Sequential Composition for Relaxed Memory

Alan Jeffrey* and James Riely†
*The Servo Project and Roblox
†DePaul University

1. 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)`

1.1. 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,
- for any set E there are registers $\mathcal{S}_E = \{s_e \mid e \in E\}$,
- expressions include at least registers and values,
- expressions do *not* include memory locations or registers in $\mathcal{S}_E$, for any set E.

We model the following language.

$$\begin{array}{rl} \mu & ::= \mathsf{rlx} \mid \mathsf{ra} \mid \mathsf{sc} \\ C,\, D & ::= \mathtt{skip} \mid r{:=}\, M \mid r{:=}\, [L]^{\mu} \mid [L]^{\mu}{:=}\, M \\ & \quad \mid \mathtt{fork}\, G \mid C; D \mid \mathtt{if}\,(M)\,\{C\}\,\mathtt{else}\,\{D\} \\ G,\, H & ::= 0 \mid \mathtt{thread}\, C \mid G \parallel H \end{array}$$

Memory modes, μ, are relaxed (rlx), release-acquire (ra), and sequentially consistent (sc). Relaxed is the default. *Commands*, C, include reads from and writes to memory 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. We often drop the words `fork` and `thread`.

The semantics is built from the following.

- a set of *actions* $\mathcal{A}$, ranged over by a,
- a set of *logical formulae* Φ, ranged over by ϕ, ψ, χ.

We require that

- actions include writes $(\mathsf{W}xv)$ and reads $(\mathsf{R}xv)$,
- formulae include equalities $(M{=}N)$ and $(M{=}x)$,
- formulae are closed under negation, conjunction, disjunction, and substitutions $[M/r]$ and $[M/x]$,
- there is an entailment relation $\vDash$ between formulae, with the expected semantics.

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 $[M/x]$; actions are not.

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

1.2. 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 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.

Independency $(\leftrightarrow \subseteq \mathcal{A} \times \mathcal{A})$ is defined as follows.

$$\begin{array}{l} \leftrightarrow = \{(\mathsf{R}xv, \mathsf{W}yw), (\mathsf{W}xv, \mathsf{R}yw), (\mathsf{W}xv, \mathsf{W}yw) \mid x \neq y\} \\ \quad \cup \{(\mathsf{R}xv, \mathsf{R}yw)\} \end{array}$$

In order to give the semantics, we define several operators over sets of pomsets.

Definition 3.
If $P \in \mathit{STOP}$ 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 $a \leftrightarrow \lambda_2(e)$.

Using these operators, we can give the semantics for a simple fragment of our language.

$$\begin{aligned}
[\![\texttt{skip}]\!] = [\![\texttt{0}]\!] &= \mathit{STOP} \\
[\![G \parallel H]\!] &= [\![G]\!] \parallel [\![H]\!] \\
[\![x{:=}\, v; C]\!] &= (\mathsf{W}xv) \to [\![C]\!] \\
[\![r{:=}\, x; C]\!] &= \textstyle\bigcup_v (\mathsf{R}xv) \to [\![C]\!]
\end{aligned}$$

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

[Do Examples.]

[Do examples with coherence.]

[Note that this allows mumbling for reads and writes.]

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

[Talk about Mazurkiewicz traces.]

1.3. Pomsets with Preconditions

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

The model described here is essentially the model of Jagadeesan et al. [2020], restricting attention to relaxed access. We discuss the differences in the appendix.

Definition 4. A *pomset with preconditions* is a pomset together with $\kappa : E \to \Phi$.

Definition 5. A pomset with preconditions is *top level* if it is fulfilled and every precondition is a tautology.

Definition 6. Let σ be a substitution. If $P \in (\mathcal{P}\sigma)$ then $(\exists P \in \mathcal{P})\ E = E'$, $\leq = \leq'$, $\lambda = \lambda'$, and $\kappa(e) = \kappa'(e)\sigma$.

Definition 7.
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 Definition 3,
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 $P \in \mathit{IF}(\psi, \mathcal{P}_1, \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

1–5) as for $\parallel$ in Definition 3 (ignoring disjointness),
6) if $e \in E_1 \setminus E_2$ then $\kappa(e)$ implies $\psi \wedge \kappa_1(e)$,
7) if $e \in E_2 \setminus E_1$ then $\kappa(e)$ implies $\neg\psi \wedge \kappa_2(e)$,
8) if $e \in E_1 \cap E_2$ then
$\kappa(e)$ implies $(\psi \wedge \kappa_1(e)) \vee (\neg\psi \wedge \kappa_2(e))$.

