ \mathtt{if}(r{=}0)\{x := 1\};\ \mathtt{if}(r{=}1)\{x := 1\}) \parallel (\mathtt{if}(x{=}1)\{y := 1\}) \qquad \text{(TC6)}$$

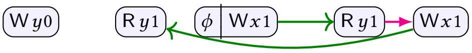


$$\phi = (1{=}r \lor 0{=}r) \Rightarrow (r{=}0 \lor r{=}1)$$

[Todo: Discuss.]

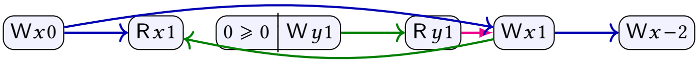
Here are some additional examples:

$$y := 0;\ (r := y;\ x := 1{+}r{*}r{-}r) \parallel (y := x) \qquad \text{(TC8)}$$

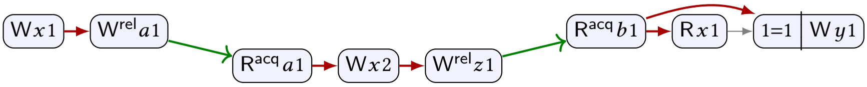


$$\phi = (1{=}r \lor 0{=}r) \Rightarrow 1{+}r{*}r{-}r = 1$$

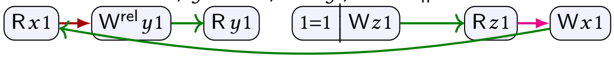
$$x := 0;\ (r := x;\ \mathtt{if}(r \geqslant 0)\{y := 1\} \parallel x := y \parallel x := -2) \qquad \text{(TC9)}$$



$$x := 1;\ a^{\mathsf{ra}} := 1;\ \mathtt{if}(z^{\mathsf{ra}})\{y := x\} \parallel \mathtt{if}(a^{\mathsf{ra}})\{x := 2;\ z^{\mathsf{ra}} := 1\} \qquad \text{(INTERNAL1)}$$

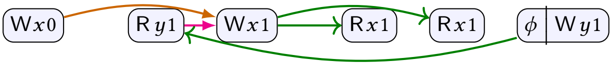


$$r := x;\ y^{\mathsf{ra}} := 1;\ s := y;\ z := s \parallel x := z \qquad \text{(INTERNAL2)}$$



Java Causality Test Case 18 asks that we justify the following execution:

$$x := 0;\ (x := y \parallel r := x;\ \mathtt{if}(r{=}0)\{x := 1\};\ s := x;\ y := s) \qquad \text{(TC18)}$$



Before we prefix $x := 0$, the precondition of $\mathsf{W}\,y\,1$ is:

$$\phi \equiv (1{=}r \lor x{=}r) \Rightarrow ([r{=}0 \land ((1{=}s \lor 1{=}s) \Rightarrow s{=}1)] \lor [r{\neq}0 \land ((1{=}s \lor x{=}s) \Rightarrow s{=}1)])$$

Simplifying:

$$\phi \equiv (1{=}r \lor x{=}r) \Rightarrow (r{=}0 \lor [r{\neq}0 \land ((1{=}s \lor x{=}s) \Rightarrow s{=}1)])$$

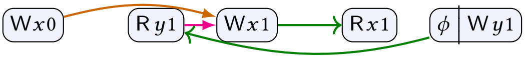
Prefixing $x := 0$:

$$\phi \equiv (1{=}r \lor 0{=}r) \Rightarrow (r{=}0 \lor [r{\neq}0 \land ((1{=}s \lor 0{=}s) \Rightarrow s{=}1)])$$

Drilling into the interesting part:

$$\phi \equiv 1{=}r \Rightarrow ((1{=}s \lor 0{=}s) \Rightarrow s{=}1)$$

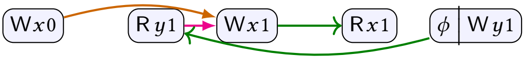
This is not a tautology. But we get one by coalescing s and r:



$$\phi \equiv 1{=}r \Rightarrow ((1{=}r \lor 0{=}r) \Rightarrow r{=}1)$$

TC20 splits the first thread of TC18:

$$x := 0;\ (x := y \mathrel{\Vert\!\!\rightarrow} r := x;\ \mathtt{if}(r{=}0)\{x := 1\});\ s := x;\ y := s \qquad \text{(TC20)}$$



Because we take register state from the right, the example is the same as for TC18 above.

TC17 replaces the condition $r{=}0$ by $r{\neq}1$ in TC18:

$$\phi \equiv (1{=}r \lor x{=}r) \Rightarrow ([r{\neq}1 \land ((1{=}s \lor 1{=}s) \Rightarrow s{=}1)] \lor [r{=}1 \land ((1{=}s \lor x{=}s) \Rightarrow s{=}1)])$$

Simplifying and prefixing $x := 0$:

$$\phi \equiv (1{=}r \lor 0{=}r) \Rightarrow (r{\neq}1 \lor [r{=}1 \land ((1{=}s \lor 0{=}s) \Rightarrow s{=}1)])$$

Again, we have:

$$\phi \equiv 1{=}r \Rightarrow ((1{=}s \lor 0{=}s) \Rightarrow s{=}1)$$

which is not a tautology. But we get one by coalescing s and r.

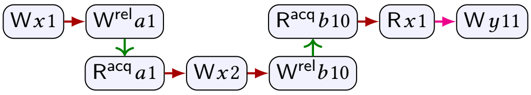
TC19 makes the same change for TC20, and follows for the same reason.

D.9 Commuting release and acquire

[Todo: Discuss.]

RA example. This is impossible, since Rx1 unfulfilled.

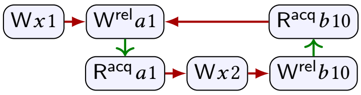
$$x := 1;\ a^{\mathsf{rel}} := 1;\ r := b^{\mathsf{acq}};\ s := x;\ y := r{+}s \parallel r := a^{\mathsf{acq}};\ x := 2;\ b^{\mathsf{rel}} := 10$$



$\mathsf{W}x1$
$\mathsf{W}^{\mathsf{rel}}a1$
$\mathsf{R}^{\mathsf{acq}}b10$
$\mathsf{R}x1$
$\mathsf{W}y11$
$\mathsf{R}^{\mathsf{acq}}a1$
$\mathsf{W}x2$
$\mathsf{W}^{\mathsf{rel}}b10$

If you swap the release and acquire, then it is impossible for the second thread to get in the middle.

