Sequential Composition for Relaxed Memory: Pomsets with Predicate Transformers

Anonymous
Anonymous Institution

***Abstract*—Program logics and semantics tell us that when executing ((S1; S2), state0), we execute (S1, state0) to arrive at state1, then execute (S2, state1) to arrive at the final state2.**

This is a delightfully simple story that can be explained to children. It is also a lie.

Processors execute instructions out of order, due to pipelines and caches. Compilers reorder programs even more dramatically. All of this reordering is meant to be unobservable in single-threaded code. In multi-threaded code, however, all bets are off. A formal attempt to understand the resulting mess is known as a "relaxed memory model."

Most of the relaxed memory models that have been proposed are designed to help us understand whole program execution: they have no compositionality properties whatsoever. Recently, denotation models have appeared that treat *concurrent* execution compositionally. One such model is "Pomsets with Preconditions". It remains an open question, however, whether it is possible to treat *sequential* execution compositionally in such a model, without overly restricting processors and compilers.

We propose adding families of predicate transformers to Pomsets with Preconditions. The resulting model is denotational, supporting both parallel and sequential composition. When composing (S1;S2), the predicate transformer used to validate the precondition of an event in S2 is chosen based on the dependency order from S1 into this event. As usual in work on relaxed memory, we have not handled loops or recursion.

Happily, most of the results expected of a relaxed memory model can be established by appeal to prior work. So here we are able to concentrate on the model itself. The model is formalized in Agda, where we have established associativity for sequential composition.

For the memory-model specialist, we retain the good properties of the prior work on Pomsets with Preconditions, fixing some errors along the way. These properties include efficient implementation on ARMv8, support for compiler optimizations, support for logics that prove the absence of thin-air behaviors, and a local data race freedom theorem.

1. Introduction

Our approach follows that of weakest precondition semantics of Dijkstra [4], which provides an alternative characterization of Hoare logic [6] by mapping postconditions to preconditions.

2. Model

Batty suggest example where dependencies are added and also go away, perhaps by store forwarding. Something like: `(r=x; y=1); (s=y; z=s+r)`

2.1. Preliminaries

The syntax is built from

- a set of *values* $\mathcal{V}$, ranged over by v, w, ℓ, k,
- a set of *registers* $\mathcal{R}$, ranged over by r, s,
- a set of *expressions* $\mathcal{M}$, ranged over by M, N, L.

Memory locations are tagged values, written $[\ell]$. Let $\mathcal{X}$ be the set of memory locations, ranged over by x, y, z.

We require that

- values and registers are disjoint,
- values include at least the constants 0 and 1,
- expressions include at least registers and values,
- expressions do *not* include memory locations.

We model the following language.

$$
\begin{array}{rcl}
\mu & ::= & \mathsf{rlx} \mid \mathsf{ra} \mid \mathsf{sc} \\
S & ::= & \mathtt{abort} \mid \mathtt{skip} \mid r := M \mid r := [L]^{\mu} \mid [L]^{\mu} := M \\
 & & \mid \mathtt{fork}\, G \mid S_1 ; S_2 \mid \mathtt{if}\,(M)\,\{S_1\}\,\mathtt{else}\,\{S_2\} \\
G & ::= & 0 \mid S \mid G_1 \parallel G_2
\end{array}
$$

Memory modes, μ, are relaxed (rlx), release-acquire (ra), and sequentially consistent (sc). Relaxed is the default. *Commands*, aka *statements*, S, include memory accesses at a given mode, as well as the usual structural constructs. *Thread groups*, G, include commands and 0, which denotes inaction. The `fork` command spawns a thread group.

The semantics is built from the following.

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

Subsets of $\mathcal{E}$ are ranged over by E, D, C, B.

We require that:

- actions include writes ($\mathsf{W}xv$) and reads ($\mathsf{R}xv$),
- formulae include equalities ($M{=}N$) and ($M{=}x$),
- formulae include symbols W, Q_{sc}, Q_{ra}, $\mathsf{Q}^{x}_{\mathsf{rw}}$, $\mathsf{Q}^{x}_{\mathsf{wo}}$, $\downarrow^{x}$,
- formulae are closed under negation, conjunction, disjunction, and substitutions $[M/r]$ and $[M/x]$,
- there is an entailment relation $\vDash$ between formulae,

- $\models$ has the expected semantics for $=$, $\neg$, $\wedge$, $\vee$, $\Rightarrow$,
- $\mathsf{Q}_{\mathsf{sc}} \models \mathsf{Q}_{\mathsf{ra}} \models \mathsf{Q}^{x}_{\mathsf{rw}} \models \mathsf{Q}^{y}_{\mathsf{wo}}$ when $x = y$.

Logical formulae include equations over locations and registers, such $(x{=}1)$ and $(r{=}s{+}1)$. We use expressions as formulae, coercing M to $M{\neq}0$. Formulae are subject to substitutions of the form $[M/r]$ and $[M/x]$; actions are not.

We say ϕ *implies* ψ if $\phi \models \psi$. We say ϕ is a *tautology* if $\mathsf{tt} \models \phi$. We say ϕ is *unsatisfiable* if $\phi \models \mathsf{ff}$.

We additionally assume that either:

- each register appears at most once in a program, or
- there are registers $\mathcal{S}_{\mathcal{E}} = \{s_e \mid e \in \mathcal{E}\}$ that do not appear in programs.

In contexts that make no use of $\mathcal{S}_{\mathcal{E}}$, we make the first assumption.

2.2. Pomsets

We first consider a fragment of our language that can be modeled using simple pomsets.

Definition 1. A *pomset* over $\mathcal{A}$ is a tuple $(E, \leq, \lambda)$ where

- $E \subset \mathcal{E}$ is a set of *events*,
- $\leq \subseteq (E \times E)$ is the *causality* partial order,
- $\lambda : E \to \mathcal{A}$ is a *labeling*.

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

We lift terminology from actions to events. For example, we say that e writes x if $\lambda(e)$ writes x. We also drop quantifiers when clear from context, such as $(\forall e \in E)(\forall x \in \mathcal{X})$.

Definition 2. Action $(\mathsf{W}xv)$ *matches* $(\mathsf{R}xw)$ when $v = w$. Action $(\mathsf{W}xv)$ *blocks* $(\mathsf{R}xw)$, for any v, w.

Event e is *fulfilled* if there is a $d \leq e$ which matches it and, for any c which can block e, either $c \leq d$ or $e \leq c$.

