Sequential Composition for Relaxed Memory: Pomsets with Predicate Transformers

Anonymous
Anonymous Institution

***Abstract*—This paper presents the first semantics for relaxed memory with a compositional definition of sequential composition. Previous definitions of relaxed memory have given detailed treatments of parallel composition, but have given sequential composition less attention, often relegating it to a (sometimes speculative) operational semantics of single-threaded programs. In this paper we show how sequential composition can be restored to a first-class citizen, by giving it a denotational semantics in a model of pomsets with preconditions, extended with a family of 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 is the first paper to show how they can be combined.**

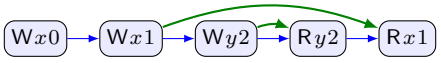
1. Introduction

This paper is about the interaction of two of the fundamental building blocks of computing: memory and sequential composition. One would like to think that these are well-worn topics, where every issue has been settled, but this is sadly not the case.

1.1. Memory

For single-threaded programs, memory can be thought of as you might expect: programs write to, and read from, memory references. This can be thought of as a total order of reads and writes, where each read has a matching *fulfilling* write, for example:

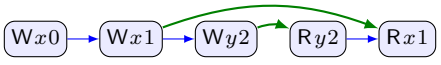
$$x{:=}\,0; x{:=}\,1; y{:=}\,2; r{:=}\,y; s{:=}\,x$$



(In examples, r–s range over thread-local registers and x-z range over shared memory references.)

This model naturally extends to the case of shared-memory concurrency in a natural way, leading to a *sequentially consistent* semantics, in which *program order* inside a thread implies a total *causal order* between read and write events, for example:

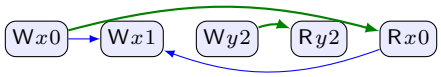
$$x{:=}\,0; x{:=}\,1; y{:=}\,2 \parallel r{:=}\,y; s{:=}\,x$$



Unfortunately, this model does not compile efficiently to commodity hardware, resulting in a 37–73% increase in CPU time [15] on ARM, and hence power consumption. Developers of software and compilers have therefore been faced with a difficult trade-off, between an elegant model of memory, and its impact on resource usage (such as size of data centers, electricity bills and carbon footprint). Unsurprisingly, many have chosen to prioritize efficiency over elegance.

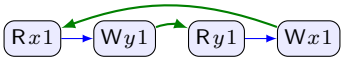
This has led to *relaxed memory models*, in which the requirement of sequential consistency is weakened to only apply *per-location* and not globally over the whole program. This allows executions which are inconsistent with program order, such as:

$$x{:=}\,0; x{:=}\,1; y{:=}\,2 \parallel r{:=}\,y; s{:=}\,x$$



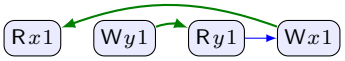
In such models, the causal order between events is important, and includes control and data dependencies, to avoid paradoxical "out of thin air" examples such as:

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



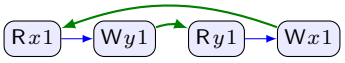
This candidate execution forms a cycle in causal order, so is disallowed, but this depends crucially on the control dependency from $(\mathsf{R}x1)$ to $(\mathsf{W}y1)$, and the data dependency from $(\mathsf{R}y1)$ to $(\mathsf{W}x1)$. If either is missing, then this execution is acyclic and hence allowed. For example dropping the control dependency results in:

$$r{:=}\,x; y{:=}\,1 \parallel s{:=}\,y; x{:=}\,s$$



Unfortunately, while a simple syntactic approach to dependency calculation suffices for hardware models, it is not preserved by common compiler optimizations. For example, if we calculate control dependencies syntactically, then there is a dependency from $(\mathsf{R}x1)$ to $(\mathsf{W}y1)$, and therefore a cycle in, the candidate execution:

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

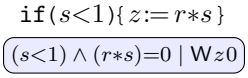


An optimizing compiler might lift the assignment $y{:=}\,1$ out of the conditional, thus removing the control dependency.

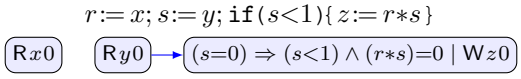
Prominent solutions to the problem of dependency calculation include:

- *syntactic* methods used in hardware models such as ARM or x86-TSO [2],
- *speculative execution* methods (which give a semantics based on multiple executions of the same program) such as the Java Memory Model [16] and related models [11, 13, 6],
- *rewriting* methods, which give an operational model up to syntactic rewrites, such as [18], and
- *logical* methods, such as the pomsets with preconditions model of [12].

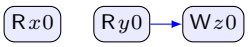
In this paper, we will focus on logical models, as those are compositional, and align well with existing models of sequential composition. The heart of the model of [12] is to add logical preconditions to events, which are introduced by store actions (modeling data dependencies) and conditionals (modeling control dependencies):



Preconditions are discharged by being ordered after a read:



Note that there is dependency order from $(\mathsf{R}y0)$ to $(\mathsf{W}z0)$ so the precondition for $(\mathsf{W}z0)$ only has to be satisfied assuming the hypothesis $(s{=}0)$. There is no matching order from $(\mathsf{R}x0)$ to $(\mathsf{W}z0)$ which is why we do not assume the hypothesis $(r{=}0)$. Nonetheless, the precondition on $(\mathsf{W}z0)$ is a tautology, and so can be elided in the diagram:



While existing models of relaxed memory have detailed treatments of parallel composition, they often give sequential composition little attention, either ignoring it altogether, or treating it operationally with its usual small-step semantics. This paper investigates how existing models of sequential composition interact with relaxed memory.

