Sequential Composition for Relaxed Memory: Pomsets with Predicate Transformers*

ALAN JEFFREY, Roblox, USA
JAMES RIELY, DePaul University, USA

This paper presents the first compositional definition of sequential composition that applies to a relaxed memory model weak enough to allow efficient implementation on Arm. We extend the denotational model of pomsets with preconditions with predicate transformers. Previous work has shown that pomsets with preconditions are a model of concurrent composition, and that predicate transformers are a model of sequential composition. This paper show how they can be combined.

CCS Concepts: • **Theory of computation → Parallel computing models**; *Preconditions*.

Additional Key Words and Phrases: Concurrency, Relaxed Memory Models, Multi-Copy Atomicity, ARMv8, Pomsets, Preconditions, Temporal Safety Properties, Thin-Air Reads, Compiler Optimizations

ACM Reference Format:
Alan Jeffrey and James Riely. 2020. Sequential Composition for Relaxed Memory: Pomsets with Predicate Transformers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 194 (November 2020), 12 pages. https://doi.org/10.1145/3428262

1 MODEL

In this section, we present the mathematical preliminaries for the model (which can be skipped on first reading). We then present the model incrementally, starting with a model built using *partially ordered multisets* (*pomsets*) [**??**], and then adding preconditions and finally predicate transformers.

In later sections, we will discuss extensions to the logic, and to the semantics of load, store and thread initialization, in order to model relaxed memory more faithfully. We stress that these features do *not* change any of the structures of the language: conditionals, parallel composition, and sequential composition are as defined in this section.

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

We require that

- values and registers are disjoint,

*Riely was supported by the National Science Foundation under grant No. CCR-1617175.

Authors' addresses: Alan Jeffrey, Roblox, Chicago, USA; James Riely, DePaul University, Chicago, USA.

2475-1421/2020/11-ART194
https://doi.org/10.1145/3428262

- values include at least the constants 0 and 1,
- expressions include at least registers and values,
- expressions do *not* include references: $M[N/x] = M$.

We model the following language.

$$\mu \ ::= \ \mathsf{rlx} \mid \mathsf{ra} \mid \mathsf{sc}$$

$$S \ ::= \ \mathtt{skip} \mid r := M \mid r := [L]^{\mu} \mid [L]^{\mu} := M \mid \mathsf{F}^{\nu} \mid \mathtt{if}(M)\{S_1\}\,\mathtt{else}\,\{S_2\} \mid S_1 ; S_2 \mid S_1 \ {}_{\gamma}\!\Vdash\!\!\!\rightarrow \ S_2$$
$$\mid r := \mathsf{CAS}([L]^{\mu_1,\mu_2}, M, N) \mid r := \mathsf{FADD}([L]^{\mu_1,\mu_2}, M) \mid r := \mathsf{EXCHG}([L]^{\mu_1,\mu_2}, M)$$

Memory modes, μ, are relaxed (rlx), release-acquire (ra), and sequentially consistent (sc). Relaxed mode is the default; we regularly elide it from examples. ra/sc accesses are collectively known as *synchronized accesses*.

Commands, aka *statements*, S, include memory accesses at a given mode, as well as the usual structural constructs. Following [**?**], $\Vdash\!\!\!\rightarrow$ denotes parallel composition, preserving thread state on the right after a join. In examples and sublanguages without join, we use the symmetric $\parallel$ operator.

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 ($x{=}M$),
- formulae are closed under negation, conjunction, disjunction, and substitutions $[M/r]$, $[M/x]$,
- there is an entailment relation $\vDash$ between formulae,
- $\vDash$ has the expected semantics for $=$, $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and substitution.

Logical formulae include equations over registers, such as ($r{=}s{+}1$). For use in §**??**, we also include equations over memory references, such as ($x{=}1$). Formulae are subject to substitutions; actions are not. We use expressions as formulae, coercing M to $M{\neq}0$. Equations have precedence over logical operators; thus $r{=}v \Rightarrow s{>}w$ is read $(r{=}v) \Rightarrow (s{>}w)$. As usual, implication associates to the right; thus $\phi \Rightarrow \psi \Rightarrow \theta$ is read $\phi \Rightarrow (\psi \Rightarrow \theta)$.

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

Throughout §1–**??** we additionally require that

- each register is assigned at most once in a program.

In §**??**, we drop this restriction, requiring instead that

- there are registers $\mathcal{S}_{\mathcal{E}} = \{s_e \mid e \in \mathcal{E}\}$,
- registers $\mathcal{S}_{\mathcal{E}}$ do not appear in programs: $S[N/s_e] = S$.