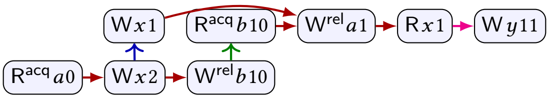
$$x := 1;\ r := b^{\mathsf{acq}};\ a^{\mathsf{rel}} := 1;\ \parallel r := a^{\mathsf{acq}};\ x := 2;\ b^{\mathsf{rel}} := 10$$



$\mathsf{W}x1$
$\mathsf{W}^{\mathsf{rel}}a1$
$\mathsf{R}^{\mathsf{acq}}b10$
$\mathsf{R}^{\mathsf{acq}}a1$
$\mathsf{W}x2$
$\mathsf{W}^{\mathsf{rel}}b10$

In this case, the following execution is possible:

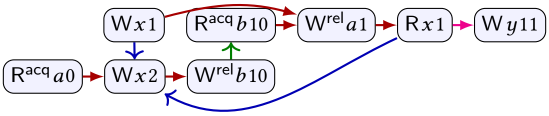
$$x := 1;\ r := b^{\mathsf{acq}};\ a^{\mathsf{rel}} := 1;\ s := x;\ y := r{+}s \parallel r := a^{\mathsf{acq}};\ x := 2;\ b^{\mathsf{rel}} := 10$$



$\mathsf{W}x1$
$\mathsf{R}^{\mathsf{acq}}b10$
$\mathsf{W}^{\mathsf{rel}}a1$
$\mathsf{R}x1$
$\mathsf{W}y11$
$\mathsf{R}^{\mathsf{acq}}a0$
$\mathsf{W}x2$
$\mathsf{W}^{\mathsf{rel}}b10$

But not:

$$x := 1;\ r := b^{\mathsf{acq}};\ a^{\mathsf{rel}} := 1;\ s := x;\ y := r{+}s \parallel r := a^{\mathsf{acq}};\ x := 2;\ b^{\mathsf{rel}} := 10$$



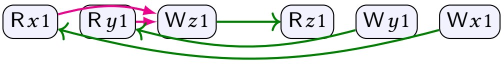
$\mathsf{W}x1$
$\mathsf{R}^{\mathsf{acq}}b10$
$\mathsf{W}^{\mathsf{rel}}a1$
$\mathsf{R}x1$
$\mathsf{W}y11$
$\mathsf{R}^{\mathsf{acq}}a0$
$\mathsf{W}x2$
$\mathsf{W}^{\mathsf{rel}}b10$

D.10 Sevcik examples

[Todo: Discuss.]

Cenciarelli et al. [2007, §7] example. (I incorrectly credit Sevčík and Aspinall [2008].)

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



Examples from [Sevčík and Aspinall 2008, §4.1] are interesting: Redundant write after read elimination:

```
|| lock m2; x=1; unlock m2
|| lock m1; x=2; unlock m1
|| lock m1; lock m2; r1=x; [x=r1;] r2=x; unlock m2; unlock m1 // [bracketed line removed]
```

Even without the write, r1 and r2 must see the same values, whereas JMM allows different values for the reads when the write is missing.

Redundant read after read elimination:

```
|| y=x
|| r2=y; if (r2==1){[r3=y]; x=r3}else{x=1} // [r3=r2]
```

Interesting case is left $\mathsf{W}x1$. Initially has predicate $r_3 = 1$. With read rule, we have $y = 1$. In read prefixing, we don't weaken. Instead we weaken with the read into r2.

$$\texttt{if}(r_2{=}1)\{r_3:=y;\ x:=r_3\} \qquad \texttt{if}(r_2{\neq}1)\{x:=1\}$$
$$\boxed{r_2{=}1 \mid \mathsf{R}\,y\,1}\quad \boxed{r_2{=}1\land y{=}1 \mid \mathsf{W}\,x\,1} \qquad \boxed{r_2{\neq}1 \mid \mathsf{W}\,x\,1}$$
$$\texttt{if}(r_2{=}1)\{r_3:=y;\ x:=r_3\}\,\texttt{else}\,\{x:=1\}$$
$$\boxed{r_2{=}1 \mid \mathsf{R}\,y\,1}\quad \boxed{(r_2{=}1\land y{=}1)\lor(r_2{\neq}1) \mid \mathsf{W}\,x\,1}$$
$$r_2:=y;\ \texttt{if}(r_2{=}1)\{r_3:=y;\ x:=r_3\}\,\texttt{else}\,\{x:=1\}$$
$$\boxed{\mathsf{R}\,y\,1}\quad \boxed{\mathsf{R}\,y\,1}\quad \boxed{(y{=}1\land y{=}1)\lor(y{\neq}1) \mid \mathsf{W}\,x\,1}$$

To ignore the second read, we use the "delay" trick that we used for JMM TC1, but this is fulfilled by a read rather than a write. In any case, the execution with $x = y = 1$ is allowed.

Roach Motel—all reads 1 impossible, but passible after swapping `r1=x` and `lock m`

```
|| lock m; x=1; unlock m
|| lock m; x=2; unlock m
|| r1=x; lock m; r2=z; if(r1==2){y=1}else{y=r2}; unlock m
|| z=y
```

So Question is whether you can read all 1 in

```
|| lock m; x=1; unlock m
|| lock m; x=2; unlock m
|| lock m; r1=x; r2=z; if(r1==2){y=1}else{y=r2}; unlock m
|| z=y
```

In any execution, we must have 1 before 2, or 2 before 1.

- If thread sees 2, then read x is 2.
- If thread sees 1, then read x is 1.

$$\texttt{if}(r_1{=}2)\{y:=1\}\,\texttt{else}\,\{y:=r_2\}$$
$$\boxed{r_1{=}2\lor(r_1{\neq}2\land r_2{=}1) \mid \mathsf{W}\,y\,1}$$
$$r_1:=x;\ r_2:=z;\ \texttt{if}(r_1{=}2)\{y:=1\}\,\texttt{else}\,\{y:=r_2\}$$
$$\boxed{\mathsf{R}\,x\,1}\quad \boxed{\mathsf{R}\,z\,1}\rightarrow\boxed{\mathsf{W}\,y\,1}$$

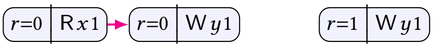
So impossible for y and z to be 1.

Irrelevant Read Introduction (can I read 1 for both y and z?)