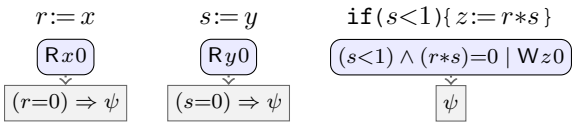
1.2. Sequential composition

Our approach follows that of weakest precondition semantics of Dijkstra [7], which provides an alternative characterization of Hoare logic [10] by mapping postconditions to preconditions. We recall the definition of $wp_S(\psi)$ for loop-free code below.

- $wp_{\texttt{skip}}(\psi) = \psi$
- $wp_{\texttt{abort}}(\psi) = \mathsf{ff}$
- $wp_{r:=M}(\psi) = \psi[M/r]$
- $wp_{S_1;S_2}(\psi) = wp_{S_1}(wp_{S_2}(\psi))$
- $wp_{\texttt{if}(M)\{S_1\}\texttt{else}\{S_2\}}(\psi) = ((M{\neq}0) \Rightarrow wp_{S_1}(\psi)) \wedge ((M{=}0) \Rightarrow wp_{S_2}(\psi))$

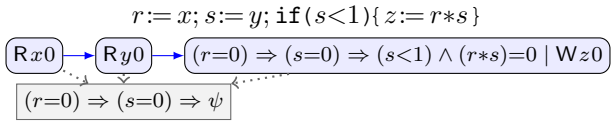
The rule we are most interested in is the one for sequential composition, which maps sequential composition of programs to function composition of predicate transformers.

Predicate transformers are a good fit to logical models of dependency calculation, since both are concerned with preconditions, and how they are transformed by sequential composition. Our first attempt is to associate a predicate transformer with each pomset. We visualize this in diagrams by showing how ψ is transformed, for example:



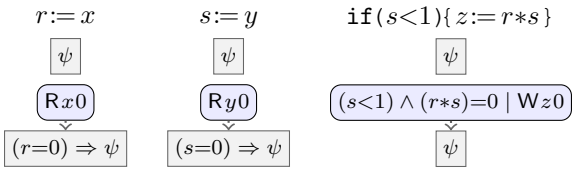
In the rightmost program above, the write to z affects the shared store, not the local state of the thread, therefore we assign it the identity transformer.

For the sequentially consistent semantics, sequential composition is straightforward: we apply each predicate transformer to the preconditions of subsequent events, and compose the predicate transformers:



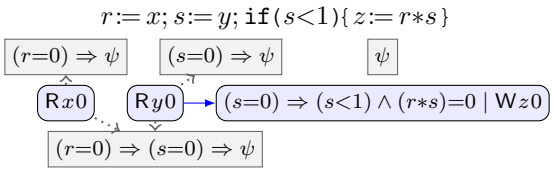
This model works for the sequentially consistent case, but needs to be weakened for the relaxed case. The key observation of this paper is that rather than working with one predicate transformer, we should work with a *family* of predicate transformers, indexed by sets of events.

For example, for single-event pomsets, there are two predicate transformers, since there are two subsets of any one-element set. We call the predicate transformer for $\emptyset$ the *independent* transformer, and the one indexed by $\{e\}$ the *dependent* transformer. We visualize this by including more than one transformed predicate, with an edge leading to the dependent one. For example:

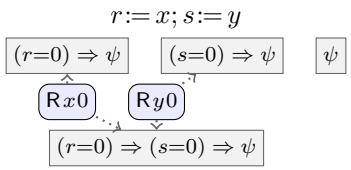


The model of sequential composition then picks which predicate transformer to apply to an event's precondition by picking the one indexed by all the events before it in causal order.

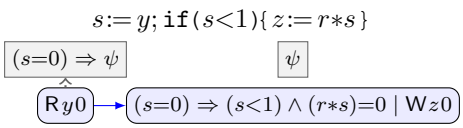
For example, we can recover the expected semantics for the above example by choosing the predicate transformer which is independent of $(\mathsf{R}x0)$ but dependent on $(\mathsf{R}y0)$, which is the transformer which maps ψ to $(s{=}0) \Rightarrow \psi$.



As a sanity check, we can see that sequential composition is associative in this case, since it does not matter whether we associate to the left, with intermediate step:



or to the right, with intermediate step:



This is an instance of a general result that sequential composition forms a monoid, as one would hope.

1.3. Contributions

This paper is the first model of relaxed memory with a compositional semantics for sequential composition. It shows how pomsets with preconditions [12] can be combined with predicate transformers [7].

- §2 presents the basic model, with few features required of the logic of preconditions, but a resulting lack of fidelity to exiting models,
- §3 adds a model of *quiescence* to the logic, required to model coherence (accessing x has a precondition that x is quiescent) and synchronization (a releasing write requires all locations to be quiescent),
- §4 adds the features required for efficient compilation to modern architectures: downgrading some synchronized accesses to relaxed, and removing read-read dependencies,
- §5 show how to address common litmus tests, and
- §6 is a discussion of the design space.

The definitions in this paper have been formalized in Agda.

Because it is closely related, we expect that the memory-model results of [12] apply to our model, including compositional reasoning for temporal safety properties and local SC-DRF. In §4, we provide an alternative proof strategy for efficient compilation to ARM8, which improves upon that of [12] by using a recent alternative characterization of ARM8.

As far as we are aware, there are no previous attempts to provide a compositional semantics of sequential composition in a relaxed memory model. For a discussion of related work for relaxed memory models in general, see [12].

2. 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*) [9, 19], 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, and parallel and sequential composition are as defined in this section.

2.1. Preliminaries

The syntax is built from

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

Memory 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,
- 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.

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

Memory modes, μ, are relaxed (rlx), release-acquire (ra), and sequentially consistent (sc). Relaxed 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. *Thread groups*, G, include commands and 0, which denotes inaction. The `fork` command spawns a thread group.

The semantics is built from the following.