Definition 1.1. Reorderability relations. Coherence-after and synchronization.

$$\ltimes_{\mathsf{ca}} = \{(\mathsf{W}x, \mathsf{W}y) \mid x \neq y\} \cup \{(\mathsf{R}x, \mathsf{W}y) \mid x \neq y\} \cup \{(\mathsf{W}x, \mathsf{R}y)\} \ \cup \ \{(\mathsf{R}x, \mathsf{R}y)\}$$

$$\ltimes_{\mathsf{sync}} = \{(\mathsf{W}^{\mu\neq\mathsf{sc}}, \mathsf{R}^{\nu\neq\mathsf{sc}})\} \cup \{(\mathsf{W}^{\mu}, \mathsf{W}^{\mathsf{rlx}})\} \cup \{(\mathsf{F}^{\mathsf{rel}}, \mathsf{R}^{\nu})\}$$
$$\cup \{(\mathsf{R}^{\mathsf{rlx}}, \mathsf{W}^{\mathsf{rlx}})\} \cup \{(\mathsf{R}^{\mathsf{rlx}}, \mathsf{R}^{\nu})\} \cup \{(\mathsf{W}^{\mu}, \mathsf{F}^{\mathsf{acq}})\} \cup \{(\mathsf{F}^{\mathsf{rel}}, \mathsf{F}^{\mathsf{acq}})\}$$

$$\ltimes \ = \ltimes_{\mathsf{sync}} \cap \ltimes_{\mathsf{ca}}$$

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

Actions ($\mathsf{W}^{\mu\neq\mathsf{rlx}}$) and ($\mathsf{F}^{\nu\neq\mathsf{acq}}$) are *release actions*.

1.2 Model

Definition 1.2. A *pomset with predicate transformers* over $\mathcal{A}$ is a tuple $(E, \lambda, \kappa, \tau, \checkmark, \mathsf{rf}, \leq)$ where

(1) $E \subset \mathcal{E}$ is a set of *events*,
(2) $\lambda : E \to \mathcal{A}$ defines a *label* for each event,
(3) $\kappa : E \to \Phi$ defines a *precondition* for each event,
(4) $\tau : 2^{\mathcal{E}} \to \Phi \to \Phi$ defines a *predicate transformer* for each set of events,
(5) $\checkmark : \Phi$ defines a *termination condition*,
(6) $\leq \subseteq (E \times E)$, is a partial order capturing *causality*,
(7) $\mathsf{rf} \subseteq (E \times E)$ is a relation capturing *reads-from* such that
(7a) if $(d, e) \in \mathsf{rf}$ and $(c, e) \in \mathsf{rf}$ then $d = c$,
(7b) if $(d, e) \in \mathsf{rf}$ then $\lambda(d)$ matches $\lambda(e)$,
(7c) if $(d, e) \in \mathsf{rf}$ and $\lambda(c)$ blocks $\lambda(e)$ then either $c \leq d$ or $e \leq c$.
(7d) if $(d, e) \in \mathsf{rf}$ then $d \leq e$.

Definition 1.3. A pomset is *top-level* if for every $e \in E$:

(1) $\kappa(e)$ is a tautology,
(2) if $\lambda(e) = (\mathsf{R})$ then $(\exists d)\ (d, e) \in \mathsf{rf}$.

Definition 1.4. If $P \in \mathcal{P}_1 \Vdash \mathcal{P}_2$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

(1) $E = (E_1 \cup E_2)$,
(2) $\lambda = (\lambda_1 \cup \lambda_2)$,
(3) $\leq \supseteq (\leq_1 \cup \leq_2)$,
(4) $(\mathsf{rf}_1 \cup \mathsf{rf}_2) \subseteq \mathsf{rf} \subseteq (\mathsf{rf}_1 \cup \mathsf{rf}_2 \cup \leq)$,
(5) E_1 and E_2 are disjoint,
(6) if $e \in E_1$ then $\kappa(e)$ implies $\kappa_1(e)$,
(7) if $e \in E_2$ then $\kappa(e)$ implies $\kappa_2(e)$,
(8) $\tau^D(\psi)$ implies $\tau_2^D(\psi)$,
(9) $\checkmark$ implies $\checkmark_1 \wedge \checkmark_2$.

