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,
- 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.

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

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.

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 *open*. Occurrences of register names and memory locations are subject to substitutions of the form $\phi[M/r]$ and $\phi[M/x]$. Actions are not subject to substitution.

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

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* $(\mathsf{R}xw)$.

We say that 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 $e \leq c$.

We say that P is *fulfilled* if every read in P is fulfilled.

We define *independency* $(\leftrightarrow\, \subseteq \mathcal{A} \times \mathcal{A})$ as follows.

$$\begin{aligned}
\leftrightarrow\, = &\{(\mathsf{R}xv, \mathsf{R}yw)\} \\
&\cup \{(\mathsf{W}xv, \mathsf{W}yw) \mid x \neq y \vee v = w\} \\
&\cup \{(\mathsf{R}xv, \mathsf{W}yw), (\mathsf{W}xv, \mathsf{R}yw) \mid x \neq y\}
\end{aligned}$$

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

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

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

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

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]

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.
If $P \in \mathit{STOP}$ then $E = \emptyset$.
If $P \in (\mathcal{P}_1 \mid\mid \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)$ $(\exists P_2 \in \mathcal{P}_2)$

1–6) as for $\mid\mid$ 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 $\mid\mid$ 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 $(\mathsf{W}xv) \to P_2$ in Definition 3,
7) if $d \in E_1 \setminus E_2$ then $\kappa(d)$ implies $(M{=}v)$,
8) if $e \in E_2$ then either $\kappa(e)$ implies $\kappa_2(e)$ or
$e \in E_1$ and $\kappa(e)$ implies $(M{=}v) \vee \kappa_2(e)$.

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

1–6) as for $(\mathsf{R}xv) \to P_2$ in Definition 3,
7) if $e \in E_2$ then either $e \in E_1$ or
$\kappa(e)$ 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 $d \in E_1$.

Note that when $d \in E_1 \setminus E_2$, *LOADPRE* does not constrain $\kappa(d)$.

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

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

This is essentially the model of Jagadeesan et al. [2020], restricted to relaxed access. There are only two substantial differences. First, for simplicity, we enforce read-read dependencies. Second, 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, the second difference can be seen. For example, the following execution is allowed with $[x/r]$, but not $[r/x]$.

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

$$\begin{tikzcd}
\mathsf{W}x0 \arrow[r, dashed] & \mathsf{W}x1 \arrow[r] & \mathsf{R}x0 & \mathsf{W}y1 \arrow[r] & \mathsf{R}y1 \arrow[r] & \mathsf{W}x1 \arrow[llll, bend right]
\end{tikzcd}$$

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

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

Definition 8. A *family of predicate transformers* indexed by subsets of E consists of predicate transformers τ^D for each set of events D, such that if $C \subseteq D$ then $\tau^C(\phi)$ implies $\tau^D(\phi)$.

Definition 9. A pomset with predicate tansformers is a pomset with preconditions, together with a family of predicate transformers τ indexed by subsets of E.

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 10. 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) $\tau^D(\phi)$ implies ϕ.

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

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

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 6,
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 $P \in \mathit{STORE}(x,\, M)$ then $(\exists v \in \mathcal{V})$

1) $\lambda(e) = (\mathsf{W}xv)$,
2) $\kappa(e)$ implies $(M{=}v)$,
3) $\tau^\emptyset(\phi)$ implies $\phi[M/x]$,
4) $\tau^D(\phi)$ implies $(M{=}v) \wedge \phi[M/x]$, if $D \neq \emptyset$,
5) if $d, e \in E$ then $d = e$.

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

1) $\lambda(e) = (\mathsf{R}xv)$,
2) $\tau^\emptyset(\phi)$ implies $(r{=}v \vee r{=}x) \Rightarrow \phi[r/x]$,
3) $\tau^D(\phi)$ implies $(r{=}v) \Rightarrow \phi[r/x]$, if $D \neq \emptyset$,
4) if $d, e \in E$ then $d = e$.