Pomset P is *fulfilled* if every read in P is fulfilled.

Definition 3. Actions a and b are *independent* (notation $a \leftrightarrow b$) if either both are reads or they are accesses to different locations. Formally $\leftrightarrow\ = \{(\mathsf{R}xv, \mathsf{R}yw)\} \cup \{(\mathsf{R}xv, \mathsf{W}yw), (\mathsf{W}xv, \mathsf{R}yw), (\mathsf{W}xv, \mathsf{W}yw) \mid x \neq y\}$.

Definition 4. If $P \in \mathit{NIL}$ then $E = \emptyset$.
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) $E = (E_1 \cup E_2)$,
2) if $e \in E_1$ then $\lambda(e) = \lambda_1(e)$,
3) if $e \in E_2$ then $\lambda(e) = \lambda_2(e)$,
4) if $d \leq_1 e$ then $d \leq e$,
5) if $d \leq_2 e$ then $d \leq e$,
6) E_1 and E_2 are disjoint.

If $P \in (a \to \mathcal{P}_2)$ then $(\exists P_2 \in \mathcal{P}_2)$

1) $E = (E_1 \cup E_2)$,
2) if $d, e \in E_1$ then $d = e$,
3) if $e \in E_1$ then $\lambda(e) = a$,
4) if $e \in E_2$ then $\lambda(e) = \lambda_2(e)$,
5) if $d \leq_2 e$ then $d \leq e$,
6) if $d \in E_1$ and $e \in E_2$ then either $d \leq e$ or $a \leftrightarrow \lambda_2(e)$.

Definition 5. For a language fragment, the semantics is:

$$[\![x := v; S]\!] = (\mathsf{W}xv) \to [\![S]\!] \qquad [\![\mathtt{skip}]\!] = [\![0]\!] = \mathit{NIL}$$
$$[\![r := x; S]\!] = \textstyle\bigcup_v (\mathsf{R}xv) \to [\![S]\!] \qquad [\![G_1 \parallel G_2]\!] = [\![G_1]\!] \parallel [\![G_2]\!]$$

If we take $\leftrightarrow\ = \emptyset$, then we have sequentially consistent execution.

[Do Examples.]

[Do examples with coherence.]

[Note that this allows mumbling.]

[Use refinement (that is subset order) as notion of compiler optimization.]

[Talk about Mazurkiewicz traces.]

2.3. Pomsets with Preconditions

[Problem with previous section is that notion of dependency is impoverished]

The model described here is essentially the model of [7], restricting attention to relaxed access. We discuss the differences in the appendix.

Definition 6. A *pomset with preconditions* is a pomset together with $\kappa : E \to \Phi$.

Definition 7. A pomset with preconditions is *top level* if it is fulfilled and every precondition is a tautology.

Definition 8. If $P \in \mathit{NIL}$ then $E = \emptyset$.
If $P \in (\mathcal{P}_1 \parallel \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

1–6) as for $\parallel$ in Def. 4,
7) if $e \in E_1$ then $\kappa(e)$ implies $\kappa_1(e)$,
8) if $e \in E_2$ then $\kappa(e)$ implies $\kappa_2(e)$.

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

1–5) as for $\parallel$ in Def. 4 (ignoring disjointness),
6) if $e \in E_1 \setminus E_2$ then $\kappa(e)$ implies $\phi \wedge \kappa_1(e)$,
7) if $e \in E_2 \setminus E_1$ then $\kappa(e)$ implies $\neg\phi \wedge \kappa_2(e)$,
8) if $e \in E_1 \cap E_2$ then $\kappa(e)$ implies $(\phi \Rightarrow \kappa_1(e)) \wedge (\neg\phi \Rightarrow \kappa_2(e))$.

If $P \in \mathit{ST}(x, M, \mathcal{P}_2)$ then $(\exists P_2 \in \mathcal{P}_2)\ (\exists v \in \mathcal{V})$

1–6) as for $(\mathsf{W}xv) \to \mathcal{P}_2$ in Def. 4,
7) if $e \in E_1 \setminus E_2$ then $\kappa(e)$ implies $M{=}v$,
8) if $e \in E_2 \setminus E_1$ then $\kappa(e)$ implies $\kappa_2(e)$,
9) if $e \in E_1 \cap E_2$ then $\kappa(e)$ implies $M{=}v \vee \kappa_2(e)$.

If $P \in \mathit{LD}(r, x, \mathcal{P}_2)$ then $(\exists P_2 \in \mathcal{P}_2)\ (\exists v \in \mathcal{V})$

1–6) as for $(\mathsf{R}xv) \to \mathcal{P}_2$ in Def. 4,
7) if $e \in E_2 \setminus E_1$ then either $\kappa(e)$ implies $r{=}v \Rightarrow \kappa_2(e)$ and $(\exists d \in E_1)\ d < e$, or $\kappa(e)$ implies $(r{=}v \vee r{=}x) \Rightarrow \kappa_2(e)$.

Definition 9. For a language fragment, the semantics is:

$$[\![\mathtt{if}\,(M)\,\{S_1\}\,\mathtt{else}\,\{S_2\}]\!] = \mathit{IF}(M{\neq}0, [\![S_1]\!], [\![S_2]\!])$$
$$[\![x := M; S]\!] = \mathit{ST}(x, M, [\![S]\!]) \qquad [\![\mathtt{skip}]\!] = [\![0]\!] = \mathit{NIL}$$
$$[\![r := x; S]\!] = \mathit{LD}(r, x, [\![S]\!]) \quad [\![G_1 \parallel G_2]\!] = [\![G_1]\!] \parallel [\![G_2]\!]$$

Following our convention for subscripts, in the final clause of *LD*, $<$ refers to the order of P. Also note that *LD* does not constrain $\kappa(e)$ if $e \in E_1$.

[Stuff about conditionals and merging events.]

2.4. Pomsets with Predicate Transformers

[The problem with the previous section is that there's no story for sequential composition.]

The final semantic functions for load and store, given in Figure 1, are quite complex. We explain the definition by looking at its constituent parts, starting with Def. 15, below, which captures dependency. In §3, we add *quiescence*, which encodes coherence, release-acquire and SC access, and termination. In §4, we add peculiarities that are necessary for efficient implementation on ARM8. In §5, we discuss the complications required to validate if-closure and to allow address calculation.

