Appendix A. Discussion

A.1. Closure properties

The semantics is closed with respect to *augments* and *downsets*. Augments include more order and stronger formulae; in examples, we typically consider pomsets that are augment-minimal. Downsets include a subset of initial events, similar to *prefixes* for strings.

Def 58. P_2 is an *augment* of P_1 if

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

Def 59. P_2 is an *downset* of P_1 if

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

Prop 60. *Suppose* $P_1 \in [\![S]\!]$.

1) If P_2 is an augment of P_1 then $P_2 \in [\![S]\!]$.
2) If P_2 is a downset of P_1 then $P_2 \in [\![S]\!]$.

A.2. Comparison with Weakest Preconditions

We compare traditional transformers to the dependent-case transformers of Def 48; thus we consider only totally ordered executions. Because we only consider the dependent case, we drop the superscript E on τ^E throughout this section. We also assume that each register appears at most once in a program, as we did throughout §2–4.

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

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

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

Here we have chosen an alternative formulation for the read rule, which is equivalent the more traditional $\psi[x/r]$, as long as registers occur at most once in a program. In Def 48, the transformers for the dependent case are as follows:

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

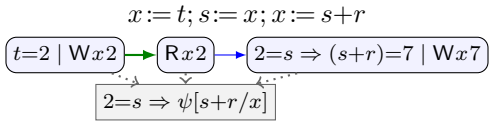
Only the read rule differs from the traditional one.

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

For example, let S_i be defined:

$$\begin{aligned} S_1 &= s{:=}x; x{:=}s{+}r \\ S_2 &= x{:=}t; S_1 \\ S_3 &= t{:=}2; r{:=}5; S_2 \end{aligned}$$

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

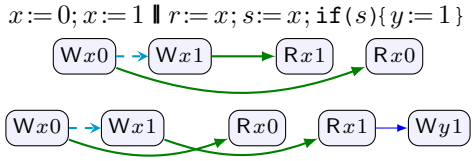


The predicate transformers are:

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

A.3. Completed Pomsets and Fork

It is sometimes useful to distinguish *terminated* or *completed* executions from partial executions. For example in $[\![x{:=}1; y{:=}1]\!]$, we expect completed executions to include two write actions. Note that this is different from being downset-maximal.



(1) is a downset of (2), but both are completed.

For pomsets with predicate transformers, we identify *completion* with *quiescence.*

Def 61. A pomset with predicate transformers P is *completed* if, for every quiescence symbol s, $\tau^E(\mathsf{s})$ implies s.

For example, there are no pomsets in $[\![\texttt{abort}]\!]$ that are completed, whereas the augment-minimal pomset of $[\![\texttt{skip}]\!]$ is completed.

While this definition is sensible for single *threads*, it is less satisfying for thread *groups*. To see why, consider that in $[\![\texttt{fork}\{S\}]\!]$:

- by T3, quiescence symbols and the symbol W have been substituted out of preconditions $\kappa(e)$,
- by F4, every predicate transformer τ^D is the identity function.

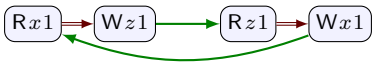
Every pomset in $[\![\texttt{fork}\{G\}]\!]$ is completed, by definition. As a result, in general, $[\![\texttt{fork}\{S\}]\!] \neq [\![S]\!]$.

The `fork` operation is asynchronous: In $[\![S_1;\texttt{fork}\{G\};S_2]\!]$, the threads in $[\![G]\!]$ run concurrently with $[\![S_1;S_2]\!]$.

$$r{:=}x;\texttt{fork}\{x{:=}1\}$$

In fact, perhaps surprisingly, $[\![r{:=}x;\texttt{fork}\{x{:=}1\}]\!] = [\![\texttt{fork}\{x{:=}1\};r{:=}x]\!]$. Order between the threads can be enforced using synchronization. For example, the "backwards" read above is forbidden in:

$$r{:=}x;z^{\mathsf{ra}}{:=}1;\texttt{fork}\{\texttt{if}(z^{\mathsf{ra}})\{x{:=}1\}\}$$