If $P \in \mathit{STOREPRE}(x, M, \mathcal{P}_2)$ 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 Definition 3,
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 $P \in \mathit{LOADPRE}(r, x, \mathcal{P}_2)$ 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 Definition 3,
7) if $e \in E_2 \setminus E_1$ then either
$\kappa(e)$ implies $(r{=}v \vee r{=}x) \Rightarrow \kappa_2(e)[r/x]$ or
$\kappa(e)$ implies $(r{=}v) \Rightarrow \kappa_2(e)[r/x]$ and $d < e$
for some $d \in E_1$.

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

The semantics of `skip`, `0`, and $\parallel$ are as before.

$$\begin{aligned}
[\![\texttt{if}\,(M)\,\{C\}\,\texttt{else}\,\{D\}]\!] &= \mathit{IF}(M{\neq}0, [\![C]\!], [\![D]\!]) \\
[\![r{:=}\, M; C]\!] &= [\![C]\!][M/r] \\
[\![x{:=}\, M; C]\!] &= \mathit{STOREPRE}(x, M, [\![C]\!]) \\
[\![r{:=}\, x; C]\!] &= \mathit{LOADPRE}(r, x, [\![r]\!])
\end{aligned}$$

[Stuff about conditionals and merging events.]

1.4. Pomsets with Predicate Transformers

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

Definition 8. A *predicate transformer* is a monotone function $\tau : \Phi \to \Phi$ such that $\tau(\mathsf{ff})$ is ff, $\tau(\phi \wedge \psi)$ is $\tau(\phi) \wedge \tau(\psi)$, and $\tau(\phi \vee \psi)$ is $\tau(\phi) \vee \tau(\psi)$.

Definition 9. A *family of predicate transformers* for E consists of a predicate transformer τ^D for each set of events D, such that if $C \cap E \subseteq D$ then $\tau^C(\phi)$ implies $\tau^D(\phi)$.

[Predicates with smaller subsets of E are stronger.]

Definition 10. A pomset with predicate tansformers is a pomset with preconditions, together with a family of predicate transformers for E.

Definition 11. If $P \in \mathit{STOP}$ then $E = \emptyset$ and

1) $\tau^D(\phi)$ implies ff.

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

1) $\tau^D(\phi)$ implies ϕ.

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

1) $\tau^D(\phi)$ implies $\phi[M/r]$.

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

1–8) as for *IF* in Definition 7,
9) $\tau^D(\phi)$ implies $(\psi \wedge \tau_1^D(e)) \vee (\neg\psi \wedge \tau_2^D(\phi))$.

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

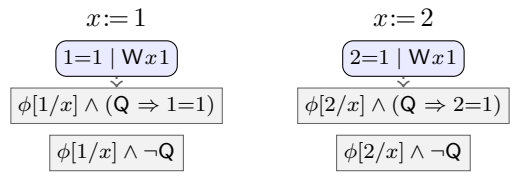
1–5) as for $\parallel$ in Definition 3 (ignoring disjointness),
6) if $e \in E_1 \setminus E_2$ then $\kappa(e)$ implies $\kappa_1(e)$,
7) if $e \in E_2 \setminus E_1$ then $\kappa(e)$ implies $\kappa_2'(e)$,
8) if $e \in E_1 \cap E_2$ then $\kappa(e)$ implies $\kappa_1(e) \vee \kappa_2'(e)$,
where $\kappa_2'(e) = \tau_1^C(\kappa_2(e))$, where $C = \{c \mid c < e\}$,
9) $\tau^D(\phi)$ implies $\tau_2^D(\tau_1^D(\phi))$.

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

S1) if $d, e \in E$ then $d = e$,
S2) $\lambda(e) = (\mathsf{W}xv)$,
S3) $\kappa(e)$ implies $M{=}v$,
S4) $\tau^D(\phi)$ implies $\phi[M/x] \wedge (\mathsf{Q} \Rightarrow M{=}v)$,
S5) $\tau^{\emptyset}(\phi)$ implies $\phi[M/x] \wedge \neg\mathsf{Q}$.