Definition 10. A *predicate transformer* is a monotone function $\tau : \Phi \to \Phi$ such that $\tau(\mathsf{ff})$ is ff, $\tau(\psi_1 \wedge \psi_2)$ is $\tau(\psi_1) \wedge \tau(\psi_2)$, and $\tau(\psi_1 \vee \psi_2)$ is $\tau(\psi_1) \vee \tau(\psi_2)$.

Definition 11. A *family of predicate transformers* for E consists of a predicate transformer τ^D for each $D \subseteq \mathcal{E}$, such that if $C \cap E \subseteq D$ then $\tau^C(\psi)$ implies $\tau^D(\psi)$.

[Predicates with smaller subsets of E are stronger.]

Definition 12. A pomset with predicate tansformers is a pomset with preconditions, together with a family of predicate transformers for E.

THRD converts a pomset with predicate transformers into a pomset with preconditions by dropping the predicate transformer. For the reverse embedding, *FORK* adopts the identity transformer.

Definition 13. If $P \in \mathit{THRD}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$

T1) $E = E_1$,
T2) $\lambda(e) = \lambda_1(e)$,
T3) $\kappa(e)$ implies $\kappa_1(e)$.

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

1) $E = E_1$,
2) $\lambda(e) = \lambda_1(e)$,
3) $\kappa(e)$ implies $\kappa_1(e)$,
4) $\tau^D(\psi)$ implies ψ.

Definition 14. Adopting *NIL* and $\|$ from Def. 8, the semantics of thread groups is:

$$[\![S]\!] = \mathit{THRD}[\![S]\!] \quad [\![G_1 \,\mathbb{I}\, G_2]\!] = [\![G_1]\!] \,\|\, [\![G_2]\!] \quad [\![\mathtt{0}]\!] = \mathit{NIL}$$

Definition 15. If $P \in \mathit{ABORT}$ then $E = \emptyset$ and

- $\tau^D(\psi)$ implies ff.

If $P \in \mathit{SKIP}$ then $E = \emptyset$ and

- $\tau^D(\psi)$ implies ψ.

If $P \in \mathit{LET}(r, M)$ then $E = \emptyset$ and

- $\tau^D(\psi)$ implies $\psi[M/r]$.

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

1–8) as for *IF* in Def. 8,
9) $\tau^D(\psi)$ implies $(\phi \Rightarrow \tau_1^D(\psi)) \wedge (\neg\phi \Rightarrow \tau_2^D(\psi))$.

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

1–5) as for $\|$ in Def. 4 (ignoring disjointness),
6) if $e \in E_1 \setminus E_2$ then $\kappa(e)$ implies $\kappa_1(e)$,
7) if $e \in E_2 \setminus E_1$ then $\kappa(e)$ implies $\kappa_2'(e)$,
8) if $e \in E_1 \cap E_2$ then $\kappa(e)$ implies $\kappa_1(e) \vee \kappa_2'(e)$, where $\kappa_2'(e) = \tau_1^C(\kappa_2(e))$, where $C = \{c \mid c < e\}$,
9) $\tau^D(\psi)$ implies $\tau_1^D(\tau_2^D(\psi))$.

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

S1) if $d, e \in E$ then $d = e$,
S2) $\lambda(e) = \mathsf{W}xv$,
S3) $\kappa(e)$ implies $M{=}v$,
S4) $\tau^D(\psi)$ implies $\psi[M/x]$,
S5) $\tau^C(\psi)$ implies $\psi[M/x]$,
where $D \cap E \neq \emptyset$ and $C \cap E = \emptyset$.

If $P \in \mathit{LOAD}(r, x, \mu)$ then either $E \neq \emptyset$ and $(\exists v \in \mathcal{V})$

L1) if $d, e \in E$ then $d = e$,
L2) $\lambda(e) = \mathsf{R}xv$,
L3) $\kappa(e)$ implies $\mathtt{tt}$,
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $(v{=}r \vee x{=}r) \Rightarrow \psi$,
where $D \cap E \neq \emptyset$ and $C \cap E = \emptyset$,

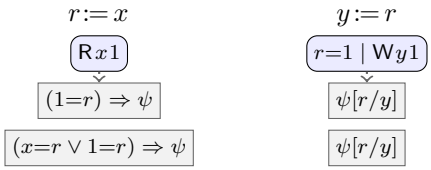
or $E = \emptyset$ and

L6) $\tau^B(\psi)$ implies ψ.

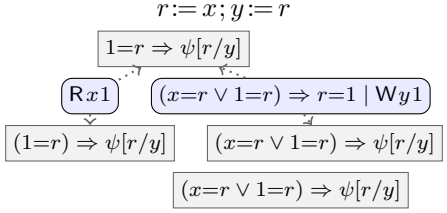
Definition 16. The semantics of commands is:

$$[\![\mathtt{if}\,(M)\,\{S_1\}\,\mathtt{else}\,\{S_2\}]\!] = \mathit{IF}(M{\neq}0, [\![S_1]\!], [\![S_2]\!])$$
$$[\![x^\mu := M]\!] = \mathit{STORE}(x, M, \mu) \qquad [\![\mathtt{abort}]\!] = \mathit{ABORT}$$
$$[\![r := x^\mu]\!] = \mathit{LOAD}(r, x, \mu) \qquad [\![\mathtt{skip}]\!] = \mathit{SKIP}$$
$$[\![r := M]\!] = \mathit{LET}(r, M) \qquad [\![\mathtt{fork}\ G]\!] = \mathit{FORK}[\![G]\!]$$
$$[\![S_1; S_2]\!] = [\![S_1]\!] \,;[\![S_2]\!]$$

Example 1. Read to write dependency, first separately:



Putting these together without order:



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

S1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
S2) $\lambda(e) = (\mathsf{W}\,[\ell_e]\,v_e)$,
S3) $\kappa(e)$ implies $\theta_e \wedge \mathsf{qs}_\mu^{[\ell_e]} \wedge L{=}\ell_e \wedge M{=}v_e$,
S4) $(\forall k)(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow (L{=}k) \Rightarrow ((\mathsf{Q}_{\mathsf{wo}}^{[k]} \Rightarrow M{=}v_e) \wedge \mathsf{ds}_\mu^{[k]}\psi[M/[k]])$,
S5) $(\forall k)\ \tau^C(\psi)$ implies $(\nexists e \in E \cap C \mid \theta_e) \Rightarrow (L{=}k) \Rightarrow (\neg\mathsf{Q}_{\mathsf{wo}}^{[k]} \wedge \mathsf{ds}_\mu^{[k]}\psi[M/[k]])$.