If $P \in \mathcal{P}_1 ; \mathcal{P}_2$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$
(1–3) as for $\Vdash$,
(4) $(\mathsf{rf}_1 \cup \mathsf{rf}_2) \subseteq \mathsf{rf} \subseteq (\mathsf{rf}_1 \cup \mathsf{rf}_2 \cup (E_1 \times E_2))$,
(5) if $(d, e) \in \mathsf{rf}$ and $\lambda(c)$ blocks $\lambda(e)$ for some $c \in E_1$ and $e \in E_2$ then $c \leq d$,
(6) if $d \in E_1$ and $e \in E_2$ then either $d \leq e$ or $\lambda_1(d) \ltimes \lambda_2(e)$,
(7) if $e \in E_2$ and $\lambda(e)$ is a release then $\kappa(e)$ implies $\checkmark_1$,
(8) if $e \in E_1 \setminus E_2$ then $\kappa(e)$ implies $\kappa_1(e)$,
(9) if $e \in E_2 \setminus E_1$ then $\kappa(e)$ implies $\kappa_2'(e)$,
(10) if $e \in E_1 \cap E_2$ then $\kappa(e)$ implies $\kappa_1(e) \vee \kappa_2'(e)$, where $\kappa_2'(e) = \tau_1^{\downarrow e}(\kappa_2(e))$,
where $\downarrow e = \{c \mid c < e\}$ if $\lambda(e)$ is a write, and $\downarrow e = E_1$, otherwise,
(11) $\tau^D(\psi)$ implies $\tau_1^D(\tau_2^D(\psi))$,
(12) $\checkmark$ implies $\checkmark_1 \wedge \tau_1^{E_1}(\checkmark_2)$.

If $P \in \mathit{IF}(\phi, \mathcal{P}_1, \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$
(1–3) as for $\Vdash$,
(4) $\mathsf{rf} = (\mathsf{rf}_1 \cup \mathsf{rf}_2)$ and $(\mathsf{rf} \cap (E_1 \times E_1)) = \mathsf{rf}_1$ and $(\mathsf{rf} \cap (E_2 \times E_2)) = \mathsf{rf}_2$,
(5) if $e \in E_1 \setminus E_2$ then $\kappa(e)$ implies $\phi \wedge \kappa_1(e)$,
(6) if $e \in E_2 \setminus E_1$ then $\kappa(e)$ implies $\neg\phi \wedge \kappa_2(e)$,
(7) if $e \in E_1 \cap E_2$ then $\kappa(e)$ implies $(\phi \Rightarrow \kappa_1(e)) \wedge (\neg\phi \Rightarrow \kappa_2(e))$,
(8) $\tau^D(\psi)$ implies $(\phi \Rightarrow \tau_1^D(\psi)) \wedge (\neg\phi \Rightarrow \tau_2^D(\psi))$,
(9) $\checkmark$ implies $(\phi \Rightarrow \checkmark_1) \wedge (\neg\phi \Rightarrow \checkmark_2)$.

If $P \in \mathit{LET}(r, M)$ then $E = \emptyset$ and $\tau^D(\psi)$ implies $\psi[M/r]$.
If $P \in \mathit{SKIP}$ then $E = \emptyset$ and $\tau^D(\psi)$ implies ψ.
If $P \in \mathit{FENCE}(\mu)$ then

(1) $E \neq \emptyset$ and if $d, e \in E$ then $d = e$,
(2) $\lambda(e) = \mathsf{F}^{\mu}$,
(3) $\tau^{D}(\psi)$ implies ψ,
(4) if $E = \emptyset$ then $\checkmark$ implies ff.

If $P \in \mathit{STORE}(x, M, \mu)$ then $(\exists v \in \mathcal{V})$
(1) if $d, e \in E$ then $d = e$,
(2) $\lambda(e) = \mathsf{W}^{\mu} x v$,
(3) $\kappa(e)$ implies $M{=}v$,
(4) $\tau^{D}(\psi)$ implies ψ,
(5) if $E = \emptyset$ then $\checkmark$ implies ff,
(6) if $E \neq \emptyset$ then $\checkmark$ implies $M{=}v$.

If $P \in \mathit{LOAD}(r, x, \mu)$ then $(\exists v \in \mathcal{V})$
(1) if $d, e \in E$ then $d = e$,
(2) $\lambda(e) = \mathsf{R}^{\mu} x v$,
(3) $\tau^{D}(\psi)$ implies $v{=}r \Rightarrow \psi$, if $(E \cap D) \neq \emptyset$,
(4) $\tau^{D}(\psi)$ implies ψ, if $(E \cap D) = \emptyset$.