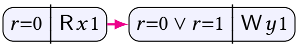
```
|| r=z; if(!r){if(x){y=1}}else{[s=x;]y=r}
|| x=1; z=y
```

$$\texttt{if}(!r)\{\texttt{if}(x)\{y:=1\}\} \qquad \texttt{if}(r)\{s:=x;\ y:=r\}$$
$$\boxed{r{=}0 \mid \mathsf{R}\,x\,1}\rightarrow\boxed{r{=}0 \mid \mathsf{W}\,y\,1} \qquad \boxed{r{\neq}0 \mid \mathsf{R}\,x\,1}\quad \boxed{r{=}1 \mid \mathsf{W}\,y\,1}$$
$$\texttt{if}(!r)\{\texttt{if}(x)\{y:=1\}\}\,\texttt{else}\,\{y:=r\}$$
$$\boxed{\mathsf{R}\,x\,1}\rightarrow\boxed{r{=}0\lor r{=}1 \mid \mathsf{W}\,y\,1}$$
$$z:=0;\ r:=z;\ \texttt{if}(!r)\{\texttt{if}(x)\{y:=1\}\}\,\texttt{else}\,\{y:=r\}$$
$$\boxed{\mathsf{W}\,z\,0}\quad \boxed{\mathsf{R}\,z\,1}\quad \boxed{\mathsf{R}\,x\,1}\rightarrow\boxed{0{=}0\lor 0{=}1 \mid \mathsf{W}\,y\,1}$$

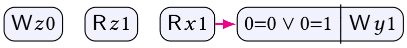
$\mathtt{if(!}r\mathtt{)\{if(}x\mathtt{)\{}y := 1\mathtt{\}\}} \qquad \mathtt{if(}r\mathtt{)\{}y := r\mathtt{\}}$



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



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



If z is initialized to 2, rather than 0, then the dependencies remain and both are disallowed. This relies crucially on the fact that par takes order from both sides.

D.11 SC Access

[Todo: Discuss.]

[Todo: Volatile read = full fence followed by acquire; Volatile write = release followed by full fence. But this is not enough on power to guarantee that all-volatile program has only SC executions. On power, release-acquire implemented with lwsync]

https://bugs.openjdk.java.net/browse/JDK-8262877

```
volatile int x, y;

Thread 1: x = 2; r1 = y // 0

Thread 2: y = 1

Thread 3: r2 = y; x = 1 // 1

Thread 4: r3 = x; r4 = x // 1,2
```

The state (r1,r2,r3,r4) = (0,1,1,2) is forbidden, as it violates sequential consistency. (You can show it by constructing the syncronization order that leads to this result and observing it is cyclic).

Current PPC code is one (the only?) platform that runs into the SC violation with current barrier placement. Current placement seems to be:

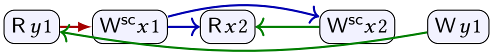
Violation of SC-DRF from [Watt et al. 2020, Fig. 9]:

```
Thread 1: x^sc = 1
Thread 2: x^sc = 2; r = x^sc; if (r==1) then s=x
```

The program is DRF. Should not be possible to have `r==1`, `s==2`.

[Dolan et al. 2018, §8.2]:

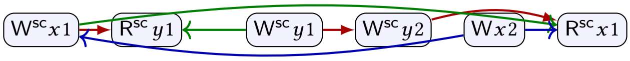
$r := y;\ x^{\mathsf{sc}} := 1;\ s := x \parallel x^{\mathsf{sc}} := 2;\ y := 1$



(sc1)

Watt et al. [2020, §3.1]:

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

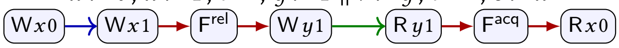


(sc2)

D.12 Fences

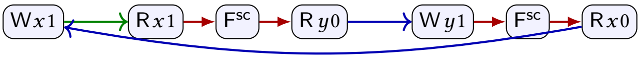
[Todo: Discuss.]

$$x := 0;\ x := 1;\ \mathsf{F}^{\mathsf{rel}};\ y := 1 \parallel r := y;\ \mathsf{F}^{\mathsf{acq}};\ s := x \tag{PUB2}$$



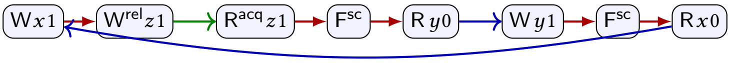
[Lahav et al. 2017, Fig. 5]:

$$x := 1 \parallel r := x;\ \mathsf{F}^{\mathsf{sc}};\ r := y \parallel y := 1;\ \mathsf{F}^{\mathsf{sc}};\ r := x \tag{SC3}$$



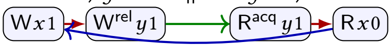
[Lahav et al. 2017, Fig. 6]

$$x := 1;\ z^{\mathsf{ra}} := 1; \parallel r := z^{\mathsf{acq}};\ \mathsf{F}^{\mathsf{sc}};\ r := y \parallel y := 1;\ \mathsf{F}^{\mathsf{sc}};\ r := x \tag{SC4}$$

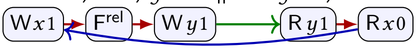


Here are several examples mixing fencing with release/acquire:

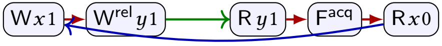
$$x := 1;\ y^{\mathsf{ra}} := 1 \parallel r := y^{\mathsf{acq}};\ s := x$$



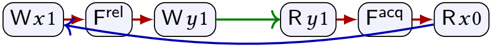
$$x := 1;\ \mathsf{F}^{\mathsf{rel}};\ y := 1 \parallel r := y^{\mathsf{acq}};\ s := x$$



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

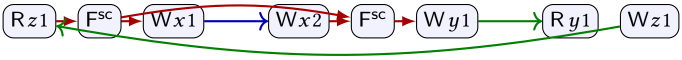


$$x := 1;\ \mathsf{F}^{\mathsf{rel}};\ y := 1 \parallel r := y;\ \mathsf{F}^{\mathsf{acq}};\ s := x$$



[Podkopaev et al. 2019, §D]: The following execution graph is not consistent in the promise-free declarative model of [Kang et al. 2017]. Nevertheless, its mapping to POWER (obtained by simply replacing Fsc with Fsync) is POWER-consistent and po ∪ rf is acyclic (so it is Strong-POWER-consistent). Note that, using promises, the promising semantics allows this behavior.

$$r := z;\ \mathsf{F}^{\mathsf{sc}};\ x := 1 \parallel x := 2;\ \mathsf{F}^{\mathsf{sc}};\ y := 1 \parallel r := y;\ z := 1$$



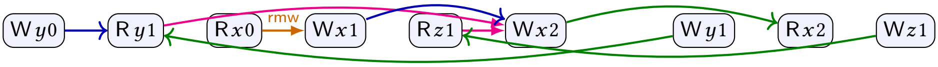
Allowed behavior on POWER… Is there a dependency in the last thread? If so, this is a problem.