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

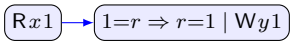
L1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
L2) $\lambda(e) = (\mathsf{R}\,[\ell_e]\,v_e)$,
L3) $\kappa(e)$ implies $\theta_e \wedge \mathsf{ql}_\mu^{[\ell_e]} \wedge L{=}\ell_e$,
L4) $(\forall k)(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow (L{=}k) \Rightarrow (v_e{=}s_e) \Rightarrow \psi[s_e/r]$,
L5) $(\forall k)(\forall e \in E \setminus C)\ \tau^C(\psi)$ implies $\theta_e \Rightarrow (L{=}k) \Rightarrow (\neg\mathsf{Q}_{\mathsf{rw}}^{[k]} \wedge \mathsf{dl}_\mu^{[k]} \wedge (\mathsf{W} \Rightarrow (v_e{=}s_e \vee [k]{=}s_e) \Rightarrow \psi[s_e/r]))$,
L6) $(\forall k)(\forall s)\ \tau^B(\psi)$ implies $(\nexists e \in E \mid \theta_e) \Rightarrow (L{=}k) \Rightarrow (\neg\mathsf{Q}_{\mathsf{rw}}^{[k]} \wedge \mathsf{dl}_\mu^{[k]} \wedge \psi[s/r])$.

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

T1) $E = E_1$,
T2) $\lambda(e) = \lambda_1(e)$,
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.

Figure 1. Full Semantics of Loads, Stores and Threads (See Def. 20 for qs/ql and Def. 22 for ds/dl)

If the read is ordered before the write, then the precondition of the write can be weakened:



3. Quiescence

Quiescence is used to capture termination, coherence, release-acquire access, and SC access.

3.1. Coherence (CO)

In order to describe coherence, we use the uninterpreted logical symbols $\mathsf{Q}_{\mathsf{rw}}^x$ and $\mathsf{Q}_{\mathsf{wo}}^x$, for each location x, with the interpretation that $\mathsf{Q}_{\mathsf{rw}}^x$ implies $\mathsf{Q}_{\mathsf{wo}}^y$ when $x = y$.

When $\mathsf{Q}_{\mathsf{rw}}^x$ is discharged in the precondition of event e, we expect that all reads and writes of x that precede e in program order also precede e in pomset order. $\mathsf{Q}_{\mathsf{wo}}^x$ is similar, but applies only to preceding writes.

Definition 17. Let $[\mathsf{tt}/\mathsf{Q}]$ be the substitution that replaces all quiescence symbols replaced by tt.

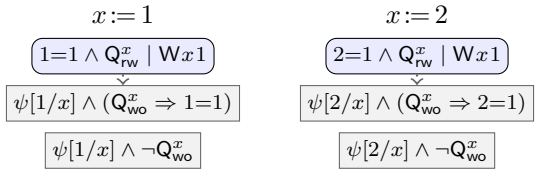
We repeat L4, although it is unchanged from Def. 15.

Definition 18 (CO)**.** Update Def. 15 to:

S3) $\kappa(e)$ implies $\mathsf{Q}_{\mathsf{rw}}^x \wedge M{=}v$,
L3) $\kappa(e)$ implies $\mathsf{Q}_{\mathsf{wo}}^x$,
T3) $\kappa(e)$ implies $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}]$,
S4) $\tau^D(\psi)$ implies $(\mathsf{Q}_{\mathsf{wo}}^x \Rightarrow M{\neq}v) \wedge \psi[M/x]$,
S5) $\tau^C(\psi)$ implies $\neg\mathsf{Q}_{\mathsf{wo}}^x \wedge \psi[M/x]$.
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $\neg\mathsf{Q}_{\mathsf{rw}}^x \wedge ((v{=}r \vee x{=}r) \Rightarrow \psi)$,
L6) $\tau^B(\psi)$ implies $\neg\mathsf{Q}_{\mathsf{rw}}^x \wedge \psi$.

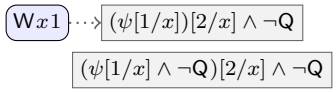
Definition 19. P is *quiescent* if $\tau^E(\mathsf{Q})$ implies Q.

Example 2. Merging left.

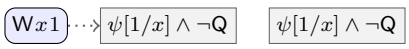


Simplifying:

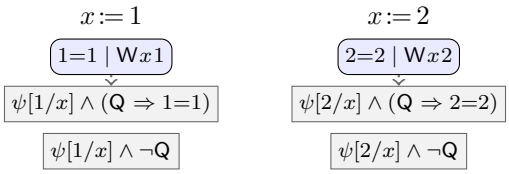
Merging the actions, we have:



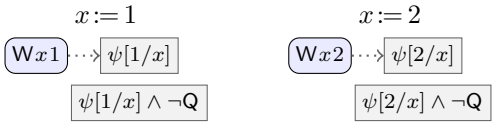
which simplifies to



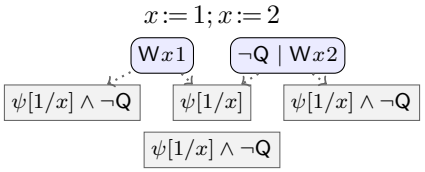
Example 3. Separate actions:



Simplifying:



Putting these together unordered:

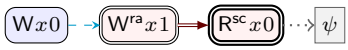


Adding order does nothing since the preconditions are tautologies.

3.2. RA and SC Access (RASC)

In order to describe ra/sc access, we use the uninterpreted logical symbols Q_{ra} and Q_{sc}, with the interpretation that Q_{sc} implies Q_{ra} implies $\mathsf{Q}^x_{\mathsf{rw}}$ implies $\mathsf{Q}^y_{\mathsf{wo}}$ when $x = y$.

[visualization. Labels to be turned off later in macros]



Definition 20. Let qs^x_μ and ql^x_μ be defined:

$$\mathsf{qs}^x_{\mathsf{rlx}} = \mathsf{Q}^x_{\mathsf{rw}} \qquad \mathsf{ql}^x_{\mathsf{rlx}} = \mathsf{Q}^x_{\mathsf{wo}}$$
$$\mathsf{qs}^x_{\mathsf{ra}} = \mathsf{Q}_{\mathsf{ra}} \qquad \mathsf{ql}^x_{\mathsf{ra}} = \mathsf{Q}^x_{\mathsf{wo}}$$
$$\mathsf{qs}^x_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{sc}} \qquad \mathsf{ql}^x_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{sc}}$$

Definition 21 (CO/RASC)**.** Update Def. 18 to:

S3) $\kappa(e)$ implies $\mathsf{qs}^x_\mu \wedge M{=}v$,
L3) $\kappa(e)$ implies ql^x_μ.

