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,
- 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, 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 $\phi \vDash \mathsf{ff}$.

[We assume $(s : E \to \mathcal{R})$]

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.

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

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 $d \leq e$ or $a \leftrightarrow \lambda_2(e)$.

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

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

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], restricted to relaxed access. We discuss differences in the appendix.

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

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 $P \in (\mathcal{P}_1 \parallel \mathcal{P}_2)$ 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 $(\mathsf{W}xv) \rightarrow P_2$ 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 $P \in \mathit{LOADPRE}(x, r, \mathcal{P}_2)$ then $(\exists P_2 \in \mathcal{P}_2)$ $(\exists v \in \mathcal{V})$

1–6) as for $(\mathsf{R}xv) \rightarrow P_2$ 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 $e \in E_1$.

[Define substitution.]

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

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

[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 7. A *predicate transformer* is a monotone function $\tau : \Phi \rightarrow \Phi$ 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 8. 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 9. A pomset with predicate tansformers is a pomset with preconditions, together with a family of predicate transformers for 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 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 6,
9) $\tau^D(\phi)$ implies $(\psi \wedge \tau_1^D(e)) \vee (\neg\psi \wedge \tau_2^D(\phi))$.

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

1) if $\psi_d \wedge \psi_e$ is satisfiable then $d = e$,
2) $\lambda(e) = (\mathsf{W}\,[\ell_e]\,v_e)$,
3) $\kappa(e)$ implies $\psi_e \wedge L{=}\ell_e \wedge M{=}v_e \wedge \mathsf{RW} \wedge \mathsf{Q}^\mu$, where $\mathsf{Q}^{\mathsf{rlx}} = \mathsf{Q}_{[\ell_e]}$ and otherwise $\mathsf{Q}^\mu = \mathsf{Q}_\mu$,
4) $(\forall k)$ if $d \in D$ then $\tau^D(\phi)$ implies $\psi_d \Rightarrow (L{=}k) \Rightarrow \big((\mathsf{QW}_{[k]} \Rightarrow M{=}v_d) \wedge \phi{\downarrow^\mu}[M/[k]]\big)$,
5) $(\forall k)$ $\tau^D(\phi)$ implies $(\nexists d \in D.\ \psi_d) \Rightarrow (L{=}k) \Rightarrow (\neg \mathsf{QW}_{[k]} \wedge \phi{\downarrow^\mu}[M/[k]])$
where $\phi{\downarrow^{\mathsf{rlx}}} = \phi[\mathsf{tt}/{\downarrow_{[k]}}]$ and otherwise $\phi{\downarrow^\mu} = \phi[\mathsf{ff}/{\downarrow_{[\star]}}]$.

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

1) if $\psi_d \wedge \psi_e$ is satisfiable then $d = e$,
2) $\lambda(e) = (\mathsf{R}\,[\ell_e]\,v_e)$,
3) $\kappa(e)$ implies $\psi_e \wedge L{=}\ell_e \wedge \mathsf{RO} \wedge \mathsf{Q}^\mu$, where $\mathsf{Q}^{\mathsf{sc}} = \mathsf{Q}_{\mathsf{sc}}$ and otherwise $\mathsf{Q}^\mu = \mathsf{QW}_{[\ell_e]}$,
4) $(\forall k)$ if $d \in D$ then $\tau^D(\phi)$ implies $\psi_d \Rightarrow (L{=}k) \Rightarrow (v{=}s_d) \Rightarrow \phi[s_d/r][s_d/[k]]$
5) $(\forall k)$ if $d \notin D$ then $\tau^D(\phi)$ implies $\psi_d \Rightarrow (L{=}k) \Rightarrow \big({\downarrow^\mu} \wedge \neg \mathsf{Q}_{[k]} \wedge (\mathsf{RW} \Rightarrow (v{=}s_d \vee x{=}s_d) \Rightarrow \phi[s_d/r][s_d/[k]])\big)$
6) $(\forall k)(\forall s)$ $\tau^D(\phi)$ implies $(\nexists d \in D.\ \psi_d) \Rightarrow (L{=}k) \Rightarrow ({\downarrow^\mu} \wedge \neg \mathsf{Q}_{[k]} \wedge \Rightarrow \phi[s/r][s/[k]])$
where ${\downarrow^{\mathsf{rlx}}} = \mathsf{tt}$ and otherwise ${\downarrow^\mu} = {\downarrow_k}$

Figure 1. Full Semantics of Load and Store

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

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

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

[Note that we could change the premise of $\tau^\emptyset$ in *LOAD* from $(v{=}r \vee x{=}r)$ to $(x{=}r)$. The requirements of a family of predicate transforms effectively adds the additional requirement.]

[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.]

The complete semantics is as follows.

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

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

[Examples.]

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

2. Complications

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

2.1. Release Acquire

Can be encoded in independency, or logic, but logic is compatible with fork.

Logic is also more flexible, and we need that for ARM8.

We use Q.

Definition 12. P is *completed* if $\tau^E(\mathsf{Q})$ implies Q.

2.2. Coherence

Q_{sc} implies Q_{ra} implies Q_x implies QW_x

Can be encoded in independency, or logic.

If you put in independency then you add this to *STORE*:

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

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

Instead we put it in the logic, using

- 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. ARM Compilation: Internal Acquires

Downgrading acquires/Anton example: ${\downarrow_x}$

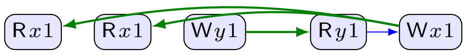
2.4. ARM Compilation: Read-read dependencies

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

2.5. Redundant Read Elimination

Requires indexing to resolve nondeterminism.

$$r{:=}\,x;\, s{:=}\,x;\, \texttt{if}\,(r{=}s)\,\{y{:=}\,1\} \parallel x{:=}\,y \qquad (\text{TC}2)$$



Precondition of (Wy1) 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

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

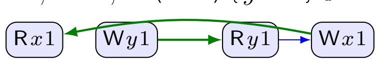
Applying this to $(r{=}s)$, we have

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

which is not a tautology.

The semantics here allows this by coalescing:

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



2.6. If Closure

Requires indexing to resolve nondeterminism.

IF closure/case analysis: ψ_e

2.7. Address Calculation

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

Definition 13. If $P \in \textit{STORE}(L,\, M)$ then $(\exists v,\, \ell \in \mathcal{V})$

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

If $P \in \textit{LOAD}(L,\, r)$ then $(\exists v,\, \ell \in \mathcal{V})$

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 $(L{=}\ell) \Rightarrow (r{=}v) \Rightarrow \phi[r/[\ell]]$,
5) if $d, e \in E$ then $d = e$.

References

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.