[Podkopaev et al. 2019, §8]: To establish the correctness of compilation of the promising semantics to POWER, Kang et al. [2017] followed the approach of Lahav and Vafeiadis [2016]. This approach reduces compilation correctness to POWER to (i) the correctness of compilation to the POWER model strengthened with po ∪ rf acyclicity; and (ii) the soundness of local reorderings of memory accesses. To establish (i), Kang et al. [2017] wrongly argued that the strengthened POWER-consistency of mapped promise-free execution graphs imply the promise-free consistency of the source execution graphs. This is not the case due to SC fences, which have relatively strong semantics in the promise-free declarative model (see [Podkopaev et al. 2018, Appendix D] for a counter example). Nevertheless, our proof shows that the compilation claim of Kang et al. [2017] is correct.

D.13 RMWs

If RMWs simply use the same semantics as read and write, then we allow LDRF-PF-FAIL, which is used to show failure of LDRF-SC for the promising semantics in [Cho et al. 2021].

$$y := 0;\ \texttt{if}(y)\{\texttt{if}(!\mathsf{CAS}(x,0,1))\{\texttt{if}(z)\{x := 2\}\}\} \parallel y := 1;\ \texttt{if}(1 \neq \mathsf{CAS}(x,0,3))\{z := 1\}$$

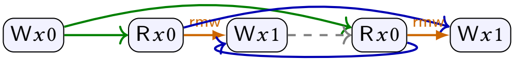


(LDRF-PF-FAIL)

To disallow this, we need to retain the dependency $(\mathsf{R}\,x2) \rightarrow (\mathsf{W}\,z1)$. For this, we need to avoid the substitution for x. This is why we use *READ′* instead of *READ* in the independent case for RMWs.

It is not possible for two RMWs to see the same write.

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

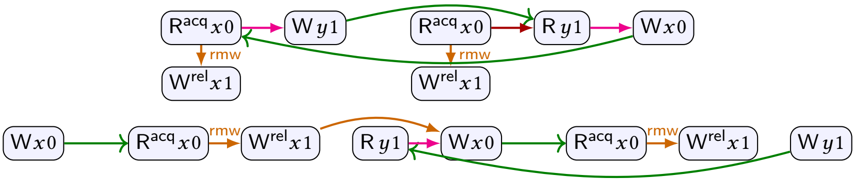


(RMW0)

The gray arrow is required the RMW atomicity axioms.

Lee et al. [2020] introduce PS2.0 to refine the treatment of RMWs in the promising semantics (PS). Their examples have the expected results here, with far less work. First they recall that PS requires quantification over multiple futures in order to disallow executions such as CDRF. (We showed the relaxed variant (CDRF-RLX) in §9.2.)

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

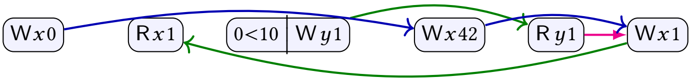


(CDRF)

This execution is clearly impossible, due to the cycle above. In this diagram, we have not drawn order adjacent to the writes of the RMWs, since this is not necessary to produce the cycle. If CDRF is allowed then DRF-RA fails.

PS does not support global value range analysis, as modeled by GA+E below. Our semantics permits GA+E:

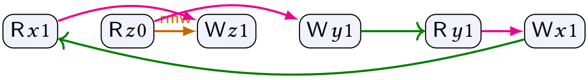
$$x := 0;\ \left(r := \mathsf{CAS}^{\mathsf{rlx},\mathsf{rlx}}(x, 0, 1);\ \texttt{if}(r{<}10)\{y := 1\} \parallel x := 42;\ x := y\right)$$



(GA+E)

PS also does not support register promotion, as modeled by RP below. Our semantics permits RP:

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

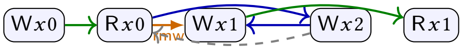


(RP)

Example D.1. Recall M10c: if $\lambda(c)$ overlaps $\lambda(d)$ and $d \xrightarrow{\mathsf{rmw}} e$ then (1) $c < e$ implies $c \le d$ and (2) $d < c$ implies $e \le c$.

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

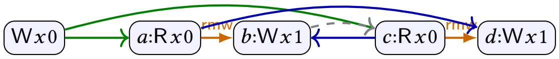
$$x := 0\,;\ \mathtt{INC}^{\mathsf{rlx},\mathsf{rlx}}(x) \parallel x := 2\,;\ r := x$$



By 1, since (Wx2) $\longrightarrow$ (Wx1), it must be that (Wx2) $\longrightarrow$ (Rx0), creating a cycle.

Example D.2. Two successful RMWs cannot see the same write:

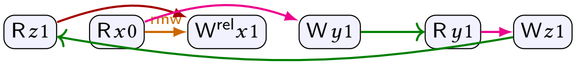
$$x := 0\,;\ (\mathtt{INC}^{\mathsf{rlx},\mathsf{rlx}}(x) \parallel \mathtt{INC}^{\mathsf{rlx},\mathsf{rlx}}(x))$$



The order from read-to-write is required by fulfillment. Apply 1 of the second RMW to $a \longrightarrow d$, we have that $a \longrightarrow c$. Subsequently applying 2 of the first RMW, we have $b \longrightarrow c$, creating a cycle.

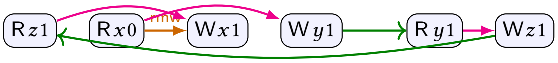
Example D.3. By using two actions rather than one, the definition allows examples such as the following, which is allowed by Arm8 [Podkopaev et al. 2019, Ex. 3.10]:

$$r := z\,;\ s := \mathtt{INC}^{\mathsf{rlx},\mathsf{rel}}(x)\,;\ y := s{+}1 \parallel r := y\,;\ z := r$$



A similar example, also allowed by Arm8 [Chakraborty and Vafeiadis 2019, Fig. 6]:

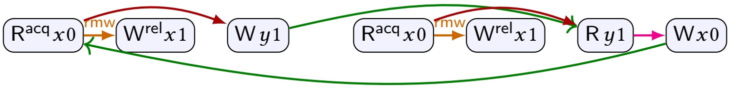
$$r := z\,;\ s := \mathtt{FADD}^{\mathsf{rlx},\mathsf{rlx}}(x, r)\,;\ y := s{+}1 \parallel r := y\,;\ z := r$$



This is allowed by WEAKESTMO, but not PS.