4. Efficient Implementation on ARMv8

4.1. Down-Grading Acquires (DGA)

We use the uninterpreted symbols $\downarrow^x$.

Definition 22. Let $[\mathsf{ff}/\downarrow^*]$ be the substitution that performs $[\mathsf{ff}/\downarrow^x]$ for every x. Let ds^x_μ and dl^x_μ be defined:

$$\mathsf{ds}^x_{\mathsf{rlx}}\psi = \psi[\mathsf{tt}/\downarrow^x] \qquad \mathsf{dl}^x_{\mathsf{rlx}} = \mathsf{tt}$$
$$\mathsf{ds}^x_{\mathsf{ra}}\psi = \psi[\mathsf{ff}/\downarrow^*] \qquad \mathsf{dl}^x_{\mathsf{ra}} = \downarrow^x$$
$$\mathsf{ds}^x_{\mathsf{sc}}\psi = \psi[\mathsf{ff}/\downarrow^*] \qquad \mathsf{dl}^x_{\mathsf{sc}} = \downarrow^x$$

Definition 23 (CO/RASC/DGA)**.** Update Def. 21 to:

S4) $\tau^D(\psi)$ implies $(\mathsf{Q}^x_{\mathsf{wo}} \Rightarrow M{\neq}v) \wedge \mathsf{ds}^x_\mu\psi[M/x]$,
S5) $\tau^C(\psi)$ implies $\neg\mathsf{Q}^x_{\mathsf{wo}} \wedge \mathsf{ds}^x_\mu\psi[M/x]$.
L5) $\tau^C(\psi)$ implies $\neg\mathsf{Q}^x_{\mathsf{rw}} \wedge \mathsf{dl}^x_\mu \wedge ((v{=}r \vee x{=}r) \Rightarrow \psi)$,
L6) $\tau^B(\psi)$ implies $\neg\mathsf{Q}^x_{\mathsf{rw}} \wedge \mathsf{dl}^x_\mu \wedge \psi$.

4.2. Read-Read dependencies (RRD)

We use the uninterpreted symbol W.

Definition 24 (RRD)**.** Update Def. 15 to:

T3) $\kappa(e)$ implies $\kappa_1(e)[\mathsf{tt}/\mathsf{W}]$ if $\lambda_1(e)$ is a write,
$\kappa(e)$ implies $\kappa_1(e)[\mathsf{ff}/\mathsf{W}]$ otherwise.
L5) $\tau^C(\psi)$ implies $\mathsf{W} \Rightarrow (v{=}r \vee x{=}r) \Rightarrow \psi$,

We repeat all of the rules for preconditions and transformers, Only L5 and T3 are changed from Def. 23. We repeat L4, although it is unchanged from Def. 15.

Definition 25 (CO/RASC/DGA/RRD)**.** Update Def. 15 to:

S3) $\kappa(e)$ implies $\mathsf{qs}^x_\mu \wedge M{=}v$,
L3) $\kappa(e)$ implies ql^x_μ.
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.
S4) $\tau^D(\psi)$ implies $(\mathsf{Q}^x_{\mathsf{wo}} \Rightarrow M{\neq}v) \wedge \mathsf{ds}^x_\mu\psi[M/x]$,
S5) $\tau^C(\psi)$ implies $\neg\mathsf{Q}^x_{\mathsf{wo}} \wedge \mathsf{ds}^x_\mu\psi[M/x]$.
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $\neg\mathsf{Q}^x_{\mathsf{rw}} \wedge \mathsf{dl}^x_\mu \wedge (\mathsf{W} \Rightarrow (v{=}r \vee x{=}r) \Rightarrow \psi)$,
L6) $\tau^B(\psi)$ implies $\neg\mathsf{Q}^x_{\mathsf{rw}} \wedge \mathsf{dl}^x_\mu \wedge \psi$.

[Control dependencies into reads as in MP with release on right and control dependency on left.]

4.3. Proof Strategy

Alternate characterization of ARM: [3, §B2.3.6] [1] [2]

5. Further Complications

[I have a note: TC1: Track local state ???]

5.1. If Closure (IF)

Requires indexing to resolve nondeterminism.
IF closure/case analysis: ψ_e

Definition 26 (IF)**.** Update Def. 15 to:
If $P \in \mathit{STORE}(x, M, \mu)$ then $(\exists v : E{\to}\mathcal{V})\ (\exists\theta : E{\to}\Phi)$

S1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
S2) $\lambda(e) = (\mathsf{W}\,[\ell_e]\,v_e)$,
S3) $\kappa(e)$ implies $\theta_e \wedge M{=}v$,
S4) $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/x]$,
S5) $\tau^C(\psi)$ implies $(\nexists e \in E \cap C \mid \theta_e) \Rightarrow \psi[M/x]$,

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

L1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
L2) $\lambda(e) = (\mathsf{R}\,[\ell_e]\,v_e)$,
L3) $\kappa(e)$ implies θ_e.
L4) $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow v_e{=}s_e \Rightarrow \psi[s_e/r]$,
L5) $(\forall e \in E \setminus C)\ \tau^C(\psi)$ implies $\theta_e \Rightarrow (v_e{=}s_e \vee x{=}s_e) \Rightarrow \psi[s_e/r]$,
L6) $(\forall s)\ \tau^B(\psi)$ implies $(\nexists e \in E \mid \theta_e) \Rightarrow \psi[s/r]$.

5.2. Address Calculation (ADDR)

Do this after if closure, because problem with punning badly.

Definition 27 (ADDR)**.** Update Def. 15 to:

S2) $\lambda(e) = \mathsf{W}\,[\ell]\,v$,
L2) $\lambda(e) = \mathsf{R}\,[\ell]\,v$.

S3) $\kappa(e)$ implies $L{=}\ell \wedge M{=}v$,
L3) $\kappa(e)$ implies $L{=}\ell$.