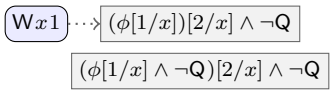
If $P \in \mathit{LOAD}(r,\, x,\, \mu)$ then $(\exists v \in \mathcal{V})\ (\forall D \neq \emptyset)$

L1) if $d, e \in E$ then $d = e$,
L2) $\lambda(e) = (\mathsf{R}xv)$,
L3) $\kappa(e)$ implies tt,
L4) $\tau^D(\phi)$ implies $(v{=}r) \Rightarrow \phi[r/x]$,
L5) $\tau^{\emptyset}(\phi)$ implies $((x{=}r \vee v{=}r) \Rightarrow \phi[r/x]) \wedge \neg\mathsf{Q}$.

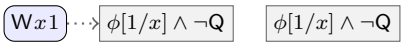


Simplifying:

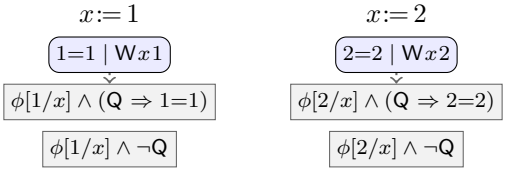
Merging the actions, we have:



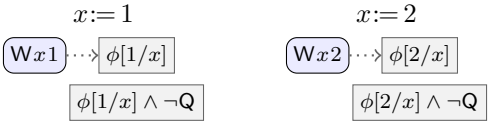
which simplifies to



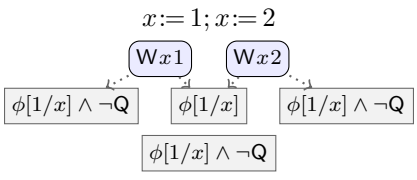
Looking at separate actions:



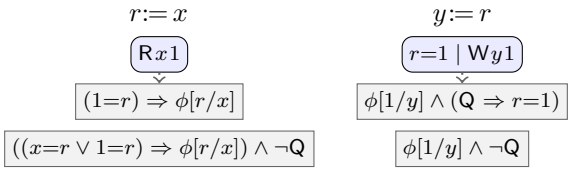
Simplifying:



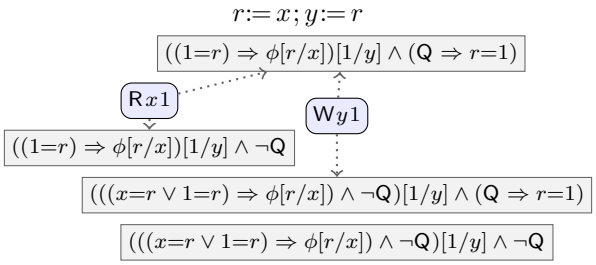
Putting these together:



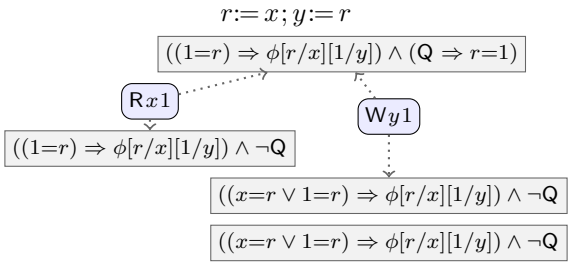
Read to write dependency, first separately:



Putting these together:



simplifying:



We have not given a semantics for parallel composition with predicate transformers. Define *THREAD* to embed pomsets with predicate transformers into pomsets with preconditions simply by dropping the predicate transformer. For the reverse embedding, *FORK* adopts the identity transformer.

Definition 12.
If $P \in \mathit{THREAD}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$

1) $E = E_1$,
2) $\lambda(e) = \lambda_1(e)$,
3) $\kappa(e)$ implies $\kappa_1(e)$.

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

F1) $E = E_1$,
F2) $\lambda(e) = \lambda_1(e)$,
F3) $\kappa(e)$ implies $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}]$,
F4) $\tau^D(\phi)$ implies ϕ.