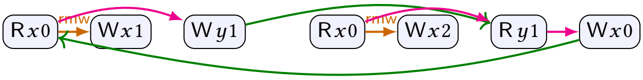
Example D.4. Consider the CDRF example from [Lee et al. 2020]:

$$\begin{gathered} r := \mathtt{INC}^{\mathsf{acq},\mathsf{rel}}(x)\,;\ \mathtt{if}(r{=}0)\{y := 1\} \\ \parallel\ r := \mathtt{INC}^{\mathsf{acq},\mathsf{rel}}(x)\,;\ \mathtt{if}(r{=}0)\{\mathtt{if}(y)\{x := 0\}\} \end{gathered}$$



Example D.5. Consider this example from [Lee et al. 2020, §C]:

$$\begin{gathered} r := \mathtt{CAS}^{\mathsf{rlx},\mathsf{rlx}}(x, 0, 1)\,;\ \mathtt{if}(r{\leqslant}1)\{y := 1\} \\ \parallel\ r := \mathtt{CAS}^{\mathsf{rlx},\mathsf{rlx}}(x, 0, 2)\,;\ \mathtt{if}(r{=}0)\{\mathtt{if}(y)\{x := 0\}\} \end{gathered}$$

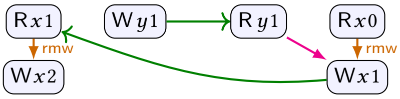


D.14 More RMW

These following examples are from [Cho et al. 2021].

CDRF shows that PwT semantics is not too permissive for ra-RMWs. But what about rlx-RMWs. The following execution is allowed by Arm8, and PS2.0, but disallowed by PS2.1.

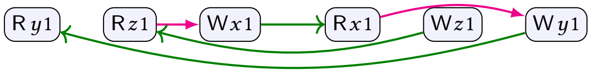
$$r := \mathtt{FADD}^{\mathsf{rlx,rlx}}(x, 1)\,;\, y := 1 \parallel r := y\,;\, s := \mathtt{FADD}^{\mathsf{rlx,rlx}}(x, r)$$



(RMW-W)

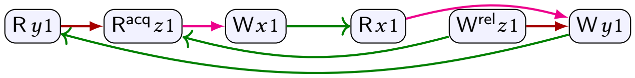
If this $\{z\}$-DRF-RA?

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



(NAIVE-LDRF-RA-FAIL)

Interpreting $\{z\}$ as ra:

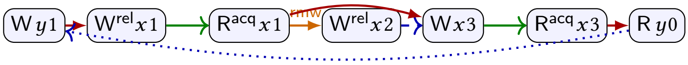


D.15 Fences and RMW

[Todo: Discuss.]

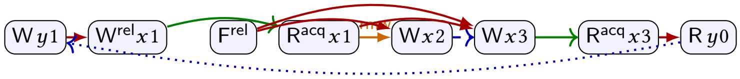
[Podkopaev et al. 2019, Remark 2, After example 3.1]: Aim: allow the splitting of release writes and RMWs into release fences followed by relaxed operations. In RC11 [Lahav et al. 2017], as well as in C/C++11 [Batty et al. 2011], this rather intuitive transformation, as we found out, is actually unsound.

$$y := 1\,;\, x^{\mathsf{ra}} := 1 \parallel \mathtt{INC}^{\mathsf{ra,ra}}(x)\,;\, x := 3 \parallel r := x^{\mathsf{acq}}\,;\, s := y$$



(R)C11 disallows the annotated behavior, due in particular to the release sequence formed from the release exclusive write to x in the second thread to its subsequent relaxed write. However, if we split the increment to fencerel; a := FADDacq,rlx(x, 1) (which intuitively may seem stronger), the release sequence will no longer exist, and the annotated behavior will be allowed. IMM overcomes this problem by strengthening sw in a way that ensures a synchronization edge for the transformed program as well

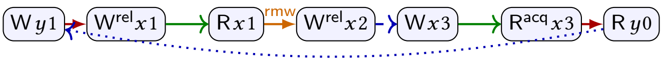
$$y := 1\,;\, x^{\mathsf{ra}} := 1 \parallel \mathsf{F}^{\mathsf{rel}}\,;\, \mathtt{INC}^{\mathsf{ra,rlx}}(x)\,;\, x := 3 \parallel r := x^{\mathsf{acq}}\,;\, s := y$$



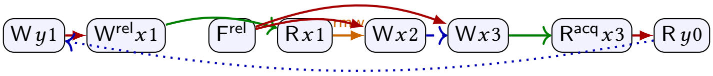
We seem to disallow both of these out of the box.

In the case of a relaxed read in the RMW, the outcome is allowed in both cases:

$$y := 1\,;\, x^{\mathsf{ra}} := 1 \parallel \mathtt{INC}^{\mathsf{rlx,ra}}(x)\,;\, x := 3 \parallel r := x^{\mathsf{acq}}\,;\, s := y$$



$$y := 1 ; x^{\mathsf{ra}} := 1 \parallel \mathsf{F}^{\mathsf{rel}} ; \mathtt{INC}^{\mathsf{rlx,rlx}}(x) ; x := 3 \parallel r := x^{\mathsf{acq}} ; s := y$$



E NOT FOR PUBLICATION

E.1 Recent discussion on JMM/JDK-dev

Raffaello Giulietti: “JEP 188: Java Memory Model Update” [1], the JMM wiki [2] and the jmm-dev mailing list [3] seem quite inactive. (The latter point explains why I’m posting to this list instead.)

The introduction of `j.l.i.VarHandle` [4] brought more access modes to Java, but in a narrative and informal way. A paper by Bender & Palsberg [5], addressing the formalization of the concurrent access modes, has been published in 2019 but I’m not sure if it caught the attention of the OpenJDK community.

So what is the current thinking for progressing the JMM spec?

Hans Boehm: I think it’s safe to say that it has been slow going, not just for Java, but for other languages as well.

In my view, the core problem, shared by pretty much all of them, is that we don’t have an established way to give well-defined semantics to potentially racing unordered accesses, like ordinary variable accesses in Java, or memory_order_relaxed accesses in C and C++. That’s particularly essential with the traditional Java language-based-security model, since we can’t just give up on racing accesses to ordinary variables.

I’m aware of a number of proposed solutions. But I don’t think we currently have enough confidence that they

- Are correct, and don’t have issues similar to the older models,
- Don’t have unintended consequences, particularly for compilation, and
- Are sufficiently comprehensible by programmers to actually be useful.

[Correctness] is hard because the models have gotten complex enough that reviewers are scarce. (A problem that I gather you’re familiar with.) The authors are commonly experts at formally analyzing the models, but it’s hard to analyze whether the model conflicts with some well-known, but perhaps not well-written-down compilation technique.

Probably even more controversially, I think we’ve realized that existing compiler technology can compile such racing code in ways that some of us are not 100% sure should really be allowed. Demonstrably unexecuted code can affect the semantics in ways that strike me as scary. (See https://wg21.link/p1217 for a down-to-assembly C++ version; [if I understand correctly], Lochbihler and others earlier came up with some closely related observations for Java.)