A.4. Fork-Join

In this subsection, we model a variant of our language that removes the asynchronous `fork` operation and adds a synchronous `fork-join`.

$$S ::= \texttt{abort} \mid \texttt{skip} \mid r{:=}M \mid r{:=}[L]^{\mu} \mid [L]^{\mu}{:=}M \mid \texttt{fork}\{G\};\texttt{join} \mid S_1;S_2 \mid \texttt{if}(M)\{S_1\}\texttt{else}\{S_2\}$$

In $(S_1;\texttt{fork}\{G\};\texttt{join};S_2)$, S_1 must complete before G begins, and threads in G must complete before S_2 begins. Thus $(\texttt{fork}\{r{:=}x\};\texttt{join})$ acts like a full fence. As modeled here, however, if G is empty, no order is imposed between S_1 and S_2. Thus $[\![\texttt{fork}\{\texttt{skip}\};\texttt{join}]\!] = [\![\texttt{skip}]\!]$.

To model `fork-join`, we give the semantics of thread groups using pomsets with preconditions *and termination*.

Def 62. A *pomset with preconditions and termination* is a pomset with preconditions (Def 12) together with a termination predicate (notation $\checkmark$).

Def 63. If $P \in \mathit{THRD}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$

1–2) as for *THRD* in Def 45,
T3) $\kappa(e)$ implies $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}][\mathsf{tt}/\mathsf{W}]$ if $\lambda_1(e)$ is a write, $\kappa(e)$ implies $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}][\mathsf{ff}/\mathsf{W}]$ otherwise.
T4) if $\checkmark$ then P is completed (Def 61).

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

1–8) as for $\parallel$ in Def 12,
9) $\checkmark$ implies $\checkmark_1 \wedge \checkmark_2$.

If $P \in \mathit{FORKJOIN}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$

1–2) as for *FORK* in Def 21,
F3) $\kappa(e)$ implies $\mathsf{Q}^*_{\mathsf{ro}} \wedge \mathsf{Q}^*_{\mathsf{wo}} \wedge \mathsf{Q}_{\mathsf{sc}} \wedge \kappa_1(e)$,
F4) $\tau^D(\psi)$ implies ψ, if $D = E$ and $\checkmark_1$,
F5) $\tau^D(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}]$, otherwise.

Def 64. Update Def 24 to include:

$$[\![\texttt{fork}\{G\};\texttt{join}]\!] = \mathit{FORKJOIN}[\![G]\!]$$

We embed pomsets with predicate transformers into pomsets with preconditions and termination using completion. The rules for thread groups keep track of the termination predicate. As noted in §A.3, every pomset in $[\![\texttt{fork}\{G\}]\!]$ is completed. In contrast, a pomset in $[\![\texttt{fork}\{G\};\texttt{join}]\!]$ is completed only if every thread in G is completed.

Top-level thread groups do not need quiescence symbols; thus, *THRD* removes all quiescence symbols by substitution. However, $\mathit{FORKJOIN}(\mathcal{P})$ adds every possible quiescence symbol as a precondition to the events of $\mathcal{P}$. For example, the preconditions of $[\![S \parallel 0]\!]$ do not contain quiescence symbols. Instead, the preconditions of $[\![\texttt{fork}\{S \parallel 0\};\texttt{join}]\!]$ are saturated with them. As a result, in completed top-level pomsets of $[\![S_1;\texttt{fork}\{G\};\texttt{join}]\!]$, all of the events from $[\![S_1]\!]$ must precede those of $[\![G]\!]$.

A similar thing happens with predicate transformers. Thread groups in $[\![S \parallel 0]\!]$ do not contain predicate transformers. Instead, all of the independent predicate transformers of $[\![\texttt{fork}\{S \parallel 0\};\texttt{join}]\!]$ take ψ to $\psi[\mathsf{ff}/\mathsf{Q}]$. As a result, in completed top-level pomsets of $[\![\texttt{fork}\{G\};\texttt{join};S_2]\!]$, all of the events from $[\![G]\!]$ must precede those of $[\![S_2]\!]$.