The complete semantics is as follows.

$$
\begin{aligned}
[\![\mathtt{skip}]\!] &= \mathit{SKIP} \\
[\![r{:=}\, x^{\mu}]\!] &= \mathit{LOAD}(r,\, x,\, \mu) \\
[\![x^{\mu}{:=}\, M]\!] &= \mathit{STORE}(x,\, M,\, \mu) \\
[\![r{:=}\, M]\!] &= \mathit{LET}(r,\, M) \\
[\![\mathtt{fork}\ G]\!] &= \mathit{FORK}[\![G]\!] \\
[\![C; D]\!] &= [\![C]\!]\,;\, [\![D]\!] \\
[\![\mathtt{if}\ (M)\ \{C\}\ \mathtt{else}\ \{D\}]\!] &= \mathit{IF}(M{\neq}0,\, [\![C]\!],\, [\![D]\!]) \\
[\![\mathtt{0}]\!] &= \mathit{STOP}
\end{aligned}
$$

$$[\![\texttt{thread}\ C]\!] = \mathit{THREAD}[\![C]\!]$$
$$[\![G \parallel H]\!] = [\![G]\!] \parallel [\![H]\!]$$

[Examples.]

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

Definition 13. P is *completed* if $\tau^E(\mathsf{Q})$ implies Q.

1.5. Fork-Join

[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 14. A *pomset with preconditions and termination* is a pomset with preconditions together with a predicate $\checkmark$.

Definition 15.
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 7,
9) $\checkmark$ implies $\checkmark_1 \wedge \checkmark_2$.

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

1–3) as for *THREAD* in Definition 12,
4) if $\checkmark$ then $\tau^E(\mathsf{Q})$ implies Q.

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

1–4) as for *FORK* in Definition 12,
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 $a \leftrightarrow \lambda_2(e)$.

2. Complications

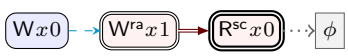
[I have a note: TC1: Track local state ???]

2.1. Release, Acquire, and SC Access

Write Q as Q_{sc} and introduce Q_{ra}.
Q_{sc} implies Q_{ra}.

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 §2.2). Using independency, one would also need another way to define completed pomsets. Finally, this use of independency is incompatible with fork (see §2.5).

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



2.2. ARM Compilation: Internal Acquires

Downgrading acquires/Anton example: $\downarrow^x$
We write $[\phi/\downarrow^*]$ for the substitution that performs $[\phi/\downarrow^x]$ for every x.

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.

2.3. ARM Compilation: Read-read dependencies

Control dependencies into reads as in MP with release on right and control dependency on left.
RW implies $\neg$RO and RO implies $\neg$RW.

2.4. Putting it together

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

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

$\mathsf{qs}_{\mathsf{rlx}} = \mathsf{tt}$ and otherwise $\mathsf{qs}_\mu = \mathsf{Q}_\mu$.
$\mathsf{ql}_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{sc}}$ and otherwise $\mathsf{ql}_\mu = \mathsf{tt}$.
$\mathsf{ds}^x_{\mathsf{rlx}}\phi = \phi[\mathsf{tt}/\downarrow^x]$ and otherwise $\mathsf{ds}^x_\mu\phi = \phi[\mathsf{ff}/\downarrow^*]$.
$\mathsf{dl}^x_{\mathsf{rlx}} = \mathsf{tt}$ and otherwise $\mathsf{dl}^x_\mu = \downarrow^x$.

Definition 16.
$\mathsf{qs}_{\mathsf{rlx}} = \mathsf{tt}$ and otherwise $\mathsf{qs}_\mu = \mathsf{Q}_\mu$.
$\mathsf{ql}_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{sc}}$ and otherwise $\mathsf{ql}_\mu = \mathsf{tt}$.

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

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

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

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

2.5. Coherence

Q_{sc} implies Q_{ra} implies $\mathsf{Q}^x_{\mathsf{rlx}}$ implies $\mathsf{Q}^x_{\mathsf{w}}$

- Coherence respects program order: $\mathsf{Q}^x_{\mathsf{rlx}}$
- Drop read-read coherence: $\mathsf{Q}^x_{\mathsf{w}}$ (Required for CSE without alias analysis over read only code, not required by hardware)

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.

Combining the features defined thus far, we have the following, assuming that each register occurs at most once.

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

$\mathsf{qs}^x_{\mathsf{rlx}} = \mathsf{Q}^x_{\mathsf{rlx}}$ and otherwise $\mathsf{qs}^x_\mu = \mathsf{Q}_\mu$.
$\mathsf{ql}^x_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{sc}}$ and otherwise $\mathsf{ql}^x_\mu = \mathsf{Q}^x_{\mathsf{w}}$.
$\mathsf{ds}^x_{\mathsf{rlx}}\phi = \phi[\mathsf{tt}/{\downarrow^x}]$ and otherwise $\mathsf{ds}^x_\mu\phi = \phi[\mathsf{ff}/{\downarrow^*}]$.
$\mathsf{dl}^x_{\mathsf{rlx}} = \mathsf{tt}$ and otherwise $\mathsf{dl}^x_\mu = {\downarrow^x}$.