It might be possible to do what we’ve involuntarily done for C++: Punt the hard cases for now, and define what the model is for programs without racing ordinary accesses.

[p1217 is [Boehm 2019].]

Andrew Haley:

```
> Probably even more controversially, I think we've realized that
> existing compiler technology can compile such racing code in ways
> that some of us are not 100\% sure should really be allowed.
```

This implies, does it not, that the problem is not formalization as such, but that we don’t really understand what the language is supposed to mean? That’s always been my problem with OOTA:

I'm unsure whether the problem is due to the inadequacy of formal models, in which case the formalists can fix their own problem, or something we all have to pay attention to.

Hans Boehm: In some sense, I'm not sure either. The p1217 examples bother me in that they seem to violate some global programming rules ("if `x` is only ever null or refers to an object properly constructed by the same thread, then `x` should never appear to refer to an incompletely constructed object"). And there seems to be disagreement about whether the currently allowed behavior is "correct."

On the other hand, in practice the weirdness doesn't seem to break things. If you ask people advocating the current behavior, the answer will be that it doesn't matter because nobody writes code that way. If you ask people trying to analyzer or verify code, they'll probably be unhappy. And I haven't been able to convince myself that you cannot get yourself into these situations just by linking components together, each of which does something perfectly reasonable.

And there are very common code patterns (like the standard implementation of reentrant locks used by all Java implementations) that break if you allow general OOTA behavior. Which at least means that you can't currently formally verify such code. The theorem you'd be trying to prove is false with respect to the part of the language spec we know how to formalize.

It's a mess.

Andrew Haley:

```
> Demonstrably unexecuted code can affect the semantics in ways that strike me
> as scary. (See wg21.link/p1217 for a down-to-assembly C++ version; IIUC, Lochbihler
> and others earlier came up with some closely related observations for Java.)
```

Looking again at p1217, it seems to me that enforcing load-store ordering would have severe effects on compilers, at least without new optimization techniques. We hoist loads before loops and sink stores after them. When it all works out, there are no memory accesses in the loop. A load-store barrier in a loop would have the effect of forcing succeeding stores out to memory, and forcing preceding loads to reload from memory. It's not hard to imagine that this would cause an order-of-margnitude performance reduction in common cases.

I suppose one could argue that such optimizations would continue to be valid, so only those stores which would have been emitted anyway would be affected. But that's not how compilers work, as far as I know. In our IR for C2, memory accesses are not pinned in any way, so the only way to make unrelated accesses execute in any particular order is to add a dependency between all loads and stores.

Hans Boehm: I think it would be a fairly pervasive change to optimizers. It has also become clear in WG21, the C++ committee, that there is not enough support for requiring this. In that case, Ou and Demsky have a paper saying that the overhead is likely to be on the order of 1% or less. For Java if it were applied everywhere, it would probably be appreciably higher.

On the other hand, it's a bit harder than that to come up with examples where the generated x86 code has to be worse. Moving loads earlier in the code, or delaying stores, as you suggest, would still be fine. The only issue is with delaying loads past stores, which seems less common, though it can certainly be beneficial for reducing live ranges, probably some vectorization etc.

But it seems unlikely that such a restriction will be applied even to C++ memory_order_relaxed, much less Java ordinary variables.

Doug Lea: My stance in the less formal account (http://gee.cs.oswego.edu/dl/html/j9mm.html) as well as Shuyang Liu et al's ongoing formalization (see links from http://compilers.cs.ucla.edu/people/) is that the most you want to say about racy Java programs is that they are typesafe. As in: you can't see a String when expecting an int. Even this looser constraint is challenging to specify, prove, and extend. But it is a path for Java that might not apply to languages like C that are not guaranteed typesafe anyway, and so enter Undefined Behavior territory (as opposed to possibly-unexpected but still typesafe behavior).

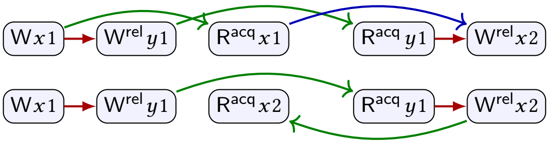
Han Boehm: But this now breaks some common idioms, right? In particular, I think a bunch of code assumes that racing assignments of equivalent primitive values or immutable objects to the same field are OK.

If, in 2004, our view of language-based security had been the same as it is now, then I completely agree that this would have been the right approach. But I think doing it now would require significant user code changes. Which might still be the best way forward ...

E.2 A Note on Mixed-Mode Data Races

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

$$x := 1;\ y^{\mathsf{rel}} := 1;\ r := x^{\mathsf{acq}} \parallel \mathtt{if}(y^{\mathsf{acq}})\{x^{\mathsf{rel}} := 2\}$$

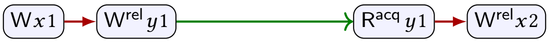


(¶)

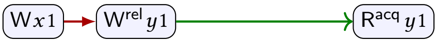
The program is data-race free. The two executions shown are the only top-level executions that include $(\mathsf{W}^{\mathsf{rel}} x 2)$.

Theorem 4.1 of [Dongol et al. 2019] is stated by extending execution sequences. In the terminology of [Dongol et al. 2019], a read is L-*weak* if it is sequentially stale. Let $\rho = (\mathsf{W} x 1)(\mathsf{W}^{\mathsf{rel}} y 1)(\mathsf{R}^{\mathsf{acq}} y 1)(\mathsf{W}^{\mathsf{rel}} x 2)$ be a sequence and $\alpha = (\mathsf{R}^{\mathsf{acq}} x 1)$. ρ is L-sequential and α is L-weak in $\rho\alpha$. But there is no execution of this program that includes a data race, contradicting the theorem. The error seems to be in Lemma A.4 of [Dongol et al. 2019], which states that if α is L-weak after an L-sequential ρ, then α must be in a data race. That is clearly false here, since $(\mathsf{R}^{\mathsf{acq}} x 1)$ is stale, but the program is data race free.

In proving the SC-LDRF result in [Jagadeesan et al. 2020, §8], we noted that our proof technique is more robust than that of [Dongol et al. 2019], because it limits the prefixes that must be considered. In (¶), the induction hypothesis requires that we add $(\mathsf{R}^{\mathsf{acq}} x 1)$ before $(\mathsf{W}^{\mathsf{rel}} x 2)$ since $(\mathsf{R}^{\mathsf{acq}} x 1) \longrightarrow (\mathsf{W}^{\mathsf{rel}} x 2)$. In particular,



is not a downset of (¶), because $(\mathsf{R}^{\mathsf{acq}} x 1) \longrightarrow (\mathsf{W}^{\mathsf{rel}} x 2)$. As 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 (¶).