S4) $(\forall k)\ \tau^D(\psi)$ implies $L{=}k \Rightarrow \psi[M/[k]]$,
S5) $(\forall k)\ \tau^C(\psi)$ implies $L{=}k \Rightarrow \psi[M/[k]]$,
L4) $(\forall k)\ \tau^D(\psi)$ implies $L{=}k \Rightarrow v{=}r \Rightarrow \psi$,
L5) $(\forall k)\ \tau^C(\psi)$ implies $L{=}k \Rightarrow (v{=}r \vee [k]{=}r) \Rightarrow \psi$,
L6) $(\forall k)\ \tau^B(\psi)$ implies $L{=}k \Rightarrow \psi$.

6. Discussion

6.1. Relation to Traditional Predicate Transformers

Proposition 1. *If $P \in [\![S]\!]$ is top-level and quiescent then $\tau^E(\psi)$ implies $wp_S(\psi)$.*

For any substitution $\sigma = [v_1/r_1, \ldots, v_n/r_n]$ there is some $P \in [\![S]\!]$ such that all preconditions in $P\sigma$ are tautologies then $wp_S(\psi)\sigma$

For a language where all programs are terminating, we have for any statement S:

$$\{\phi\}\ S\ \{\psi\} \quad \Leftrightarrow \quad \phi \text{ implies } wp_S(\psi)$$

Interpretation is that if $\sigma \models wp_S(\psi)$ and $(\sigma, S) \Downarrow \rho$ then $\rho \models \psi$.

Let S_0 be $x_1 := v_1; \cdots; x_n := v_n$, such that $wp_{S_0}(\phi)$ is a tautology, and $x_i = x_j$ implies $i = j$.

Let $\sigma_P = [v_1/x_1, \ldots, v_n/x_n]$ be the final state of P.

For example, let $S_1 = r{:=}x$ and $S_2 = x{:=}r{+}1$ and $S = S_1; S_2$.

$$wp_{S_2}(x{>}1) = (r{+}1{>}1) = (r{>}0)$$
$$wp_{S_1}(r{>}0) = wp_{S_0}(x{>}1) = (x{>}0)$$

Let $P_i \in [\![S_i]\!]$.

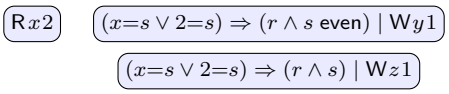
$$\tau_2^{E_2}(x{>}1) = (r{+}1{>}1) = (r{>}0)$$
$$\tau_0^{E_0}(x{>}1) = (0{=}r \Rightarrow r{>}0)$$
$$\tau_0^{E_0}(x{>}1) = (1{=}r \Rightarrow r{>}0)$$
$$\tau_0^{E_0}(x{>}1) = (2{=}r \Rightarrow r{>}0)$$

Proposition 2. *If $P \in [\![S]\!]$ is top-level and quiescent then $\tau^E(\phi)$ implies $wp_S(\phi)$.*

For any substitution $\sigma = [v_1/r_1, \ldots, v_n/r_n]$ there is some $P \in [\![S]\!]$ such that all preconditions in $P\sigma$ are tautologies then $wp_S(\phi)\sigma$

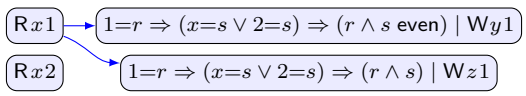
6.2. [r/x] v [x/r]

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

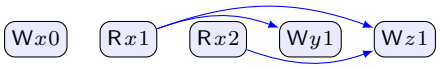


Without substitution:

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

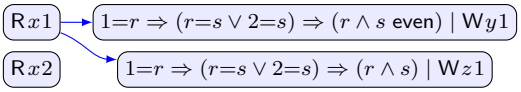


Prepending $x := 0$

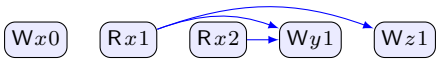


With the substitution $[r/x]$:

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



Prepending $x := 0$



6.3. Fork-Join

It is also possible to put coherence in the independency relation, in which case, the semantics of ; includes the following.

10) if $d \in E_1$ and $e \in E_2$ either $d < e$ or $a \leftrightarrow \lambda_2(e)$.

One must be careful, however, due to *inconsistency*. Consider that `x=0;x=1` should not have completed pomset with only $(\mathsf{W}x0)$.

(10) does not do the right thing with fork either. If you want to enforce coherence this way then you need to use fork-join as the sequential combinator, rather than fork.

[We drop $\leftrightarrow$ because incompatible with *FORK*. If you want to use $\leftrightarrow$, then you need to use fork-join as the sequential combinator, rather than fork.]

Definition 28. A *pomset with preconditions and termination* is a pomset with preconditions together with a predicate ✓.

Definition 29.
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 Definition 8,
9) ✓ implies $\checkmark_1 \wedge \checkmark_2$.

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

??–??) as for *THRD* in Definition **??**,
1) if ✓ then $\tau^E(\mathsf{Q})$ implies Q.

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

??–??) as for *FORK* in Definition **??**,
F5) $\checkmark_1$.

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

We can then encode coherence as follows.

10) if $d \in E_1$ and $e \in E_2$ either $d < e$ or $a \leftrightarrow \lambda_2(e)$.

Access modes can be encoded in the independency relation, indexing labels by μ, but the extra flexibility of the logic is necessary for ARM8 (see §4.1). Using independency, one would also need another way to define completed pomsets. Finally, this use of independency is incompatible with fork (see §3.1).

If we move coherence to independency (and use fork-join), we have the following, assuming that each register occurs at most once.

$$\begin{array}{lll} \mathsf{qs}_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{sc}} & \mathsf{qs}_{\mathsf{ra}} = \mathsf{Q}_{\mathsf{ra}} & \mathsf{qs}_{\mathsf{rlx}} = \mathsf{Q}^x_{\mathsf{rw}} \\ \mathsf{ql}_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{sc}} & \mathsf{ql}_{\mathsf{ra}} = \mathsf{Q}^x_{\mathsf{wo}} & \mathsf{ql}_{\mathsf{rlx}} = \mathsf{Q}^x_{\mathsf{wo}} \\ \mathsf{ds}^x_{\mathsf{sc}}\psi = \psi[\mathsf{ff}/{\downarrow^*}] & \mathsf{ds}^x_{\mathsf{ra}}\psi = \psi[\mathsf{ff}/{\downarrow^*}] & \mathsf{ds}^x_{\mathsf{rlx}}\psi = \psi[\mathsf{tt}/{\downarrow^x}] \\ \mathsf{dl}^x_{\mathsf{sc}} = {\downarrow^x} & \mathsf{dl}^x_{\mathsf{ra}} = {\downarrow^x} & \mathsf{dl}^x_{\mathsf{rlx}} = \mathsf{tt} \end{array}$$