Definition 17.
If $P \in \mathit{STORE}(x, M, \mu)$ then

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

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

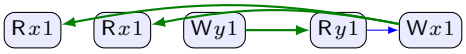
L1–L2) as before,
L3) $\kappa(e)$ implies $\mathsf{RO} \wedge \mathsf{ql}^x_\mu$,
L4) $\tau^D(\phi)$ implies $(v{=}r) \Rightarrow \phi[r/x]$
L5) $\tau^\emptyset(\phi)$ implies $\mathsf{dl}^x_\mu \wedge \neg\mathsf{Q}^x_{\mathsf{rlx}} \wedge (\mathsf{RW} \Rightarrow (v{=}r \vee x{=}r) \Rightarrow \phi[r/x])$.

3. Further Complications

3.1. Redundant Read Elimination

Requires indexing to resolve nondeterminism.

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



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

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

Combining the transformers, we have

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

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 oopsla, where we have:

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

Combining the transformers, we have

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

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:

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

3.2. If Closure

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

3.3. Address Calculation

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

In *STORE*:

S1) $\lambda(e) = (\mathsf{W}\,[\ell]\,v)$,
1) $\kappa(e)$ implies $(L{=}\ell \wedge M{=}v)$,
2) $\tau^\emptyset(\phi)$ implies $(L{=}\ell) \Rightarrow \phi[M/[\ell]]$,
3) $\tau^D(\phi)$ implies $(L{=}\ell) \Rightarrow (M{=}v) \wedge \phi[M/[\ell]]$,

In *LOAD*:

1) $\lambda(e) = (\mathsf{R}\,[\ell]\,v)$,
2) $\kappa(e)$ implies $(L{=}\ell)$,
3) $\tau^\emptyset(\phi)$ implies $(L{=}\ell) \Rightarrow (r{=}v \vee r{=}[\ell]) \Rightarrow \phi[r/[\ell]]$,
4) $\tau^D(\phi)$ implies $(L{=}\ell) \Rightarrow (r{=}v) \Rightarrow \phi[r/[\ell]]$,

3.4. Putting it together

The full semantics of load and store is given in Figure 1. Recall that $\mathcal{S}_D = \{s_d \mid d \in D\}$.

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

S1) if $\psi_d \wedge \psi_e$ is satisfiable then $d = e$,
S2) $\lambda(e) = (\mathsf{W}\,[\ell_e]\,v_e)$,
S3) $\kappa(e)$ implies $\psi_e \wedge L{=}\ell_e \wedge M{=}v_e \wedge \mathsf{RW} \wedge \mathsf{qs}_{\mu}^{[\ell_e]}$,
S4) $(\forall k)$ if $d \in D$ then $\tau^D(\phi)$ implies $\psi_d \Rightarrow (L{=}k) \Rightarrow \big((\mathsf{Q}_{\mathsf{w}}^{[k]} \Rightarrow M{=}v_d) \wedge \mathsf{ds}_{\mu}^{[k]}\phi[M/\lceil k\rceil]\big)$,
S5) $(\forall k)\ \tau^D(\phi)$ implies $(\nexists d \in D.\ \psi_d) \Rightarrow (L{=}k) \Rightarrow (\neg\mathsf{Q}_{\mathsf{w}}^{[k]} \wedge \mathsf{ds}_{\mu}^{[k]}\phi[M/\lceil k\rceil])$.