F OLD NOTES

F.1 More optimizations

- Sound to strengthen the annotation on an action from rlx to ra, and from ra to sc.

From [Manson et al. 2005]:

- synchronization on thread local objects can be ignored or removed altogether (the caveat to this is the fact that invocations of methods like wait and notify have to obey the correct semantics – for example, even if the lock is thread local, it must be acquired when performing a wait),
- volatile fields of thread local objects can be treated as normal fields.
- redundant synchronization (e.g., when a synchronized method is called from another synchronized method on the same object) can be ignored or removed,

Counterexample for first two:

```
y=1; x^AR=1; r=X^AR; z=1
```

If you see $z = 1$ you must see $y = 1$

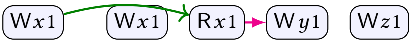
It would be nice if we could get at these with a strength reducing result: synchronization actions can be replaced by relaxed actions in some cases. Then the rules for relaxed read elimination and relaxed write elimination can be used to get rid of them.

F.2 Examples for semicolon semantics

- Parallel asymmetric: state result for *joint free* programs.
- Subsumption can be allowed on registers only
- We build substitutions
- Ignore substitutions when considering semantic equality.

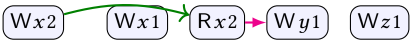
Value for r in (r=1 | Wz1) from (Wx1):

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



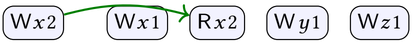
Value for r in (r=1 | Wz1) from (Wx1):

$$x := 2 \parallel x := 1;\, r := x;\, \texttt{if}(r{>}0)\{y := 1\};\, \texttt{if}(r{>}0)\{z := 1\}$$

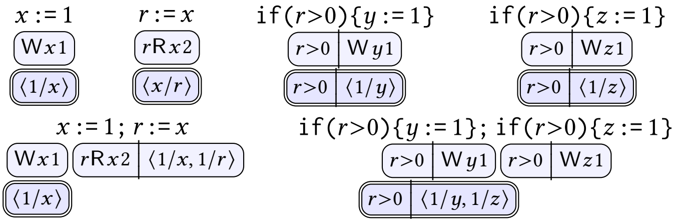


Note that this also contains pomset where value for r in (r=1 | Wy1) also comes from (Wx1):

$$x := 2 \parallel x := 1;\, r := x;\, \texttt{if}(r{>}0)\{y := 1\};\, \texttt{if}(r{>}0)\{z := 1\}$$

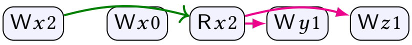


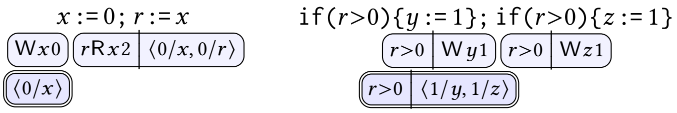
So our semantics will calculate the least ordered version. Then rely on augmentation to get the others.



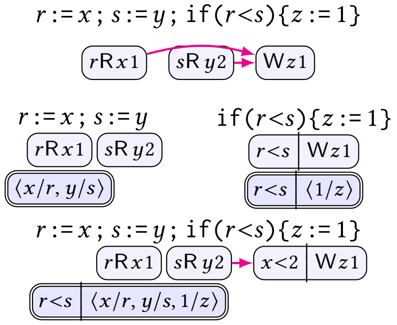
It is also possible that the read is necessary to give a value for r:

$$x := 2 \parallel x := 0;\, r := x;\, \texttt{if}(r{>}0)\{y := 1\};\, \texttt{if}(r{>}0)\{z := 1\}$$

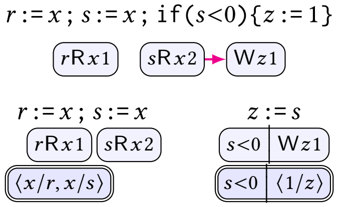




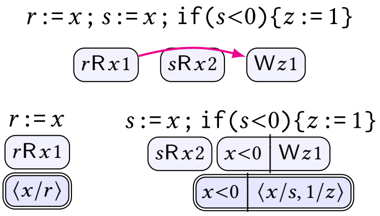
Dependency on two reads:



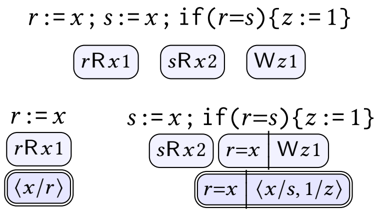
Don’t need to worry about confusing reads:



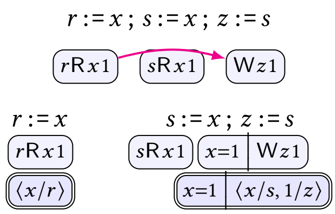
But we also have



Dependency on two reads (No dependency here):

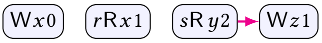


Another example:



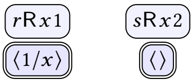
Value for r in ($r<s$ | Wz1) from (Wx0):

$$x := 0;\ r := x;\ s := y;\ \mathtt{if}(r<s)\{z := 1\}$$



Contrary to submission, reverse subsumption not okay.

$$x := 1 \qquad x := 2$$



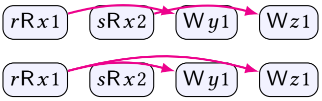
F.3 Playing around with 5a and 4b

If we do this, then swap 4b and 4c, In definition 2.10, take 1-4b of def 2.8, rather than all of it.

Another

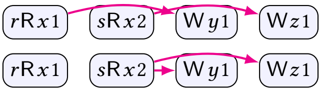
$$r := x;\ s := x;\ \mathtt{if}(r>0)\{y := 1\};\ \mathtt{if}(s>0)\{z := 1\}$$

$$r := x;\ \mathtt{if}(r>0)\{y := 1\};\ s := x;\ \mathtt{if}(s>0)\{z := 1\}$$

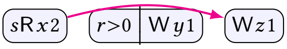


$$s := x;\ r := x;\ \mathtt{if}(r>0)\{y := 1\};\ \mathtt{if}(s>0)\{z := 1\}$$

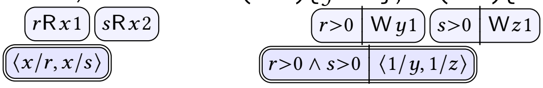
$$s := x;\ \mathtt{if}(s>0)\{z := 1\};\ r := x;\ \mathtt{if}(r>0)\{y := 1\}$$



$$s := x;\ \mathtt{if}(r>0)\{y := 1\};\ \mathtt{if}(s>0)\{z := 1\}$$



$$r := x;\ s := x \qquad \mathtt{if}(r>0)\{y := 1\};\ \mathtt{if}(s>0)\{z := 1\}$$



Idea to get rid of 4b and change 5a to the following:

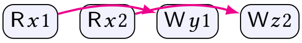
5a. if e writes then either $\kappa'(e)$ implies $\kappa(e)$, or some $c <' e$ reads v from x and $\kappa'(e)$ implies $\kappa(e)[v/x]$,

Need to get rid of 4b because it is sensitive to order of reads.