$$
\begin{aligned}
[\![\mathtt{if}(M)\{S_1\}\,\mathtt{else}\,\{S_2\}]\!] &= \mathit{IF}(M{\neq}0,\ [\![S_1]\!],\ [\![S_2]\!]) \\
[\![x^{\mu} := M]\!] &= \mathit{STORE}(x,\ M,\ \mu) \qquad [\![\mathtt{skip}]\!] = \mathit{SKIP} \\
[\![r := x^{\mu}]\!] &= \mathit{LOAD}(r,\ x,\ \mu) \qquad [\![S_1 \Vdash S_2]\!] = [\![S_1]\!] \Vdash [\![S_2]\!] \\
[\![r := M]\!] &= \mathit{LET}(r,\ M) \qquad [\![S_1\,;\ S_2]\!] = [\![S_1]\!];\ [\![S_2]\!] \\
[\![\mathsf{F}^{\mu}]\!] &= \mathit{FENCE}(\mu)
\end{aligned}
$$

Full versions:

If $P \in \mathit{STORE}(x, M, \mu)$ then $(\exists v : E \to \mathcal{V})$ $(\exists \theta : E \to \Phi)$
(1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
(2) $\lambda(e) = \mathsf{W}^{\mu} x v_e$,
(3) $\kappa(e)$ implies $\theta_e \wedge M{=}v_e$,
(4) $\tau^{D}(\psi)$ implies $\theta_e \Rightarrow \psi[M/x]$,
(5) $\checkmark$ implies $\bigvee_{e \in E} \theta_e$.

If $P \in \mathit{LOAD}(r, x, \mu)$ then $(\exists v : E \to \mathcal{V})$ $(\exists \theta : E \to \Phi)$
(1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
(2) $\lambda(e) = \mathsf{R}^{\mu} x v_e$
(3) $\kappa(e)$ implies θ_e,
(4) $(\forall e \in E \cap D)$ $\tau^{D}(\psi)$ implies $\theta_e \Rightarrow v_e{=}s_e \Rightarrow \psi[s_e/r]$,
(5) $(\forall e \in E \setminus D)$ $\tau^{D}(\psi)$ implies $\theta_e \Rightarrow (v_e{=}s_e \vee x{=}s_e) \Rightarrow \psi[s_e/r]$,
(6) $(\forall s)$ $\tau^{D}(\psi)$ implies $(\bigwedge_{e \in E} \neg\theta_e) \Rightarrow \psi[s/r]$.

In *IF*, (4) stops coalescing the rf in

$$\mathtt{if}(b)\{r := x \parallel x := 1\}\,\mathtt{else}\,\{r := x\,;\ x := 1\}$$

In diagrams, we use different colors for arrows. We distinguish rf edges that are included in order from those that are not.

- $e \rightarrow d$ arises from rf, where $e \trianglelefteq d$,
- $e \dashrightarrow d$ arises from rf, where $\neg(e \trianglelefteq d)$.

To help the reader understand why order is included, we also different colors for arrows induced by order. We adopt the following conventions:

- $e \dashrightarrow d$ arises from *fulfillment*,
- $e \rightarrow d$ arises from control/data/address *dependency*,
- $e \Rightarrow d$ arises from *synchronized access*.

1.3 Pomsets

We first consider a fragment of our language that can be modeled using simple pomsets. This captures read and write actions which may be reordered, but as we shall see does *not* capture control or data dependencies.

Definition 1.5. 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. Let Pom be the set of all 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 1.6. Action $(\mathsf{W}xv)$ *matches* $(\mathsf{R}xw)$ when $v = w$. Action $(\mathsf{W}xv)$ *blocks* $(\mathsf{R}xw)$, for any v, w.

A read 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 $e \leq c$.

We introduce reorderability [?] in order to provide examples with coherence in this subsection. In §?? we show that coherence can be encoded in the logic, making reorderability unnecessary.

Definition 1.7. Actions a and b are *reorderable* $(a \bowtie b)$ if either both are reads or they are accesses to different locations. Formally $\bowtie = \{(\mathsf{R}xv, \mathsf{R}yw)\} \cup \{(\mathsf{R}xv, \mathsf{W}yw), (\mathsf{W}xv, \mathsf{R}yw), (\mathsf{W}xv, \mathsf{W}yw) \mid x \neq y\}$.

Actions that are not reorderable are in *conflict*.

We can now define a model of processes given as sets of pomsets sufficient to give the semantics for a fragment of our language without control or data dependencies.