A.5. Using Independency for Coherence

In §3.2, we encoded coherence and synchronized access using quiescence symbols. Building on the language with `fork-join`, it is possible to model coherence using independency (§2.2), rather than encoding it in the logic.

Def 65. If $P \in (\mathcal{P}_1\,;\mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

1–8) as for ; in Def 23,
9) if $d \in E_1$ and $e \in E_2$ either $d < e$ or $a \leftrightarrow \lambda_2(e)$.

In the logic, we remove the symbols $\mathsf{Q}^x_{\mathsf{wo}}$ and $\mathsf{Q}^x_{\mathsf{ro}}$. Previously, we had given the semantics of ra access using $\mathsf{Q}^*_{\mathsf{wo}}$ and $\mathsf{Q}^*_{\mathsf{ro}}$, which were encoded using $\mathsf{Q}^x_{\mathsf{wo}}$ and $\mathsf{Q}^x_{\mathsf{ro}}$. With these gone, we introduce the quiescence symbol Q^* and $\mathsf{Q}_{\mathsf{acq}}$. Thus, the only quiescence symbols required are Q^*, $\mathsf{Q}_{\mathsf{acq}}$ and Q_{sc}. Fig 2 shows the difference with the semantics of §3.2.

Def 66. Let formulae $\mathsf{Q}^{\mathsf{S}}_{\mu}$ and $\mathsf{Q}^{\mathsf{L}}_{\mu}$ be defined:

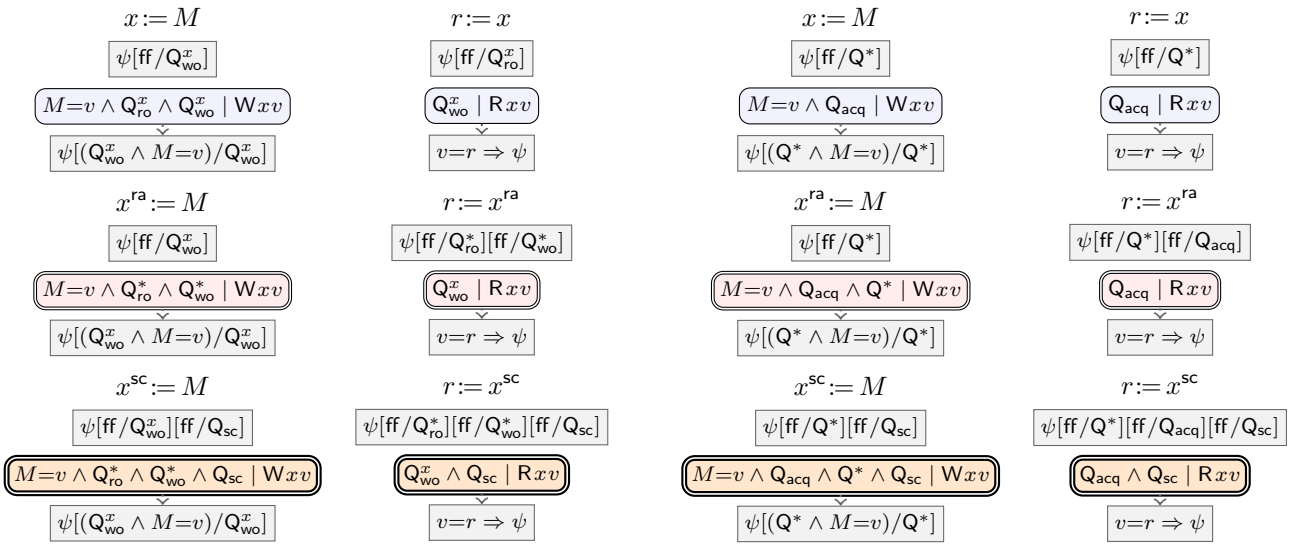
$$\begin{array}{ll} \mathsf{Q}^{\mathsf{S}}_{\mathsf{rlx}} = \mathsf{Q}_{\mathsf{acq}} & \mathsf{Q}^{\mathsf{L}}_{\mathsf{rlx}} = \mathsf{Q}_{\mathsf{acq}} \\ \mathsf{Q}^{\mathsf{S}}_{\mathsf{ra}} = \mathsf{Q}_{\mathsf{acq}} \wedge \mathsf{Q}^* & \mathsf{Q}^{\mathsf{L}}_{\mathsf{ra}} = \mathsf{Q}_{\mathsf{acq}} \\ \mathsf{Q}^{\mathsf{S}}_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{acq}} \wedge \mathsf{Q}^* \wedge \mathsf{Q}_{\mathsf{sc}} & \mathsf{Q}^{\mathsf{L}}_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{acq}} \wedge \mathsf{Q}_{\mathsf{sc}} \end{array}$$