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

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

We require that:

- actions include writes ($\mathsf{W}xv$) and reads ($\mathsf{R}xv$),
- formulae include equalities ($M{=}N$) and ($x{=}M$),
- formulae include symbols Q_{sc}, $\mathsf{Q}^x_{\mathsf{ro}}$, $\mathsf{Q}^x_{\mathsf{wo}}$, $\downarrow^x$, W, (which are used in §3–4),
- formulae are closed under negation, conjunction, disjunction, and substitutions $[M/r]$, $[M/x]$, and $[\phi/\mathsf{s}]$ for each symbol s,
- 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 §5.1, 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 §2–4 we additionally require that

- each register appears at most once in a program.

In §5, we drop this restriction, requiring instead that

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

2.2. 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.

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

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

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

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

Def 2. 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 $e \leq c$.

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

We introduce independency [17] in order to provide examples with coherence in this subsection. In §3 we show that coherence can be encoded in the logic, making independency unnecessary.

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

Actions that are not independent 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.

Def 4. If $P \in \mathit{NIL}$ then $E = \emptyset$.
If $P \in (\mathcal{P}_1 \parallel \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

1) $E = (E_1 \cup E_2)$,
2) if $e \in E_1$ then $\lambda(e) = \lambda_1(e)$,
3) if $e \in E_2$ then $\lambda(e) = \lambda_2(e)$,
4) if $d \leq_1 e$ then $d \leq e$,
5) if $d \leq_2 e$ then $d \leq e$,
6) E_1 and E_2 are disjoint.

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

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

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

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

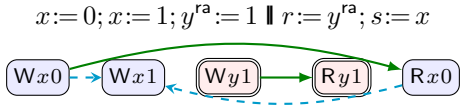
In this semantics, both `skip` and 0 map to the empty pomset. Parallel composition is disjoint union, inheriting labeling and order from the two side. Prefixing may add a new action (on the left) to an existing pomset (on the right), inheriting labeling and order from the right.

It is worth noting that if $\leftrightarrow$ is taken to be the empty relation, then fulfilled pomsets of Def 1 correspond to sequentially consistent executions [14] up to mumbling [5].

Ex 6. 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.

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

Ex 7. Although this model enforces coherence, it is very weak. For example, it makes no distinction between synchronizing and relaxed access, thus allowing:



We show how to enforce the intended semantics, where $(\mathsf{W}y1)$ *publishes* $(\mathsf{W}x1)$ in Ex 31.

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

Def 8. $\mathcal{P}_1$ *refines* $\mathcal{P}_2$ if $\mathcal{P}_1 \subseteq \mathcal{P}_2$.

Ex 9. Ex 6 shows that $[\![x := 1]\!]$ refines $[\![x := 1; x := 1]\!]$.

2.3. 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 [12]. We discuss the differences in §6.

Def 10. A *pomset with preconditions* is a pomset (Def 1) together with $\kappa : E \to \Phi$.

Def 11. A pomset with preconditions is *top level* if it is fulfilled (Def 2) and every precondition is a tautology.

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 $(x{:=}\,M; S)$ or $(r{:=}\,x; S)$.

Def 12. If $P \in \mathit{NIL}$ then $E = \emptyset$.
If $P \in (\mathcal{P}_1 \parallel \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

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

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

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

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

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

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

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

Def 13. For a language fragment, the semantics is:

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

Ex 14. 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:

$$y{:=}\,r$$
$$\boxed{r{=}1 \mid \mathsf{W}y1}$$

If v is chosen badly, we have a pomset with a precondition that cannot be part of a top-level pomset such as:

$$r{:=}\,x; y{:=}\,r$$
$$\boxed{\mathsf{R}x0} \to \boxed{r{=}0 \Rightarrow r{=}1 \mid \mathsf{W}y1}$$

But if v is 1 then we have two cases, the independent case, which again cannot be part of a top-level pomset:

$$r{:=}\,x; y{:=}\,r$$
$$\boxed{\mathsf{R}x1} \qquad \boxed{r{=}1 \mid \mathsf{W}y1}$$

or the dependent case:

$$r{:=}\,x; y{:=}\,r$$
$$\boxed{\mathsf{R}x1} \to \boxed{r{=}1 \Rightarrow r{=}1 \mid \mathsf{W}y1}$$

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

Ex 15. Control dependencies are similar, for example for any $P \in [\![r{:=}\,x; \texttt{if}(r)\{\,y{:=}\,1\,\}]\!]$, there must be an $v \in \mathcal{V}$ and $P' \in [\![\texttt{if}(r)\{\,y{:=}\,1\,\}]\!]$ such as:

$$\texttt{if}(r)\{\,y{:=}\,1\,\}$$
$$\boxed{r{\neq}0 \mid \mathsf{W}y1}$$

The rest of the reasoning is the same as for a data dependency.

Ex 16. A simple example of an independency is a pomset $P \in [\![r{:=}\,x; y{:=}\,1]\!]$, for which there must be an $v \in \mathcal{V}$ and $P' \in [\![y{:=}\,r]\!]$ such as:

$$y{:=}\,1$$
$$\boxed{1{=}1 \mid \mathsf{W}y1}$$

In this case it doesn't matter what v is, for example:

$$r{:=}\,x; y{:=}\,1$$
$$\boxed{\mathsf{R}x0} \qquad \boxed{1{=}1 \mid \mathsf{W}y1}$$

Ex 17. Consider $P \in [\![\texttt{if}(r{=}1)\{\,y{:=}\,r\,\}\texttt{else}\{\,y{:=}\,1\,\}]\!]$, so there must be $P_1 \in [\![y{:=}\,r]\!]$, and $P_2 \in [\![y{:=}\,1]\!]$, such as:

$$y{:=}\,r \qquad\qquad y{:=}\,1$$
$$\boxed{r{=}1 \mid \mathsf{W}y1} \qquad\qquad \boxed{1{=}1 \mid \mathsf{W}y1}$$

Since there is no requirement for disjointness in the semantics of conditionals, we can consider the case where the pomsets share their event, in which case:

$$\texttt{if}(r{=}1)\{\,y{:=}\,r\,\}\texttt{else}\{\,y{:=}\,1\,\}$$
$$\boxed{(r{=}1 \Rightarrow r{=}1) \wedge (r{\neq}1 \Rightarrow 1{=}1) \mid \mathsf{W}y1}$$

where the precondition on $(\mathsf{W}y1)$ is a tautology, and so is independent of r.