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

L1) if $\psi_d \wedge \psi_e$ is satisfiable then $d = e$,
L2) $\lambda(e) = (\mathsf{R}\,[\ell_e]\,v_e)$,
L3) $\kappa(e)$ implies $\psi_e \wedge L{=}\ell_e \wedge \mathsf{RO} \wedge \mathsf{ql}_{\mu}^{[\ell_e]}$,
L4) $(\forall k)$ if $d \in D$ then $\tau^D(\phi)$ implies $\psi_d \Rightarrow (L{=}k) \Rightarrow (v{=}s_d) \Rightarrow \phi[s_d/r][s_d/\lceil k\rceil]$,
L5) $(\forall k)$ if $d \notin D$ then $\tau^D(\phi)$ implies $\psi_d \Rightarrow (L{=}k) \Rightarrow \big(\mathsf{dl}_{\mu}^{[k]} \wedge \neg\mathsf{Q}_{\mathsf{rlx}}^{[k]} \wedge (\mathsf{RW} \Rightarrow (v{=}s_d \vee x{=}s_d) \Rightarrow \phi[s_d/r][s_d/\lceil k\rceil])\big)$,
L6) $(\forall k)(\forall s)\ \tau^D(\phi)$ implies $(\nexists d \in D.\ \psi_d) \Rightarrow (L{=}k) \Rightarrow (\mathsf{dl}_{\mu}^{[k]} \wedge \neg\mathsf{Q}_{\mathsf{rlx}}^{[k]} \wedge \Rightarrow \phi[s/r][s/\lceil k\rceil])$.

Figure 1. Full Semantics of Load and Store

Appendix A.
Differences from the OOPSLA Model

A.1. Must Allow Inconsistent Preconditions

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

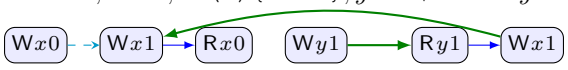
[The definition does not give a sensible notion of completed execution without consistency and causal strengthening.]

A.2. 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.]

Appendix B.
Errors in the OOPSLA Model

This paper addresses several errors in [**?**], which we henceforth refer to as [JJR].

B.1. Parallel Composition

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

B.2. Redundant Read Elimination

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

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

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 $\kappa(e)[v/x]$.

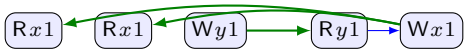
(P5A) If d reads and e writes then either $\kappa'(e)$ implies $\kappa(e)$ or $d \leq' e$.

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 [JJR]. 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 [JJR]: $(\mathsf{W}y1)$ 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]

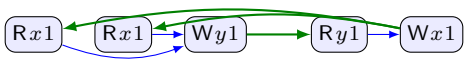
The substitution $[x/r]$ leaves the obligation on x to be fulfilled by the preceding write. Thus, the read does not update the *value* of x in subsequent predicates. The substitution $[r/x]$, instead, does update the value of x, thus removing any obligation on x for preceding code.

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

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



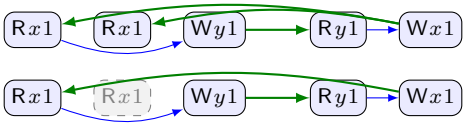
entail $r{=}1 \wedge r{=}s$ in $[\![s{:=}x; \texttt{if}\ (r{=}s)\ \{y{:=}1\}]\!]$. P4C imposes $r{=}1$, and P5A imposes $r{=}x$. Adding the second read, the precondition of $(\mathsf{W}y1)$ must entail both $1{=}1 \wedge 1{=}x$ and also $x{=}1 \wedge x{=}x$. This can be simplified to $x{=}1$. This leaves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

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

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

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:

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

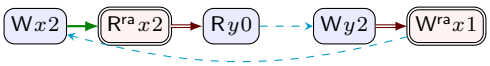


But this is not a problem, since fulfillment requires that $(\mathsf{W}x1)$ precede both reads of x.

B.3. Internal Acquiring Reads

The second issue concerns acquiring reads. Shortly after publication, Podkopaev [2020] noticed a shortcoming of the implementation on ARM8 in [JJR, §7]. The proof given there assumes that all internal reads can be dropped. However, this is not the case for acquiring reds. For example, [JJR] disallows the following execution, which is allowed by ARM8 and TSO.

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

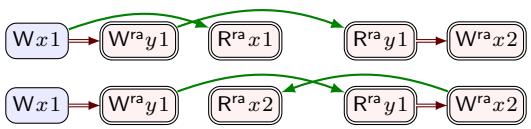


The solution we have adopted is to allow an acquiring read to be downgraded to a relaxed read when it is preceded (sequentially) by a relaxed write that could fulfill it. Backporting this solution to [JJR] requires that we add access predicates to the logic and allow

B.4. Triangular Races

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

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



Bug is in [Dongol et al., 2019, 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 [Alglave, 2020; Arm Limited, 2020; Alglave et al., 2020].

References

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.

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

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

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.

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.

A. Podkopaev. Private correspondence, Nov. 2020.