Let substitutions $[\phi/\mathsf{Q}^{\mathsf{S}}_{\mu}]$ and $[\phi/\mathsf{Q}^{\mathsf{L}}_{\mu}]$ be defined:

$$\begin{array}{ll} [\phi/\mathsf{Q}^{\mathsf{S}}_{\mathsf{rlx}}] = [\phi/\mathsf{Q}^*] & [\phi/\mathsf{Q}^{\mathsf{L}}_{\mathsf{rlx}}] = [\phi/\mathsf{Q}^*] \\ [\phi/\mathsf{Q}^{\mathsf{S}}_{\mathsf{ra}}] = [\phi/\mathsf{Q}^*] & [\phi/\mathsf{Q}^{\mathsf{L}}_{\mathsf{ra}}] = [\phi/\mathsf{Q}^*, \phi/\mathsf{Q}_{\mathsf{acq}}] \\ [\phi/\mathsf{Q}^{\mathsf{S}}_{\mathsf{sc}}] = [\phi/\mathsf{Q}^*, \phi/\mathsf{Q}_{\mathsf{sc}}] & [\phi/\mathsf{Q}^{\mathsf{L}}_{\mathsf{sc}}] = [\phi/\mathsf{Q}^*, \phi/\mathsf{Q}_{\mathsf{acq}}, \phi/\mathsf{Q}_{\mathsf{sc}}] \end{array}$$

Def 67. Update Def 23 and 63 to:

S3) $\kappa(e)$ implies $M{=}v \wedge \mathsf{Q}^{\mathsf{S}}_{\mu}$,
L3) $\kappa(e)$ implies $\mathsf{Q}^{\mathsf{L}}_{\mu}$,
F3) $\kappa(e)$ implies $\mathsf{Q}^* \wedge \mathsf{Q}_{\mathsf{acq}} \wedge \mathsf{Q}_{\mathsf{sc}} \wedge \kappa_1(e)$,
S4) $\tau^D(\psi)$ implies $\psi[(\mathsf{Q}^* \wedge M{=}v)/\mathsf{Q}^*]$,



(a) Quiescence Examples (§3.2)

(b) Quiescence Examples (§A.5)

Figure 2: Quiescence Examples

S5) $\tau^C(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}}_\mu]$,
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}}_\mu]$.

The most interesting examples in Fig 2b concern ra access. Every independent transformer substitutes $[\mathsf{ff}/\mathsf{Q}^*]$. Q^* is a precondition for any releasing write e, ensuring that all preceding events must are ordered before e. Conversely, $\mathsf{Q}_{\mathsf{acq}}$ is a precondition of every event. The independent transformer for any acquiring read e substitutes $[\mathsf{ff}/\mathsf{Q}_{\mathsf{acq}}]$, ensuring that all following events must be ordered after e.

As before, the substitution in S4 ensures that left merges are not quiescent (Ex 31).

Item 9 of Def 65 ensures coherence. This definition is incompatible with asynchronous `fork` parallelism of Def 21, where we expect executions such as:

$$\texttt{fork}\{r := x\}; x := 1$$
$$\mathsf{R}x1 \longleftarrow \mathsf{W}x1$$

Item 9 would require $(\mathsf{R}x1) \dashrightarrow (\mathsf{W}x1)$, forbidding this.