2.4. Pomsets with Predicate Transformers

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

The final semantic functions for load and store, given in Figure 1, are quite complex. We explain the definition by looking at its constituent parts, starting with Def 23, below, which captures dependency. In §3, we add *quiescence*, which encodes coherence, release-acquire and SC access, and termination. In §4, we add peculiarities that are necessary for efficient implementation on ARM8. In §5, we discuss the

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

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

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

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

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

T1) $E = E_1$,
T2) $\lambda(e) = \lambda_1(e)$,
T3) $\kappa(e)$ implies $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}][\mathsf{tt}/\mathsf{W}]$ if $\lambda_1(e)$ is a write,
$\kappa(e)$ implies $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}][\mathsf{ff}/\mathsf{W}]$ otherwise.

Figure 1. Full Semantics of Loads, Stores and Threads (See Def 34 for $\mathsf{Q}^{\mathsf{W}}/\mathsf{Q}^{\mathsf{R}}$ and Def 39 for ds/dl)

complications required to validate if-closure and to allow address calculation.

Def 18. A *predicate transformer* is a function $\tau : \Phi \to \Phi$ such that

- $\tau(\mathsf{ff})$ is ff,
- $\tau(\psi_1 \wedge \psi_2)$ is $\tau(\psi_1) \wedge \tau(\psi_2)$,
- $\tau(\psi_1 \vee \psi_2)$ is $\tau(\psi_1) \vee \tau(\psi_2)$,
- if ϕ implies ψ, then $\tau(\phi)$ implies $\tau(\psi)$.

Def 19. 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 $\tau^D(\psi)$.

Note that in a family of predicate transformers for E, transformers for smaller subsets of E are stronger.

Def 20. A pomset with predicate transformers is a pomset with preconditions (Def 12), together with a family of predicate transformers for E.

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

Def 21. If $P \in \mathit{THRD}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$

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

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

F1) $E = E_1$,
F2) $\lambda(e) = \lambda_1(e)$,
F3) $\kappa(e)$ implies $\kappa_1(e)$,
F4) $\tau^D(\psi)$ implies ψ.

Def 22. Adopting *NIL* and $\|$ from Def 12, the semantics of thread groups is:

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

Def 23. 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 12,
9) $\tau^D(\psi)$ implies $(\phi \Rightarrow \tau_1^D(\psi)) \wedge (\neg\phi \Rightarrow \tau_2^D(\psi))$.

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

1–5) as for $\|$ in Def 1 (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 ψ,

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 $\mathtt{tt}$,
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies ψ,
where $D \cap E \neq \emptyset$ and $C \cap E = \emptyset$,

Def 24. The semantics of commands is:

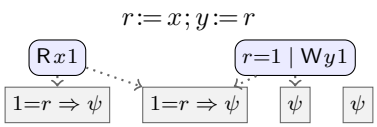
$$[\![\mathtt{if}(M)\{S_1\}\mathtt{else}\{S_2\}]\!] = \mathit{IF}(M{\neq}0, [\![S_1]\!], [\![S_2]\!])$$
$$[\![x^\mu := M]\!] = \mathit{STORE}(x, M, \mu) \qquad [\![\mathtt{abort}]\!] = \mathit{ABORT}$$
$$[\![r := x^\mu]\!] = \mathit{LOAD}(r, x, \mu) \qquad [\![\mathtt{skip}]\!] = \mathit{SKIP}$$
$$[\![r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

[Chat about abort, skip, let and if.]
[Chat about semicolon.]

In the predicate transformers for store and load, S4 and L4 denote the *dependent case*, whereas S5 and L5 denote the *independent case*. For stores, the dependent and independent cases are the same; this will change in the next section, where we introduce quiescence. In the dependent case for load, we can assume that r is the value v, which has appears in the read action, when proving ψ. In the independent case for load, we can only make the weaker assumption that either r is v or it is value defined by preceding code for x. That is, we do not know whether subsequent code sees the value v, or the value of some preceding write of x.

Ex 25. Read to write dependency, first separately:

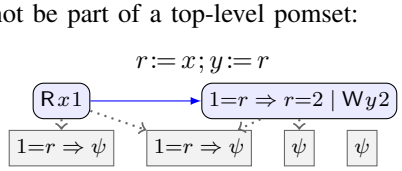
Putting these together without order:



The precondition of $(\mathsf{W}y1)$ can be simplified to $(x{=}r \Rightarrow r{=}1)$, which is only a tautology when $x{=}1$. If we choose a pomset that orders $(\mathsf{R}x1) \rightarrow (\mathsf{W}y1)$, then the predicate transformers are the same, but the precondition of $(\mathsf{W}y1)$ can be weakened to $(1{=}r \Rightarrow r{=}1)$, which is a tautology.