The complete semantics is as follows.

$$
\begin{aligned}
[\![\texttt{skip}]\!] &= \mathit{SKIP} \\
[\![r{:=}\,x]\!] &= \mathit{LOAD}(x,\, r) \\
[\![x{:=}\,M]\!] &= \mathit{STORE}(x,\, M) \\
[\![r{:=}\,M]\!] &= \mathit{LET}(r,\, M) \\
[\![\texttt{fork}\, G]\!] &= \mathit{FORK}[\![G]\!] \\
[\![C; D]\!] &= [\![C]\!]\, ;[\![D]\!] \\
[\![\texttt{if}(\psi)\{C\}\,\texttt{else}\,\{D\}]\!] &= \mathit{IF}(\psi,\, [\![C]\!],\, [\![D]\!]) \\
[\![0]\!] &= \mathit{STOP} \\
[\![\texttt{thread}\, C]\!] &= \mathit{THREAD}[\![C]\!] \\
[\![G \parallel H]\!] &= [\![G]\!] \parallel [\![H]\!]
\end{aligned}
$$

[Examples.]

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

2. Complications

[Very drafty. These are just notes.]

- Dependency: preconditions
- TC1: Track local state

2.1. Address Calculation

Definition 12. If $P \in \mathit{STORE}(x,\, M)$ then $(\exists v \in \mathcal{V})$

1) $\lambda(e) = (\mathsf{W}[\ell]v)$,
2) $\kappa(e)$ implies $(M{=}v)$,
3) $\tau^\emptyset(\phi)$ implies $\phi[M/x]$,
4) $\tau^D(\phi)$ implies $(M{=}v) \wedge \phi[M/x]$, if $D \neq \emptyset$,
5) if $d, e \in E$ then $d = e$.

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

1) $\lambda(e) = (\mathsf{R}xv)$,
2) $\tau^\emptyset(\phi)$ implies $(r{=}v \vee r{=}x) \Rightarrow \phi[r/x]$,
3) $\tau^D(\phi)$ implies $(r{=}v) \Rightarrow \phi[r/x]$, if $D \neq \emptyset$,
4) if $d, e \in E$ then $d = e$.

2.2. Coherence

Can be encoded in independency, or logic.

- Coherence respects program order: Q_x
- Drop read-read coherence: QW_x (Required for CSE without alias analysis over read only code, not required by hardware)

2.3. Release Acquire

Q

Can be encoded in independency, or logic, but logic is more flexible, and we need that for ARM8.

2.4. ARM Compilation: Internal Acquires

Downgrading acquires/Anton example: $\downarrow x$

2.5. ARM Compilation: Read-read dependencies

RW/RO (control dependencies into reads as in MP with release on right and control dependency on left)

2.6. Redundant Read Elimination

Requires indexing to resolve nondeterminism.

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

$$\begin{tikzcd}
\mathsf{R}x1 & \mathsf{R}x1 & \mathsf{W}y1 \arrow[r] & \mathsf{R}y1 \arrow[r] & \mathsf{W}x1 \arrow[llll, bend right] \arrow[lll, bend right]
\end{tikzcd}$$

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

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

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:

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

$$\begin{tikzcd} \mathsf{R}x1 & \mathsf{W}y1 \arrow[r] & \mathsf{R}y1 \arrow[r] & \mathsf{W}x1 \arrow[lll, bend right] \end{tikzcd}$$

2.7. If Closure

Requires indexing to resolve nondeterminism.