Definition 1.8. 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) $E = (E_1 \cup E_2)$,
(2) if $d \leq_1 e$ then $d \leq e$,
(3) if $d \leq_2 e$ then $d \leq e$,
(4) if $e \in E_1$ then $\lambda(e) = \lambda_1(e)$,
(5) if $e \in E_2$ then $\lambda(e) = \lambda_2(e)$,
(6) E_1 and E_2 are disjoint.
If $P \in (a \to \mathcal{P}_2)$ then $(\exists E_1)\ (\exists P_2 \in \mathcal{P}_2)$
(1) $E = (E_1 \cup E_2)$,
(2) if $d, e \in E_1$ then $d = e$,
(3) if $d \leq_2 e$ then $d \leq e$,
(4) if $e \in E_1$ then $\lambda(e) = a$,
(5) if $e \in E_2$ then $\lambda(e) = \lambda_2(e)$,
(6) if $d \in E_1, e \in E_2$ then either $d \leq e$ or $a \bowtie \lambda_2(e)$.
If $P \in \mathit{TOP}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$
(1) $E = E_1$,
(2) $\lambda(e) = \lambda_1(e)$,
(3) if $d \leq_1 e$ then $d \leq e$,
(4) if $\lambda_1(e)$ is a read then e is fulfilled (Def 1.6).

Definition 1.9. For a language fragment, the semantics is:

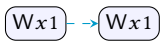
$$[\![x^{\mu} := v\,;\, S]\!] = (\mathsf{W}xv) \rightarrow [\![S]\!] \qquad [\![\mathtt{skip}]\!] = [\![\mathtt{0}]\!] = \mathit{NIL}$$
$$[\![r := x^{\mu}\,;\, S]\!] = \bigcup_v (\mathsf{R}xv) \rightarrow [\![S]\!] \qquad [\![S_1 \parallel S_2]\!] = [\![S_1]\!] \parallel [\![S_2]\!]$$

In this semantics, both `skip` and `0` map to the empty pomset. Parallel composition is disjoint union, inheriting labeling and order from the two sides. Prefixing may add a new action (on the left) to an existing pomset (on the right), inheriting labeling and order from the right.

It is worth noting that if $\bowtie$ is taken to be the empty relation, then top-level pomsets of Def 1.5 correspond to sequentially consistent executions up to mumbling [**?**].

Example 1.10. Mumbling is allowed, since there is no requirement that left and right be disjoint in the definition of prefixing. Both of the pomsets below are allowed.

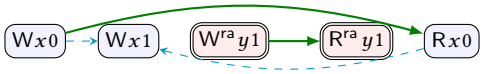
$x := 1\,;\, x := 1$ $\qquad$ $x := 1\,;\, x := 1$



In the left pomset, the order between the events is enforced by clause 6, since the actions are in conflict.

Example 1.11. Although this model enforces coherence, it is very weak. For example, it makes no distinction between synchronizing and relaxed access, thus allowing:

$$x := 0\,;\, x := 1\,;\, y^{\mathsf{ra}} := 1 \parallel r := y^{\mathsf{ra}}\,;\, s := x$$



We show how to enforce the intended semantics, where $(\mathsf{W}^{\mathsf{ra}}y1)$ *publishes* $(\mathsf{W}x1)$ in Ex **??**.

In diagrams, we use different shapes and colors for arrows and events. These are included only to help the reader understand why order is included. We adopt the following conventions (dependency and synchronization order will appear later in the paper):

- relaxed accesses are blue, with a single border,
- synchronized accesses are red, with a double border,
- $e \rightarrow d$ arises from fulfillment, where e *matches* d,
- $e \dashrightarrow d$ arises either from fulfillment, where e *blocks* d, or from prefixing, where e was prefixed before d and their actions *conflict*,
- $e \rightarrow d$ arises from control/data/address *dependency*,
- $e \Rightarrow d$ arises from *synchronized access*.

Definition 1.12. $\mathcal{P}_1$ *refines* $\mathcal{P}_2$ if $\mathcal{P}_1 \subseteq \mathcal{P}_2$.

Example 1.13. Ex 1.10 shows that $[\![x := 1]\!]$ refines $[\![x := 1\,;\, x := 1]\!]$.

1.4 Pomsets with Preconditions

The previous section modeled a language fragment without conditionals (and hence no control dependencies) or expressions (and hence no data dependencies). We now address this, by adopting a *pomsets with preconditions* model similar to [**?**].

Definition 1.14. A *pomset with preconditions* is a pomset (Def 1.5) together with $\kappa : E \rightarrow \Phi$.

Definition 1.15. Let $[\phi/\mathsf{Q}]$ substitute all quiescence symbols by ϕ.

We can now define a model of processes given as sets of pomsets with preconditions sufficient to give the semantics for a fragment of our language where every use of sequential composition is either $(x^\mu := M\,;\, S)$ or $(r := x^\mu\,;\, S)$.