Ex 26. 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:

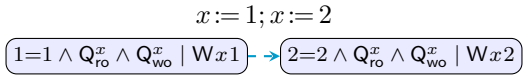


3. Quiescence

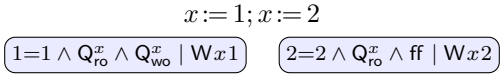
We introduce *quiescence*, which captures *coherence*, *synchronized access*, and *completion*. Recall from §2.1 that formulae include symbols Q_{sc}, $\mathsf{Q}^x_{\mathsf{ro}}$, and $\mathsf{Q}^x_{\mathsf{wo}}$. We refer to these collectively as *quiescence symbols*. In this section, we will show how these logical symbols can be used to capture coherence and synchronization. This illustrates a feature of our model, which is that many features of weak memory can be cautured in the logic, not in the pomset model itself.

3.1. Coherence (CO)

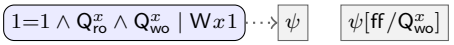
In the logic, the quiescence symbols are just uninterpreted formula, but the semantics uses them as preconditions, to ensure appropriate causal order. For example, *write-write coherence* enforces order between writes to the same location in the same thread. We model this by adding the precondition $(\mathsf{Q}^x_{\mathsf{ro}} \wedge \mathsf{Q}^x_{\mathsf{wo}})$ to events that write to x, for example:



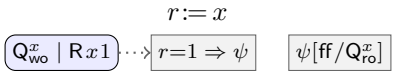
These symbols are left alone in the dependent case, but in the independent case we substitute ff for $\mathsf{Q}^x_{\mathsf{wo}}$:



This substitution is part of the predicate transformer for store:

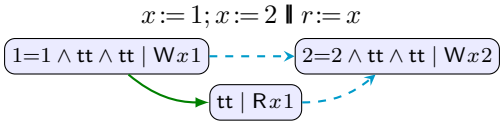


We treat read-write and write-read coherence similarly:



In this model, there is no read-read coherence, but to restore it we would identify $\mathsf{Q}^x_{\mathsf{ro}}$ with $\mathsf{Q}^x_{\mathsf{wo}}$.

When threads are initialized, we substitute every quiesence symbol with $\mathtt{tt}$, so at top level there are no remaining quiescence symbols, for example:



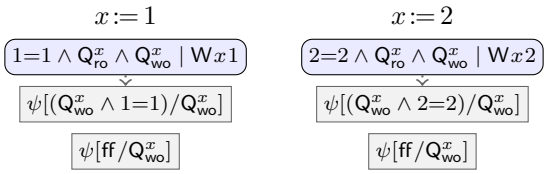
Def 27. Let $[\phi/\mathsf{Q}^*_{\mathsf{ro}}]$ be the substitution that replaces all symbols $\mathsf{Q}^x_{\mathsf{ro}}$ by ϕ, and similarly $[\phi/\mathsf{Q}^*_{\mathsf{wo}}]$.

Def 28 (CO)**.** Update Def 23 to:

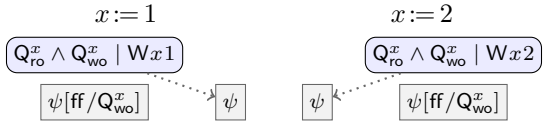
S3) $\kappa(e)$ implies $\mathsf{Q}^x_{\mathsf{ro}} \wedge \mathsf{Q}^x_{\mathsf{wo}} \wedge M{=}v$,
L3) $\kappa(e)$ implies $\mathsf{Q}^x_{\mathsf{wo}}$,
T3) $\kappa(e)$ implies $\kappa_1(e)[\mathtt{tt}/\mathsf{Q}^*_{\mathsf{ro}}][\mathtt{tt}/\mathsf{Q}^*_{\mathsf{wo}}]$,
S4) $\tau^D(\psi)$ implies $\psi[(\mathsf{Q}^x_{\mathsf{wo}} \wedge M{=}v)/\mathsf{Q}^x_{\mathsf{wo}}]$,

S5) $\tau^C(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}^x_{\mathsf{wo}}]$,.
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}^x_{\mathsf{ro}}]$.

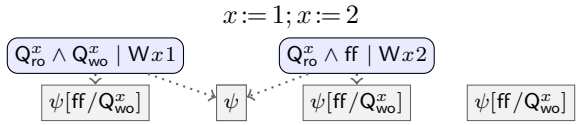
Ex 29. Def 28 enforces coherence. Consider:



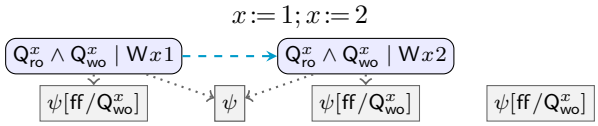
Simplifying, we have:



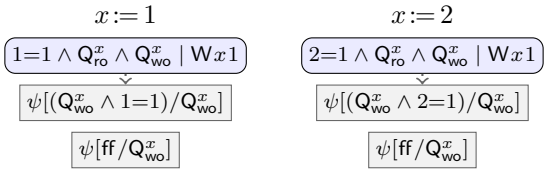
If we attempt to put these together unordered, the precondition of $(\mathsf{W}x2)$ becomes unsatisfiable:



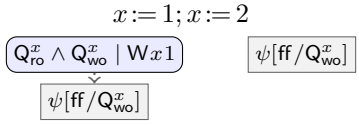
In order to get a satisfiable precondition for $(\mathsf{W}x2)$, we must introduce order:



Ex 30. S4 includes the substitution $\psi[(\mathsf{Q}^x_{\mathsf{wo}} \wedge M{=}v)/\mathsf{Q}^x_{\mathsf{wo}}]$ to ensure that *left merges* are not quiescent. Consider the following.



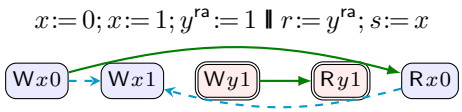
Simplifying: and merging the actions, we have:



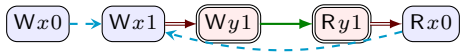
which is what we would hope, that the program $x := 1; x := 2$ should only be quiescent if there is a $(\mathsf{W}x2)$ event.