IF closure/case analysis: ψ_e

2.8. Agda

```
κLOAD : Address → Value → Formula
κLOAD a v = RO ∧ Qw[ a ]

τLOADD : Register → Register → Address → Value → Formula → Formula
τLOADD r s a v ϕ = (value v == register s) ⇒ (ϕ [ register s / r ] [ register s /[ a ]])

τLOADI : Register → Register → Address → AccessMode → Formula → Formula
τLOADI r s a rlx ϕ =          ¬ Q[ a ] ∧ (RW ⇒ ([ a ]== register s) ⇒ (ϕ [ register s / r ] [ register s /[ a ]]))
τLOADI r s a ra  ϕ = ↓[ a ] ∧ ¬ Q[ a ] ∧ (RW ⇒ ([ a ]== register s) ⇒ (ϕ [ register s / r ] [ register s /[ a ]]))

τLOAD∅ : Register → Register → Address → AccessMode → Formula → Formula
τLOAD∅ r s a rlx ϕ =          ¬ Q[ a ] ∧ (ϕ [ register s / r ] [ register s /[ a ]])
τLOAD∅ r s a ra  ϕ = ↓[ a ] ∧ ¬ Q[ a ] ∧ (ϕ [ register s / r ] [ register s /[ a ]])

record LOAD (r : Register) (L : Expression) (μ : AccessMode) (P : PomsetWithPredicateTransformers) : Set₁ where

  open PomsetWithPredicateTransformers P

  field a : Event → Address
  field v : Event → Value
  field ψ : Event → Formula

  field d=e : ∀ d e → (d ∈ E) → (e ∈ E) → ((ψ(d) ∧ ψ(e)) ∈ Satisfiable) → (d ≡ e)
  field ℓ=Rav : ∀ e → (e ∈ E) → ℓ(e) ≡ (R (a(e)) (v(e)))
  field κ⊨κLOAD :  ∀ e → (e ∈ E) → κ(e) ⊨ (ψ(e) ∧ (L == address (a(e))) ∧ κLOAD (a(e)) (v(e)))
  field τC⊨τLOADD : ∀ C ϕ e → (e ∈ E) → (e ∈ C) → (τ(C)(ϕ) ⊨ (ψ(e) ⇒ τLOADD r (r[ e ]) (a(e)) (v(e)) ϕ))
  field τC⊨τLOADI : ∀ C ϕ a e → (e ∈ E) → (e ∉ C) → (τ(C)(ϕ) ⊨ (ψ(e) ⇒ (L == address a) ⇒ τLOADI r (r[ e ]) a μ ϕ))
  field τC⊨τLOAD∅ : ∀ C ϕ a s χ → (∀ e → (e ∈ E) → (e ∈ C) → (χ ⊨ ¬(ψ(e)))) → (τ(C)(ϕ) ⊨ (χ ⇒ (L == address a) ⇒ τLOAD∅ r s a μ ϕ))

κSTORE : AccessMode → Expression → Address → Value → Formula
κSTORE rlx M a v = (M == value v) ∧ RW ∧ Q[ a ]
κSTORE ra  M a v = (M == value v) ∧ RW ∧ Q

τSTORED : AccessMode → Expression → Address → Value → Formula → Formula
τSTORED rlx M a v ϕ = (Qw[ a ] ⇒ (M == value v)) ∧ (ϕ [ M /[ a ]] [ tt /↓[ a ]])
τSTORED ra  M a v ϕ = (Qw[ a ] ⇒ (M == value v)) ∧ (ϕ [ M /[ a ]] [ ff /↓[*]])

τSTOREI : AccessMode → Expression → Address → Formula → Formula
τSTOREI rlx M a ϕ = (¬ Qw[ a ]) ∧ (ϕ [ M /[ a ]] [ tt /↓[ a ]])
τSTOREI ra  M a ϕ = (¬ Qw[ a ]) ∧ (ϕ [ M /[ a ]] [ ff /↓[*]])

record STORE (L : Expression) (μ : AccessMode) (M : Expression) (P : PomsetWithPredicateTransformers) : Set₁ where

  open PomsetWithPredicateTransformers P

  field a : Event → Address
  field v : Event → Value
  field ψ : Event → Formula

  field d=e : ∀ d e → (d ∈ E) → (e ∈ E) → ((ψ(d) ∧ ψ(e)) ∈ Satisfiable) → (d ≡ e)
  field ℓ=Wav : ∀ e → (e ∈ E) → ℓ(e) ≡ (W (a(e)) (v(e)))
  field κ⊨κSTORE :  ∀ e → (e ∈ E) → (κ(e) ⊨ (ψ(e) ∧ (L == address (a(e))) ∧ κSTORE μ M (a(e)) (v(e))))
  field τC⊨τSTORED : ∀ C ϕ a e → (e ∈ E) → (e ∈ C) → (τ(C)(ϕ) ⊨ (ψ(e) ⇒ (L == address a) ⇒ τSTORED μ M a (v(e)) ϕ))
  field τC⊨τSTOREI : ∀ C ϕ a χ → (∀ e → (e ∈ E) → (e ∈ C) → (χ ⊨ ¬(ψ(e)))) → (τ(C)(ϕ) ⊨ (χ ⇒ (L == address a) ⇒ τSTOREI μ M a ϕ))
```