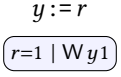
Definition 1.16. If $P \in \mathit{NIL}$ then $E = \emptyset$.
If $P \in \mathcal{P}_1 \parallel \mathcal{P}_2$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$
1–6) as for $\parallel$ in Def 1.8,
(7) if $e \in E_1$ then $\kappa(e)$ implies $\kappa_1(e)$,
(8) if $e \in E_2$ then $\kappa(e)$ 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 1.8 (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 $(\mathsf{W}xv) \rightarrow \mathcal{P}_2$ in Def 1.8,
(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 $(\mathsf{R}xv) \rightarrow \mathcal{P}_2$ in Def 1.8,
(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 $\kappa_2(e)$.
If $P \in \mathit{TOP}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$
1–4) as for *TOP* in Def 1.8,
(5) if $\lambda_1(e)$ is a write, $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}][\mathsf{tt}/\mathsf{W}]$ is a tautology,
(6) if $\lambda_1(e)$ is a read, $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}][\mathsf{ff}/\mathsf{W}]$ is a tautology.

Let PomPre be the set of all pomsets with preconditions. The function $\mathit{TOP} : 2^{\mathsf{PomPre}} \rightarrow 2^{\mathsf{Pom}}$ embeds sets of pomsets with preconditions into sets of pomsets. It also substitutes formulae for quiescence and write symbols, for use in §??-??. In these "top-level" pomsets, every read is fulfilled and every precondition is a tautology.

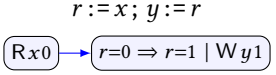
Definition 1.17. For a language fragment, the semantics is:

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

Example 1.18. A simple example of a data dependency is a pomset $P \in [\![r := x\,;\, y := r]\!]$, for which there must be an $v \in \mathcal{V}$ and $P' \in [\![y := r]\!]$ such as the following, where $v = 1$:



The value chosen for the read may be different from that chosen for the write:



In this case, the pomset's preconditions depend on a bound register, so cannot contribute to a top-level pomset.

If the values chosen for read and write are compatible, then we have two cases: the independent case, which again cannot be part of a top-level pomset,

$$r := x;\ y := r$$

$$\boxed{\mathsf{R}\,x\,1} \qquad \boxed{r{=}1 \mid \mathsf{W}\,y\,1}$$

and the dependent case:

$$\boxed{\mathsf{R}\,x\,1} \longrightarrow \boxed{r{=}1 \Rightarrow r{=}1 \mid \mathsf{W}\,y\,1}$$

Since $r{=}1 \Rightarrow r{=}1$ is a tautology, this can be part of a top-level pomset.

Example 1.19. Control dependencies are similar, for example for any $P \in [\![r := x;\ \mathtt{if}(r)\{y := 1\}]\!]$, there must be an $v \in \mathcal{V}$ and $P' \in [\![\mathtt{if}(r)\{y := 1\}]\!]$ such as:

$$\mathtt{if}(r)\{y := 1\}$$

$$\boxed{r{\neq}0 \mid \mathsf{W}\,y\,1}$$

The rest of the reasoning is the same as Ex 1.18.

Example 1.20. A simple example of an independency is a pomset $P \in [\![r := x;\ y := 1]\!]$, for which there must be:

$$y := 1$$

$$\boxed{1{=}1 \mid \mathsf{W}\,y\,1}$$

In this case it doesn't matter what value the read chooses:

$$r := x;\ y := 1$$

$$\boxed{\mathsf{R}\,x\,0} \qquad \boxed{1{=}1 \mid \mathsf{W}\,y\,1}$$

Example 1.21. Consider $P \in [\![\mathtt{if}(r{=}1)\{y := r\}\,\mathtt{else}\,\{y := 1\}]\!]$, so there must be $P_1 \in [\![y := r]\!]$, and $P_2 \in [\![y := 1]\!]$, such as:

$$y := r \qquad\qquad y := 1$$

$$\boxed{r{=}1 \mid \mathsf{W}\,y\,1} \qquad\qquad \boxed{1{=}1 \mid \mathsf{W}\,y\,1}$$

Since there is no requirement for disjointness in the semantics of conditionals, we can consider the case where the event *coalesces* from the two pomsets, in which case:

$$\mathtt{if}(r{=}1)\{y := r\}\,\mathtt{else}\,\{y := 1\}$$

$$\boxed{(r{=}1 \Rightarrow r{=}1) \wedge (r{\neq}1 \Rightarrow 1{=}1) \mid \mathsf{W}\,y\,1}$$

Here, the precondition is a tautology, independent of r.

1.5 Pomsets with Predicate Transformers

Having reviewed the work we are building on, we now turn to the contribution of this paper, which is a model of *pomsets with predicate transformers*. *Predicate transformers* are functions on formulae which preserve logical structure, providing a natural model of sequential composition.