A.6. Substitutions

Recall the load rules from §5.1:

L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $(v{=}r \vee x{=}r) \Rightarrow \psi$, when $E \neq \emptyset$,
L6) $\tau^B(\psi)$ implies ψ, when $E = \emptyset$.

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

L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi[r/x]$,
L5) $\tau^C(\psi)$ implies $(v{=}r \vee x{=}r) \Rightarrow \psi[r/x]$, when $E \neq \emptyset$.
L6) $\tau^B(\psi)$ implies $\psi[r/x]$, when $E = \emptyset$.

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

$$\texttt{if}(r \wedge s \text{ even})\{y := 1\}; \texttt{if}(r \wedge s)\{z := 1\}$$
$$r \wedge s \text{ even} \mid \mathsf{W}y1$$
$$r \wedge s \mid \mathsf{W}z1$$

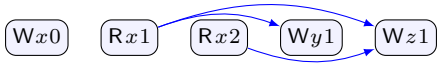
Prepending $(s{:=}x)$, we get the same result regardless of whether we substitute $[s/x]$, since x does not occur in either precondition. Here we show the independent case:

$$s := x; \texttt{if}(r \wedge s \text{ even})\{y := 1\}; \texttt{if}(r \wedge s)\{z := 1\}$$
$$(2{=}s \vee x{=}s) \Rightarrow (r \wedge s \text{ even}) \mid \mathsf{W}y1$$
$$\mathsf{R}x2 \qquad (2{=}s \vee x{=}s) \Rightarrow (r \wedge s) \mid \mathsf{W}z1$$

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

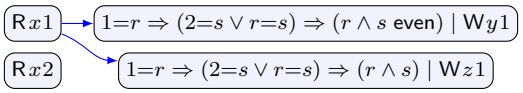
$$r := x; s := x; \texttt{if}(r \wedge s \text{ even})\{y := 1\}; \texttt{if}(r \wedge s)\{z := 1\}$$
$$\mathsf{R}x1 \to 1{=}r \Rightarrow (2{=}s \vee x{=}s) \Rightarrow (r \wedge s \text{ even}) \mid \mathsf{W}y1$$
$$\mathsf{R}x2 \to 1{=}r \Rightarrow (2{=}s \vee x{=}s) \Rightarrow (r \wedge s) \mid \mathsf{W}z1$$

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

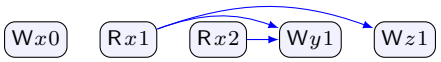


The situation reverses with the substitution $[r/x]$:

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



Prepending $(x{:=}0)$:



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

Thinking in terms of hardware, the difference is whether reads update the cache, thus clobbering preceding writes. With $[r/x]$, reads clobber the cache, whereas without the substitution, they do not. Since most caches work this way, the model with $[r/x]$ is likely preferred for modeling hardware. In a software model, however, we see no reason to prefer one of these over the other.

Appendix B. Differences with OOPSLA

Substitution. [12] uses substitution rather than Skolemizing. Indeed our use of Skolemization is motivated by disjunction closure for predicate transformers, which do not appear in [12]; see §2.4.

In §5.1, we give the semantics of load for nonempty pomsets as:

L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $(v{=}r \vee x{=}r) \Rightarrow \psi$.

In [12], the definition is roughly as follows:

L4) $\tau^D(\psi)$ implies $\psi[v/r][v/x]$,
L5) $\tau^C(\psi)$ implies $\psi[v/r][v/x] \wedge \psi[x/r]$.

These substitutions collapse x and r, allowing local invariant reasoning, as in §5.1. Without Skolemizing it is necessary to substitute $[x/r]$, since the reverse substitution $[r/x]$ is useless when r is bound.

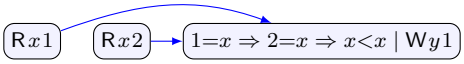
Removing the substitution of $[x/r]$ in the independent case has a small technical advantage: we no longer require *extended* expressions (which include memory references), since substitutions no longer introduce memory references.