Appendix

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

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 $\vee$.

2. Redundant Read Elimination

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

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

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

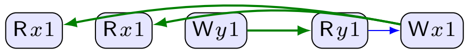
(P5A) If d reads and e writes then either $\kappa'(e)$ implies $\kappa(e)$ or $d \leq' e$.

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.

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

We claimed that TC2 allowed the following execution:



But this execution is not possible using the semantics of [JJR]: (Wy1) has precondition $r{=}s$ in $[\![\mathtt{if}(r{=}s)\{y{:=}\,1\}]\!]$. Given the lack of order in the execution, the precondition of (Wy1) must entail $r{=}1 \wedge r{=}x$ in $[\![s{:=}\,x; \mathtt{if}(r{=}s)\{y{:=}\,1\}]\!]$. P4C imposes $r{=}1$, and P5A imposes $r{=}x$. Adding the second read, the precondition of (Wy1) 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 $\kappa(e)[v/r]$.

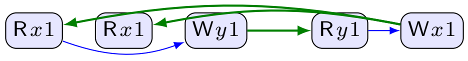
We can then reason with TC2 as follows: (Wy1) has precondition $r{=}s$ in $[\![\mathtt{if}(r{=}s)\{y{:=}\,1\}]\!]$. To avoid introducing order in the execution, the precondition of (Wy1) must entail $r{=}1 \wedge r{=}s$ in $[\![s{:=}\,x; \mathtt{if}(r{=}s)\{y{:=}\,1\}]\!]$. P4C imposes $r{=}1$, and P5A imposes $r{=}x$. Adding the second read, the precondition of (Wy1) 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.

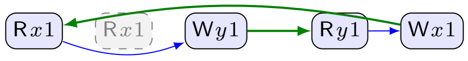
With read elimination, the rule for relaxed reads is as follows:

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

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:

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



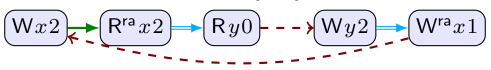


But this is not a problem, since fulfillment requires that (Wx1) precede both reads of x.

3. Internal Acquiring Reads

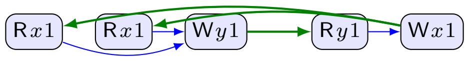
The second issue concerns acquiring reads. Shortly after publication, Podkopaev [2020] noticed a shortcoming of the implementation on ARM8 in [JJR, §7]. The proof given there assumes that all internal reads can be dropped. However, this is not the case for acquiring reds. For example, [JJR] disallows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

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



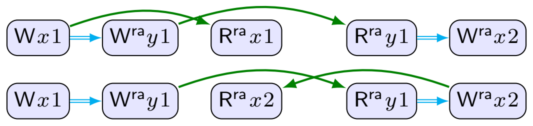
The solution we have adopted is to allow an acquiring read to be downgraded to a relaxed read when it is preceded (sequentially) by a relaxed write that could fulfill it. Backporting this solution to [JJR] requires that we add access predicates to the logic and allow

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



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.