If $P \in \mathit{STORE}(x, M, \mu)$ then

S1–S2) as before,
S3) $\kappa(e)$ implies $M{=}v \wedge \mathsf{W} \wedge \mathsf{qs}_\mu$,
S4) $\tau^D(\psi)$ implies $M{=}v \wedge \mathsf{ds}^x_\mu\psi[M/x]$,
S5) $\tau^\emptyset(\psi)$ implies $\neg\mathsf{Q}_{\mathsf{ra}} \wedge \mathsf{ds}^x_\mu\psi[M/x]$

If $P \in \mathit{LOAD}(r, x, \mu)$ then

L1–L2) as before,
L3) $\kappa(e)$ implies $\neg\mathsf{W} \wedge \mathsf{ql}_\mu$,
L4) $\tau^D(\psi)$ implies $(v{=}r) \Rightarrow \psi[r/x]$
L5) $\tau^\emptyset(\psi)$ implies $\mathsf{dl}^x_\mu \wedge \neg\mathsf{Q}_{\mathsf{ra}} \wedge (\mathsf{W} \Rightarrow (v{=}r \vee x{=}r) \Rightarrow \psi[r/x])$.

6.4. Must Allow Inconsistent Preconditions

Removing the requirements for *consistency* and *causal strengthening*, and

[The definition does not give a sensible notion of completed execution without consistency and causal strengthening.]

6.5. Skolemization

[7] is non-skolemized, using substitution instead, and collapsing x and r. There, item 7 of *LD* is written

> if $e \in E_2 \setminus E_1$ then either
> $\kappa(e)$ implies $\kappa_2(e)[x/r][v/x]$ and $(\exists d \in E_1)d{<}e$, or
> $\kappa(e)$ implies $\kappa_2(e)[x/r][v/x] \wedge \kappa_2(e)[x/r]$.

[7] is non-skolemized—with $[x/r]$ rather than no substitution.

L4) $\tau^D(\psi)$ implies $\psi[x/r][v/x]$,
L5) $\tau^\emptyset(\psi)$ implies $\psi[x/r][v/x] \wedge \psi[x/r]$,
L6) $\tau^\emptyset(\psi)$ implies $\psi[x/r]$.

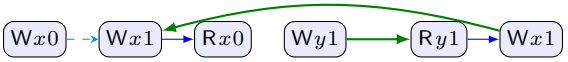
[Skolemization ensures disjunction closure, which is necessary for associativity. Show example.]

6.6. Reads Update Local State

In the rule for read prefixing we have substituted $[r/x]$, rather than $[x/r]$. This means that reads clobber local state. We assume registers are only used once—otherwise, one needs to generate a fresh register for the substitution.

With read-read dependencies, this difference can be seen. For example, the following execution is allowed with $[x/r]$, but not $[r/x]$.

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



[Is there a difference w/o read-read dependencies?]

[Don't need extended expressions anymore, since never substituting with x for anything.]

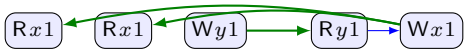
6.7. Parallel Composition

In [7, §2.4], parallel composition is defined allowing coalescing of events. Here we have forbidden coalescing. This difference appears to be arbitrary. In [7], however, there is a mistake in the handling of termination actions. The predicates should be joined using $\wedge$, not $\vee$.

6.8. Redundant Read Elimination

Requires indexing to resolve nondeterminism.

$$r{:=}x; s{:=}x; \texttt{if}\ (r{=}s)\ \{y{:=}1\} \parallel x{:=}y \qquad \text{(TC2)}$$



Precondition of $(\mathsf{W}y1)$ is $(r{=}s)$ in $[\![\texttt{if}\ (r{=}s)\ \{y{:=}1\}]\!]$. Predicate transformers for $\emptyset$ in $[\![r{:=}x]\!]$ and $[\![s{:=}x]\!]$ are

$$\langle (r{=}1 \vee r{=}x) \Rightarrow \psi[r/x] \mid \phi\rangle,$$
$$\langle (s{=}1 \vee s{=}x) \Rightarrow \psi[s/x] \mid \phi\rangle.$$

Combining the transformers, we have

$$\langle (r{=}1 \vee r{=}x) \Rightarrow (s{=}1 \vee s{=}r) \Rightarrow \psi[s/x] \mid \phi\rangle.$$

Applying this to $(r{=}s)$, we have

$$\langle (r{=}1 \vee r{=}x) \Rightarrow (s{=}1 \vee s{=}r) \Rightarrow (r{=}s) \mid \phi\rangle,$$

which is not a tautology.

Same problem occurs [7], where we have:

$$\langle \psi[v/x, r] \wedge \psi[x/r] \mid \phi\rangle,$$
$$\langle \psi[v/x, s] \wedge \psi[x/s] \mid \phi\rangle.$$

Combining the transformers, we have

$$\langle \psi[v/x, r, s] \wedge \psi[v/x, r][x/s] \wedge \psi[x/r][v/x, s] \wedge \psi[x/r, s] \mid \phi\rangle.$$

Applying this to $(r{=}s)$, we have

$$\langle v{=}v \wedge v{=}x \wedge x{=}v \wedge x{=}x \mid \phi \rangle,$$

which is not a tautology.

The semantics here allows this by coalescing:

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

6.9. Redundant Read Elimination

In [7, §2.6] the semantics of read is defined as follows:

$$[\![r{:=}x^{\mu}; S]\!] \triangleq \bigcup_v (\mathsf{R}^{\mu} x v) \Rightarrow [\![S]\!][x/r]$$

The definition of prefixing$((\phi \mid a) \Rightarrow \mathcal{P})$ has several clauses. The most relevant are as follows, where d is the new event labeled with $(\phi \mid a)$ and e is an event from $\mathcal{P}$:

(P4C) If d reads v from x then either $e = d$ or $\kappa'(e)$ implies $\kappa(e)[v/x]$.

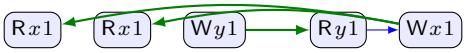
(P5A) If d reads and e writes then either $\kappa'(e)$ implies $\kappa(e)$ or $d \leq' e$.

We have discovered two issues with this definition.