3.2. Synchronized Access (SYNC)

Ex 31. The publication idiom requires that we disallow the execution below, which is allowed by Def 28.



We disallow this by introducing order $(\mathsf{W}x1) \rightarrow (\mathsf{W}y1)$ and $(\mathsf{R}y1) \rightarrow (\mathsf{R}x0)$.



Def 32. Let formulae $\mathsf{Q}^{\mathsf{W}x}_\mu$ and $\mathsf{Q}^{\mathsf{R}x}_\mu$ be defined:

$$\mathsf{Q}^{\mathsf{W}x}_{\mathsf{rlx}} = \mathsf{Q}^x_{\mathsf{ro}} \wedge \mathsf{Q}^x_{\mathsf{wo}} \qquad \mathsf{Q}^{\mathsf{R}x}_{\mathsf{rlx}} = \mathsf{Q}^x_{\mathsf{wo}}$$
$$\mathsf{Q}^{\mathsf{W}x}_{\mathsf{ra}} = \textstyle\bigwedge_y \mathsf{Q}^y_{\mathsf{ro}} \wedge \mathsf{Q}^y_{\mathsf{wo}} \qquad \mathsf{Q}^{\mathsf{R}x}_{\mathsf{ra}} = \mathsf{Q}^x_{\mathsf{wo}}$$
$$\mathsf{Q}^{\mathsf{W}x}_{\mathsf{sc}} = \textstyle\bigwedge_y \mathsf{Q}^y_{\mathsf{ro}} \wedge \mathsf{Q}^y_{\mathsf{wo}} \wedge \mathsf{Q}_{\mathsf{sc}} \qquad \mathsf{Q}^{\mathsf{R}x}_{\mathsf{sc}} = \mathsf{Q}^x_{\mathsf{wo}} \wedge \mathsf{Q}_{\mathsf{sc}}$$

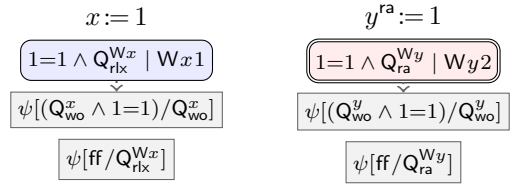
Def 33. Let substitutions $[\phi/\mathsf{Q}^{\mathsf{W}x}_\mu]$ and $[\phi/\mathsf{Q}^{\mathsf{R}x}_\mu]$ be defined:

$$[\phi/\mathsf{Q}^{\mathsf{W}x}_{\mathsf{rlx}}] = [\phi/\mathsf{Q}^x_{\mathsf{wo}}] \qquad [\phi/\mathsf{Q}^{\mathsf{R}x}_{\mathsf{rlx}}] = [\phi/\mathsf{Q}^x_{\mathsf{ro}}]$$
$$[\phi/\mathsf{Q}^{\mathsf{W}x}_{\mathsf{ra}}] = [\phi/\mathsf{Q}^x_{\mathsf{wo}}] \qquad [\phi/\mathsf{Q}^{\mathsf{R}x}_{\mathsf{rlx}}] = [\phi/\mathsf{Q}^*_{\mathsf{ro}}, \phi/\mathsf{Q}^*_{\mathsf{wo}}]$$
$$[\phi/\mathsf{Q}^{\mathsf{W}x}_{\mathsf{sc}}] = [\phi/\mathsf{Q}^x_{\mathsf{wo}}, \phi/\mathsf{Q}_{\mathsf{sc}}] \qquad [\phi/\mathsf{Q}^{\mathsf{R}x}_{\mathsf{rlx}}] = [\phi/\mathsf{Q}^*_{\mathsf{ro}}, \phi/\mathsf{Q}^*_{\mathsf{wo}}, \phi/\mathsf{Q}_{\mathsf{sc}}]$$

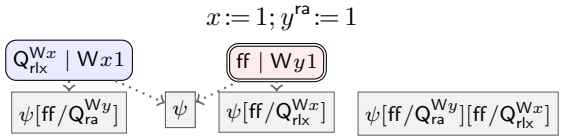
Def 34 (CO/SYNC)**.** Update Def 28 to:

S3) $\kappa(e)$ implies $\mathsf{Q}^{\mathsf{W}x}_\mu \wedge M{=}v$,
L3) $\kappa(e)$ implies $\mathsf{Q}^{\mathsf{R}x}_\mu$.
T3) $\kappa(e)$ implies $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}^*_{\mathsf{ro}}][\mathsf{tt}/\mathsf{Q}^*_{\mathsf{wo}}][\mathsf{tt}/\mathsf{Q}_{\mathsf{sc}}]$,
S5) $\tau^C(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{W}x}_\mu]$,.
L5) $\tau^C(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{R}x}_\mu]$.

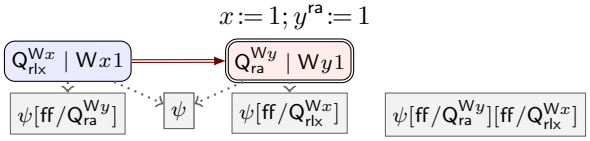
Ex 35. Def 28 enforces publication. Consider:



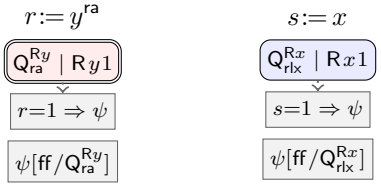
Since $\mathsf{Q}^{\mathsf{W}y}_{\mathsf{ra}}[\mathsf{ff}/\mathsf{Q}^{\mathsf{W}x}_{\mathsf{rlx}}]$ is ff, composing these without order simplifies to:



In order to get a satisfiable precondition for $(\mathsf{W}y1)$, we must introduce order:

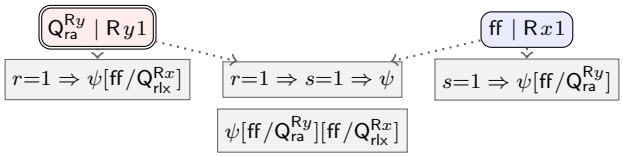


Ex 36. Def 28 enforces subscription. Consider:



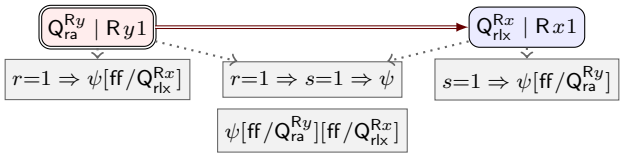
Since $\mathsf{Q}^{\mathsf{R}x}_{\mathsf{rlx}}[\mathsf{ff}/\mathsf{Q}^{\mathsf{R}y}_{\mathsf{ra}}]$ is ff, composing these without order simplifies to:

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



In order to get a satisfiable precondition for $(\mathsf{R}x1)$, we must introduce order:

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



3.3. Completed Pomsets

Def 37. A pomset with predicate transformers P is *completed* if, for every quiescence symbol s, $\tau^E(\mathsf{s})$ implies s.

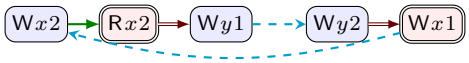
4. Efficient Implementation on ARMv8

We discuss ARM8 using *external global completion* (EGC) [1] [4, §B2.3.6] which is very close to our model.

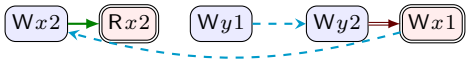
4.1. Downgraded Reads (DGR)

Ex 38. The following execution is allowed by ARM8, but disallowed by Def 34. The coherence order between the writes can be witnessed by a separate thread, which we have elided.

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



Under EGC, this is explained by dropping the order $(\mathsf{R}x2) \longrightarrow (\mathsf{W}y1)$, because $(\mathsf{R}x2)$ is fulfilled by a relaxed write in the same thread.



More generally, this can be understood as a compiler optimization that downgrades a read from ra to rlx when it can be fulfilled by a relaxed write in the same thread.

To model such *downgraded reads*, we use the uninterpreted symbols $\downarrow^x$. Load actions that requiring downgrading introduce $\downarrow^x$. Relaxed stores on x substitute true for $\downarrow^x$, whereas synchronizing stores substitute false for $\downarrow^x$.

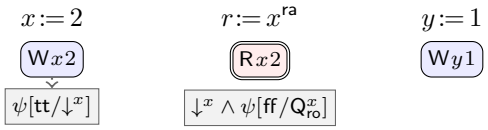
Def 39. Let $[\mathsf{ff}/\downarrow^*]$ be the substitution that performs $[\mathsf{ff}/\downarrow^x]$ for every x. Let ds^x_μ and dl^x_μ be defined:

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

Def 40 (CO/SYNC/DGR)**.** Update Def 34 to:

S4) $\tau^D(\psi)$ implies $\mathsf{ds}^x_\mu\psi[(\mathsf{Q}^x_{\mathsf{wo}} \wedge M{=}v)/\mathsf{Q}^x_{\mathsf{wo}}]$,
S5) $\tau^C(\psi)$ implies $\mathsf{ds}^x_\mu\psi[\mathsf{ff}/\mathsf{Q}^x_{\mathsf{wo}}]$.
L5) $\tau^B(\psi)$ implies $\mathsf{dl}^x_\mu\psi[\mathsf{ff}/\mathsf{Q}^x_{\mathsf{ro}}]$.

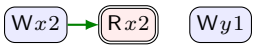
Ex 41. Revisiting Ex 38 and eliding irrelevant transformers:



Associating right:

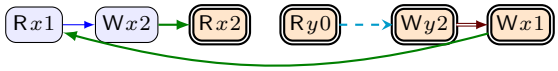
Composing, we have, as desired:

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



Ex 42. One might worry that our model is too permissive for sc access, but ARM8 itself allows some very counterintuitive results for sc access. In the following execution we elide the initializing write $(\mathsf{W}y0)$.

$$\mathtt{if}(x)\{x := 2\}; r := x^{\mathsf{sc}}; s := y^{\mathsf{sc}} \parallel y^{\mathsf{sc}} := 2; x^{\mathsf{sc}} := 1$$

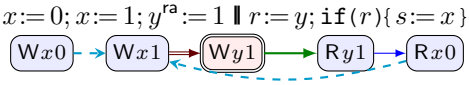


Under EGC, this is explained by dropping the order $(\mathsf{R}x2) \longrightarrow (\mathsf{R}y0)$, because $(\mathsf{R}x2)$ is fulfilled by a relaxed write in the same thread.

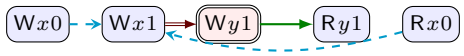
4.2. Removing Read-Read dependencies (RRD)

Ex 43. The following execution is allowed by ARM8, but disallowed by Def 34.

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



Under EGC, this is explained by dropping the order $(\mathsf{R}y1) \longrightarrow (\mathsf{R}x0)$, because ARM8 does not include control dependencies between reads in the locally-ordered-before relation.



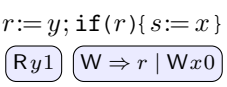
Since we do not distinguish control dependencies from other dependencies, we are forced to drop all dependencies between reads. In order to do so, we use the uninterpreted symbol W.