Definition 1.22. A *predicate transformer* is a function $\tau : \Phi \longrightarrow \Phi$ such that

- $\tau(\mathsf{ff})$ is ff,
- $\tau(\psi_1 \wedge \psi_2)$ is $\tau(\psi_1) \wedge \tau(\psi_2)$,
- $\tau(\psi_1 \vee \psi_2)$ is $\tau(\psi_1) \vee \tau(\psi_2)$,

- if ϕ implies ψ, then $\tau(\phi)$ implies $\tau(\psi)$.

Note that substitutions ($\tau(\psi) = \psi[M/r]$) and implications on the right ($\tau(\psi) = \phi \Rightarrow \psi$) are predicate transformers.

As discussed in §??, predicate transformers suffice for sequentially consistent models, but not relaxed models, where dependency calculation is crucial. For dependency calculation, we use a *family* of predicate transformers, indexed by sets of events. We use τ^D as the predicate transformer applied to any event e where if $d \in D$ then $d < e$.

Definition 1.23. 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 $\tau^D(\psi)$.

Definition 1.24. A *pomset with predicate transformers* is a pomset with preconditions (Def 1.16), together with a family of predicate transformers for E.

Definition 1.25. 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 1.16,
(9) $\tau^D(\psi)$ implies $(\phi \Rightarrow \tau_1^D(\psi)) \wedge (\neg\phi \Rightarrow \tau_2^D(\psi))$.

If $P \in \mathcal{P}_1 \Vdash \mathcal{P}_2$ then $(\exists P_1 \in \mathcal{P}_1)$ $(\exists P_2 \in \mathcal{P}_2)$
1–8) as for $\|$ in Def 1.16,
(9) $\tau^D(\psi)$ implies $\tau_2^D(\psi)$,
(10) $\tau^D(\mathsf{s})$ implies $\tau_1^D(\mathsf{s})$, for every quiescence symbol s.

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 1.8 (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 $\psi \wedge M{=}v$,
S5) $\tau^C(\psi)$ implies ψ,
where $D \cap E \neq \emptyset$ and $C \cap E = \emptyset$.

If $P \in \mathit{LOAD}(r, x, \mu)$ then $(\exists v \in \mathcal{V})$
L1) if $d, e \in E$ then $d = e$,
L2) $\lambda(e) = \mathsf{R}xv$,
L3) $\kappa(e)$ implies tt,
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies ψ,
where $D \cap E \neq \emptyset$ and $C \cap E = \emptyset$,

If $P \in TOP(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$
1–6) as in Def 1.16,
(7) $\tau^{E_1}(\mathsf{s})$ implies s, for every quiescence symbol s.

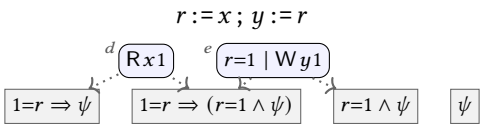
Definition 1.26. The semantics of commands is:

$$\begin{aligned}
[\![\mathtt{if}(M)\{S_1\}\,\mathtt{else}\,\{S_2\}]\!] &= IF(M{\neq}0,\ [\![S_1]\!],\ [\![S_2]\!]) \\
[\![x^\mu := M]\!] &= STORE(x,\ M,\ \mu) \qquad [\![\mathtt{skip}]\!] = SKIP \\
[\![r := x^\mu]\!] &= LOAD(r,\ x,\ \mu) \qquad [\![S_1 \ {}_\gamma\!\!\Vdash\!\!\!\rightarrow\ S_2]\!] = [\![S_1]\!] \Vdash\!\!\!\rightarrow [\![S_2]\!] \\
[\![r := M]\!] &= LET(r,\ M) \qquad [\![S_1\,;\,S_2]\!] = [\![S_1]\!]\,;\,[\![S_2]\!] \\
[\![\mathsf{F}^\mu]\!] &= FENCE(\mu)
\end{aligned}$$

Most of these definitions are straightforward adaptations of §1.4, but the treatment of sequential composition is new. This uses the usual rule for composition of predicate transformers (but preserving the indexing set). For the pomset, we take the union of their events, preserving actions, but crucially in cases 7 and 8 we apply a predicate transformer τ_1^C from the left-hand side to a precondition $\kappa_2(e)$ from the right-hand side to build the precondition $\kappa_2'(e)$. The indexing set C for the predicate transformer is $\{c \mid c < e\}$, so can depend on the causal order.