The substitution $[x/r]$ does not work with Skolemization, even for the dependent case, since we lose the unique marker for each read. In effect, this forces the reads to the same values. To be concrete, the candidate definition would modify L4 to be:

L4) $\tau^D(\psi)$ implies $v{=}x \Rightarrow \psi[x/r]$.

Using this definition, consider the following:

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



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

Consistency. [12] imposes *consistency*, which requires that for every pomset P, $\bigwedge_e \kappa(e)$ is satisfiable. Associativity requires that we allow pomsets with inconsistent preconditions. Consider a variant of Ex 53 from §5.3.

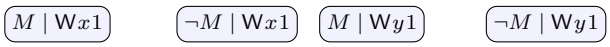
$\texttt{if}(M)\{x{:=}1\}$ $\texttt{if}(!M)\{x{:=}1\}$ $\texttt{if}(M)\{y{:=}1\}$ $\texttt{if}(!M)\{y{:=}1\}$

Associating left and right, we have:

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

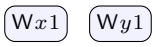
Associating into the middle, instead, we require:

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



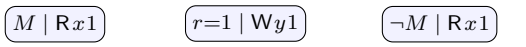
Joining left and right, we have:

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



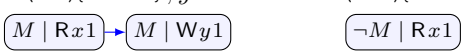
Causal Strengthening. [12] imposes *causal strengthening*, which requires for every pomset P, if $d \leq e$ then $\kappa(e)$ implies $\kappa(d)$. Associativity requires that we allow pomsets without causal strengthening. Consider the following.

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



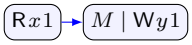
Associating left, with causal strengthening:

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



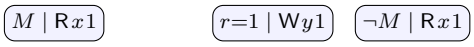
Finally, merging:

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



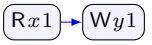
Instead, associating right:

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



Merging:

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

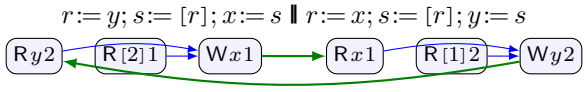


With causal strengthening, the precondition of $\mathsf{W}y1$ depends upon how we associate. This is not an issue in [12], which always associates to the right.

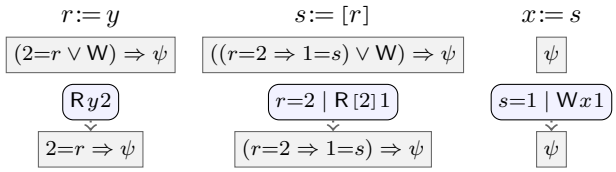
Causal Strengthening and Address Dependencies. In order to guarantee that address calculation does not introduce thin-air executions, the predicate transformer for address calculation must be chosen carefully. Combing Def 43 and Def 56 we have:

L4) $\tau^D(\psi)$ implies $(L{=}\ell \Rightarrow v{=}r) \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $((L{=}\ell \Rightarrow v{=}r) \vee \mathsf{W}) \Rightarrow \psi$.

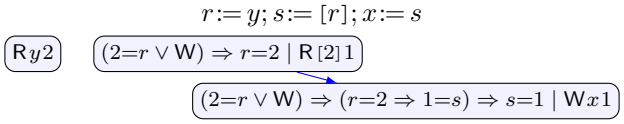
Consider the following program, from [12, §5], where initially $x = 0$, $y = 0$, $[0] = 0$, $[1] = 2$, and $[2] = 1$. It should only be possible to read 0, disallowing the attempted execution below:



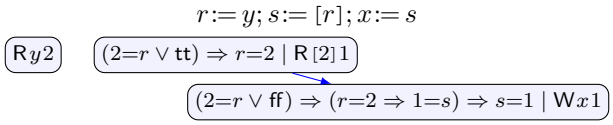
Looking at the left thread:



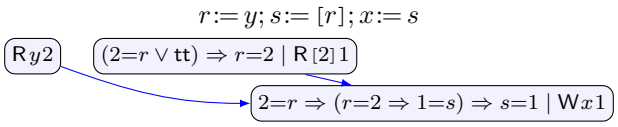
Composing, we have:



Substituting for W:



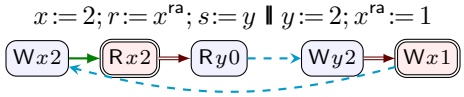
The precondition of $(\mathsf{R}\,[2]\,1)$ is a tautology, but the precondition of $(\mathsf{W}x1)$ is not. This forces a dependency:



All the preconditions are now tautologies.