This change seems sound, because of consistency. But it also fails to validate read reordering on same variable, due to consistency.

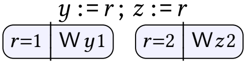
Without 4b, we still do not allow:

$$r := x;\ s := x;\ y := r;\ z := r$$



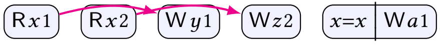
The following is not a pomset (consistency):

$$y := r;\ z := r$$



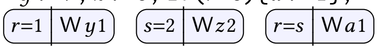
Without 4b, we still do not allow:

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



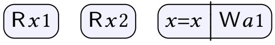
The following is not a pomset (consistency):

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



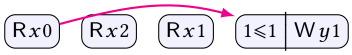
We do allow:

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



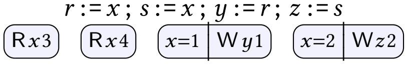
And also

$$r_1 := x;\ r_2 := x;\ r_3 := x;\ \mathtt{if}(r_3 \leqslant 1)\{y := 1\};$$



But we cannot wait forever to satisfy a precondition. This is not a pomset:

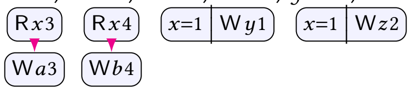
$$r := x;\ s := x;\ y := r;\ z := s$$



Note that reads that we delay must all be consistent.

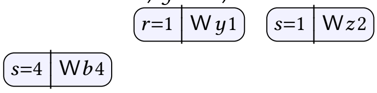
Also note that we cannot have:

$$r := x;\ a := r;\ s := x;\ b := s;\ y := r;\ z := s$$



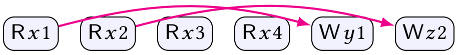
Because the following is not a pomset:

$$b := s;\ y := r;\ z := s$$



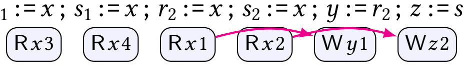
But we can have the following, since there is no order the reads:

$$r_1 := x;\ s_1 := x;\ r_2 := x;\ s_2 := x;\ y := r_2;\ z := s_2$$



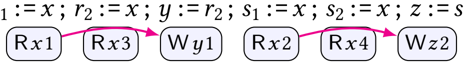
Because this is indistinguishable from:

$$r_1 := x;\ s_1 := x;\ r_2 := x;\ s_2 := x;\ y := r_2;\ z := s_2$$



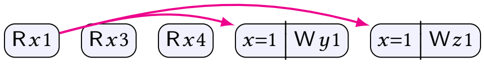
which is the same as:

$$r_1 := x;\ r_2 := x;\ y := r_2;\ s_1 := x;\ s_2 := x;\ z := s_2$$



But we can have:

$$p := x;\ r := x;\ s := x;\ y := r;\ z := s$$



Reads can only swap when their values are interchangeable in the following program.

F.4 Alan comments

```
x=s; y=r; z=3s+2r

x=s; y=r; z1=s; if(r odd){ z2=1} // using 1 and 3 as the reads
```

REFERENCES

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

Mark Batty. 2015. *The C11 and C++11 concurrency model.* Ph.D. Dissertation. University of Cambridge, UK.

Hans-J. Boehm. 2019. Out-of-thin-air, Revisited, Again (Revision 2). https://wg21.link/p1217.

Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. 2007. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4421)*, Rocco De Nicola (Ed.). Springer, 331–346. https://doi.org/10.1007/978-3-540-71316-6_23

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

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

Minki Cho, Sung-Hwan Lee, Chung-Kil Hur, and Ori Lahav. 2021. Modular Data-Race-Freedom Guarantees in the Promising Semantics. *Proc. ACM Program. Lang.* 2, PLDI. To Appear.

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

Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 architecture, operationally: concurrency and ISA. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 608–621. https://doi.org/10.1145/2837614.2837615

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

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

Alan Jeffrey and James Riely. 2019. On Thin Air Reads: Towards an Event Structures Model of Relaxed Memory. *Logical Methods in Computer Science* 15, 1 (2019), 25 pages. https://doi.org/10.23638/LMCS-15(1:33)2019

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

Ori Lahav and Udi Boker. 2020. Decidable verification under a causally consistent shared memory. 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, 211–226. https://doi.org/10.1145/3385412.3385966

Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. 2016. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 649–662. https://doi.org/10.1145/2837614.2837643

Ori Lahav and Viktor Vafeiadis. 2016. Explaining Relaxed Memory Models with Program Transformations. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9995)*, John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou (Eds.). Springer, 479–495. https://doi.org/10.1007/978-3-319-48989-6_29

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

Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. 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, 633–647. https://doi.org/10.1145/3062341.3062343

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

Nuno Lopes. 2016. RFC: Killing undef and spreading poison. https://lists.llvm.org/pipermail/llvm-dev/2016-October/106182.html.

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

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. 1999. Fixing the Java Memory Model. In *Proceedings of the ACM 1999 Conference on Java Grande, JAVA '99, San Francisco, CA, USA, June 12-14, 1999*, Geoffrey C. Fox, Klaus E. Schauser, and Marc Snir (Eds.). ACM, 89–98. https://doi.org/10.1145/304065.304106

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

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2018. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL* 2, POPL (2018), 19:1–19:29. https://doi.org/10.1145/3158107

Jaroslav Sevčík and David Aspinall. 2008. On Validity of Program Transformations in the Java Memory Model. In *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings (Lecture Notes in Computer Science, Vol. 5142)*, Jan Vitek (Ed.). Springer, 27–51. https://doi.org/10.1007/978-3-540-70592-5_3

Kasper Svendsen, Jean Pichon-Pharabod, Marko Doko, Ori Lahav, and Viktor Vafeiadis. 2018. A Separation Logic for a Promising Semantics. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*. Springer, 357–384. https://doi.org/10.1007/978-3-319-89884-1_13

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