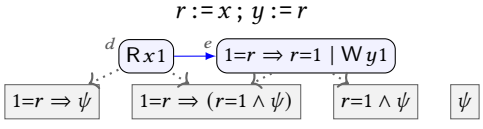
Example 1.27. For read to write dependency, consider:

Putting these together without order, we calculate the precondition $\kappa(e)$ as $\tau_1^C(\kappa_2(e))$, where C is $\{c \mid c < e\}$, which is $\emptyset$. Since $\tau_1^\emptyset(\psi)$ is ψ, this gives that $\kappa(e)$ is $\kappa_2(e)$, which is $r{=}1$. This gives the pomsaet with predicate transformers:



This pomset's preconditions depend on a bound register, so cannot contribute to a top-level pomset.

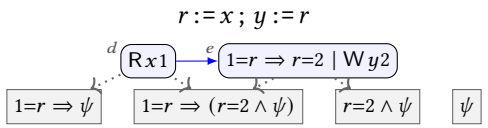
Putting them together with order, we calculate the precondition $\kappa(e)$ as $\tau_1^C(\kappa_2(e))$, where C is $\{c \mid c < e\}$, which is $\{d\}$. Since $\tau_1^{\{d\}}(\psi)$ is $(1{=}r \Rightarrow \psi)$, this gives that $\kappa(e)$ is $(1{=}r \Rightarrow \kappa_2(e))$, which is $(1{=}r \Rightarrow r{=}1)$. This gives the pomset with predicate transformers:



This pomset's preconditions do not depend on a bound register, so can contribute to a top-level pomset.

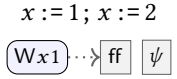
Example 1.28. If the read and write choose different values:

Putting these together with order, we have the following, which cannot be part of a top-level pomset:



Example 1.29. S4 includes $M{=}v$ to ensure that spurious merges do not go undetected. Consider the following.

Merging the actions, since 2=1 is unsatisfiable, we have:



This pomset cannot be part of a top-level pomset, since $\tau^E(\mathsf{s}) = \mathsf{ff}$ for every quiescence symbol s. This is what we would hope: that the program $x := 1 ; x := 2$ should only be top-level if there is a $(\mathsf{W}x2)$ event.

Example 1.30. The predicate transformer we have chosen for L4 is different from the one used traditionally, which is written using substitution. Substitution is also used in [?]. Attempting to write the predicate transformers in this style we have:

L4) $\tau^D(\psi)$ implies $\psi[v/r]$,

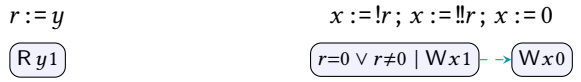
L5) $\tau^C(\psi)$ implies $(\forall r)\psi$.

This phrasing of L5 says that ψ must be independent of r in order to appear in a top-level pomset. This choice for L5 is forced by Def 1.23, which states that the predicate transformer for a small subset of E must imply the transformer for a larger subset.

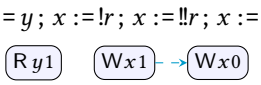
Sadly, this definition fails associativity.

Consider the following, eliding transformers:

Associating to the right and merging:



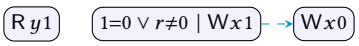
The precondition of $(\mathsf{W}x1)$ is a tautology, thus we have:



If, instead, we associate to the left:

Sequencing and merging:

$$r := y \,;\; x := !r \,;\; x := !!r \,;\; x := 0$$



In this case, the precondition of $(\mathsf{W}x1)$ is not a tautology, forcing a dependency $(\mathsf{R}y1) \longrightarrow (\mathsf{W}x1)$.

Our solution is to Skolemize. We have proven associativity of Def 1.25 in Agda. The proof requires that predicate transformers distribute through disjunction (Def 1.22). Since universal quantification does not distribute through disjunction, the attempt to define predicate transformers using substitution fails (in particular for L5.)

1.6 The Road Ahead

The final semantic functions for load, store, and thread initialization are given in Fig **??**, at the end of the paper. In §**??**–**??**, we explain this definition by looking at its constituent parts, building on Def 1.25. In §**??**, we add *quiescence*, which encodes coherence, release-acquire access, and SC access. In §**??**, we add peculiarities that are necessary for efficient implementation on ARM8. In §**??**, we discuss other features such as invariant reasoning, case analysis and register recycling.

The final definitions of load and store are quite complex, due to the inherent complexities of relaxed memory. The core of Def 1.25, modeling sequential composition, parallel composition, and conditionals, is stable, remaining unchanged in later sections. The messiness of relaxed memory is quarantined to the rules for load and store, rather than permeating the entire semantics.