Parallel Composition. In [12, §2.4], parallel composition is defined allowing coalescing of events. Here we have forbidden coalescing. This difference appears to be arbitrary. In [12], however, there is a mistake in the handling of termination actions. The predicates should be joined using $\wedge$, not $\vee$.

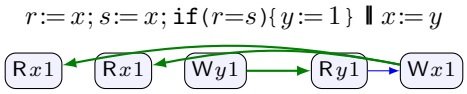
Internal Acquiring Reads. Shortly after publication, Podkopaev [20] noticed a shortcoming of the implementation on ARM8 in [12, §7]. The proof given there assumes that all internal reads can be dropped. However, this is not the case for acquiring reds. For example, [12] disallows the following execution, which is allowed by ARM8 and TSO.



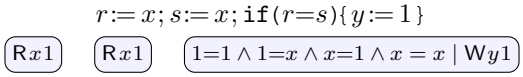
The solution we have adopted is to allow an acquiring read to be downgraded to a relaxed read when it is preceded (sequentially) by a relaxed write that could fulfill it. This solution allows executions that are not allowed under ARM8 since we do not insist that the local relaxed write is actually read from. This may seem counterintuitive, but we don't see a local way to be more precise.

As a result, we use a different proof strategy for ARM8 implementation, which does not rely on read elimination. The proof idea uses a recent alternative characterization of ARM8 [1, 3].

Redundant Read Elimination. Contrary to the claim, redundant read elimination fails for [12]. We discussed redundant read elimination in §5.2. Consider JMM Causality Test Case 2, which we discussed there.

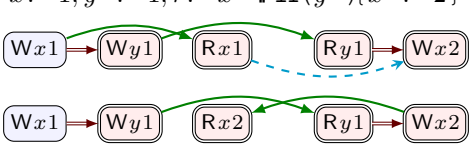


Under the semantics of [12], we have



The precondition of $(\mathsf{W}y1)$ is *not* a tautology, and therefore redundant read elimination fails. (It is a tautology in $r{:=}x$; $s{:=}r$; $\texttt{if}(r{=}s)\{y{:=}1\}$.) In [12, §3.1], we incorrectly stated that the precondition of $(\mathsf{W}y1)$ was $1{=}1 \wedge x{=}x$.

A Note on Mixed Races. In preparing this paper, we came across the following example, which appears to invalidate Theorem 4.1 of [8].



(3)

(4)

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

Theorem 4.1 of [8] is stated by extending execution sequences. In the terminology of [8], a read is *L-weak* if it is sequentially stale. Let $\rho = (\mathsf{W}x1)(\mathsf{W}y1)(\mathsf{R}y1)(\mathsf{W}x2)$ be a sequence and $\alpha = (\mathsf{R}x1)$. ρ is L-sequential and α is L-weak in $\rho\alpha$. But there is no execution of this program that includes a data race, contradicting the theorem. The error seems to be in Lemma A.4 of [8], 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 $(\mathsf{R}x1)$ is stale, but the program is data race free.

In proving the SC-LDRF result in [12, §8], we noted that our proof technique is more robust than that of [8], because it limits the prefixes that must be considered. In (3), the induction hypothesis requires that we add $(\mathsf{R}x1)$ before $(\mathsf{W}x2)$ since $(\mathsf{R}x1)$ -→ $(\mathsf{W}x2)$. In particular,

is not a downset of (3), because $(\mathsf{R}x1)$ -→ $(\mathsf{W}x2)$. As we noted in [12, §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 (3).