The first issue concerns the substitution $[x/r]$. It should be $[r/x]$. We noticed this error while developing the alternative characterization presented here. The error causes redundant read elimination to fail in [7]. As a result, common subexpression elimination also fails. The problem can be seen in TC2.

$$r{:=}x; s{:=}x; \texttt{if}\ (r{=}s)\ \{y{:=}1\} \parallel x{:=}y \qquad \text{(TC2)}$$

We claimed that TC2 allowed the following execution:



But this execution is not possible using the semantics of [7]: $(\mathsf{W}y1)$ has precondition $r{=}s$ in $[\![\texttt{if}\ (r{=}s)\ \{y{:=}1\}]\!]$. Given the lack of order in the execution, the precondition of $(\mathsf{W}y1)$ must entail $r{=}1 \wedge r{=}x$ in $[\![s{:=}x; \texttt{if}\ (r{=}s)\ \{y{:=}1\}]\!]$. P4C imposes $r{=}1$, and P5A imposes $r{=}x$. Adding the second read, the precondition of $(\mathsf{W}y1)$ must entail both $1{=}1 \wedge 1{=}x$ and also $x{=}1 \wedge x{=}x$. This can be simplified to $x{=}1$. This leaves a requirement that must be satisfied by a preceding write. Since the preceding write is the initialization to 0, the requirement cannot be satisfied, and the execution is impossible.[1]

The substitution $[x/r]$ leaves the obligation on x to be fulfilled by the preceding write. Thus, the read does not update the *value* of x in subsequent predicates. The substitution $[r/x]$, instead, does update the value of x, thus removing any obligation on x for preceding code.

In order to write this, we must update the definition of prefixing reads to include the register. Then P4C becomes:

(P4C) If d reads v from x then either $e = d$ or $\kappa'(e)$ implies $\kappa(e)[v/r]$.

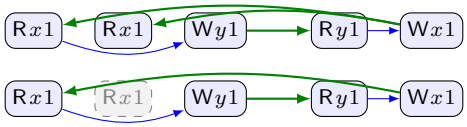
We can then reason with TC2 as follows: $(\mathsf{W}y1)$ has precondition $r{=}s$ in $[\![\texttt{if}\ (r{=}s)\ \{y{:=}1\}]\!]$. To avoid introducing order in the execution, the precondition of $(\mathsf{W}y1)$ must entail $r{=}1 \wedge r{=}s$ in $[\![s{:=}x; \texttt{if}\ (r{=}s)\ \{y{:=}1\}]\!]$. P4C imposes $r{=}1$, and P5A imposes $r{=}x$. Adding the second read, the precondition of $(\mathsf{W}y1)$ must entail both $1{=}1 \wedge 1{=}x$ and also $x{=}1 \wedge x{=}x$. This can be simplified to $x{=}1$. This leaves a requirement that must be satisfied by a preceding write.

With read elimination, the rule for relaxed reads is as follows:

$$[\![r{:=}x; S]\!] \triangleq [\![S]\!][x/r] \cup \bigcup_v (\mathsf{R}x v) \Rightarrow_r [\![S]\!][r/x]$$

It is interesting to note that the substitution is $[x/r]$ on eliminated reads, and $[r/x]$ on non-eliminated reads. Intuitively, the subsequent value of x is fixed by an explicit read, but not for an eliminated read. In the latter case, the value is fixed by some preceding action. The preceding action may itself be a read. This gives rise to some fear that we might introduce thin-air reads, since we do not enforce read-read coherence. But this is not the case. Consider the following example:

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



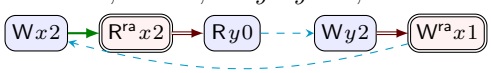
But this is not a problem, since fulfillment requires that $(\mathsf{W}x1)$ precede both reads of x.

6.10. Internal Acquiring Reads

Our 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.

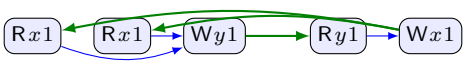
The second issue concerns acquiring reads. Shortly after publication, Podkopaev [8] noticed a shortcoming of the implementation on ARM8 in [7, §7]. The proof given there assumes that all internal reads can be dropped. However, this is not the case for acquiring reds. For example, [7] disallows the following execution, which is allowed by ARM8 and TSO.

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



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. Backporting this solution to [7] requires that we add access predicates to the logic and allow

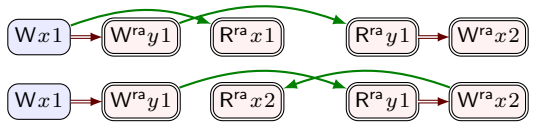
1. In [7] we ignore the middle terms, mistakenly simplifying this to $1{=}1 \wedge x{=}x$. Correcting the error, the attempted execution is:



6.11. Triangular Races

The notion of data-race is incorrect in [7].

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



Bug is in [5, Lemma A.4]. It assumes that $(\mathsf{R}^{\mathsf{ra}}x1)$ and $(\mathsf{W}^{\mathsf{ra}}x2)$ are racing in the first execution because they are not ordered by happens-before. But this is false since neither is plain.

In addition, the ARM8 implementation result given here does not rely on read elimination. Instead we use a recent alternative characterization of ARM8 [1, 3, 2].

7. Outro

References

[1] J. Alglave. This commit adds three alternative formulations of the arm model, both for non-mixed and mixed size accesses. https://github.com/herd/herdtools7/commit/685ee4b5f821254c947888c6cc731e9eedbe937d, June 2020.

[2] J. Alglave, W. Deacon, R. Grisenthwaite, A. Hacquard, and L. Maranget. Armed cats: Formal concurrency modelling at arm. Draft, 2020.

[3] Arm Limited. Arm architecture reference manual: Armv8, for Armv8-A architecture profile (issue F.c). https://developer.arm.com/documentation/ddi0487/latest, July 2020.

[4] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18 (8):453–457, 1975. doi: 10.1145/360933.360975. URL https://doi.org/10.1145/360933.360975.

[5] B. Dongol, R. Jagadeesan, and J. Riely. Modular transactions: bounding mixed races in space and time. In J. K. Hollingsworth and I. Keidar, editors, *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, pages 82–93. ACM, 2019. doi: 10.1145/3293883.3295708. URL https://doi.org/10.1145/3293883.3295708.

[6] C. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

[7] R. Jagadeesan, A. Jeffrey, and J. Riely. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.*, 4(OOPSLA):194:1–194:30, 2020. doi: 10.1145/3428262. URL https://doi.org/10.1145/3428262.

[8] A. Podkopaev. Private correspondence, Nov. 2020.