Def 44 (RRD)**.** Update Def 23 to:

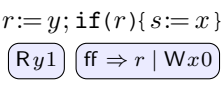
T3) $\kappa(e)$ implies $\kappa_1(e)[\mathsf{tt}/\mathsf{W}]$ if $\lambda_1(e)$ is a write,
$\kappa(e)$ implies $\kappa_1(e)[\mathsf{ff}/\mathsf{W}]$ otherwise.
L5) $\tau^C(\psi)$ implies $\mathsf{W} \Rightarrow \psi$,

Ex 45. Revisiting Ex 43 and eliding irrelevant transformers:

Composing sequentially:



Embedding the thread in thread group, we have, as desired:



4.3. Full semantics for ARM

Def 46 combines all of the features of §3–4. For completeness, we repeat L4, which is unchanged from Def 23.

Def 46 (CO/SYNC/DGR/RRD)**.** Update Def 23 to:

S3) $\kappa(e)$ implies $\mathsf{Q}_{\mu}^{\mathsf{W}x} \wedge M{=}v$,
L3) $\kappa(e)$ implies $\mathsf{Q}_{\mu}^{\mathsf{R}x}$.
T3) $\kappa(e)$ implies $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}][\mathsf{tt}/\mathsf{W}]$ if $\lambda_1(e)$ is a write,
$\kappa(e)$ implies $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}][\mathsf{ff}/\mathsf{W}]$ otherwise.

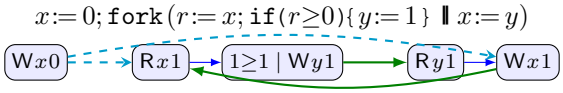
S4) $\tau^D(\psi)$ implies $(\mathsf{Q}_{\mathsf{wo}}^x \Rightarrow M{\neq}v) \wedge \mathsf{ds}_{\mu}^x\psi$,
S5) $\tau^C(\psi)$ implies $\neg\mathsf{Q}_{\mathsf{wo}}^x \wedge \mathsf{ds}_{\mu}^x\psi$.
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $\neg\mathsf{Q}_{\mathsf{rw}}^x \wedge \mathsf{dl}_{\mu}^x \wedge (\mathsf{W} \Rightarrow \psi)$.

Every ARM8 execution is allowed by Def 46. The proof of this fact is simplified by the recent characterization of ARM8 in terms of EGC [4, §B2.3.6]. Under EGC, an ARM8 execution is a linearization of per-location program order and a subset of local-order. Every such linearization is also a valid pomset under Def 46.

5. Other Features

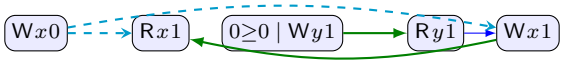
5.1. Local Invariant Reasoning (LIR)

Ex 47. JMM causality Test Case 1 [21] states the following execution should be allowed "since interthread compiler analysis could determine that x and y are always non-negative, allowing simplification of $r{\geq}0$ to true, and allowing write $y{:=}1$ to be moved early."



Under the definitions given thus far, the precondition on $(\mathsf{W}y1)$ can only be satisfied by the read of x, disallowing this execution.

In order to allow such executions, we include memory references in formula, resulting in:



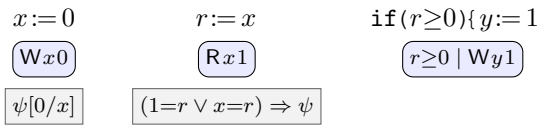
Def 48 (LIR)**.** Update Def 23 to (repeating L4 unchanged):

S4) $\tau^D(\psi)$ implies $\psi[M/x]$,
S5) $\tau^C(\psi)$ implies $\psi[M/x]$,
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $(v{=}r \vee x{=}r) \Rightarrow \psi$, when $E \neq \emptyset$,
L6) $\tau^B(\psi)$ implies ψ, $E = \emptyset$.

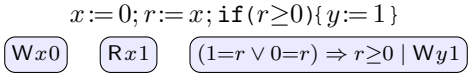
L5 introduces memory references. It states that to be independent of the read, we establish both $\psi[v/r]$ and $\psi[x/r]$. If a precondition holds in both circumstances, S5 allows a local write to satisfy the precondition without introducing dependence.

One reading of L5 is that when satisfying a precondition ϕ it is safe to ignore a read as long as ϕ is compatible with both the value of the read and the value of the preceding local write. This begs the question: what value must ϕ be compatible with in the case that the pomset is empty? In this case, there is no value v to check! Therefore the best we can do is to emulate skip, as in L6. In order to eventually arrive at a top-level pomset, this means that subsequent code must be independent of r.

Ex 49. Revisiting Ex 47 and eliding irrelevant transformers:



Composing:

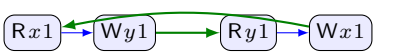


The precondition of $(\mathsf{W}y1)$ is a tautology, as required.

5.2. Register Recycling (ALPHA)

The semantics considered thus far assume that each register is assigned at most once in a program. We relax this by renaming.

Ex 50. JMM causality Test Case 2 [21] states the following execution should be allowed "since redundant read elimination could result in simplification of $r{=}s$ to true, allowing $y{:=}1$ to be moved early."



This execution is not allowed under Def 48, since the precondition of $(\mathsf{W}y1)$ in the independent case is

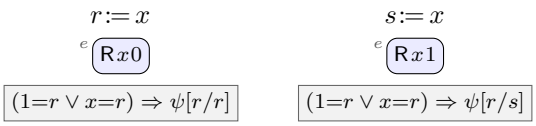
$$(r{=}1 \vee r{=}x) \Rightarrow (s{=}1 \vee s{=}r) \Rightarrow (r{=}s),$$

which is not a tautology. Our solution is to rename registers using the set $\mathcal{S}_{\mathcal{E}} = \{s_e \mid e \in \mathcal{E}\}$, which are banned from source programs, as per §2.1. This allows us to resolve nondeterminism in loads when merging, resulting in:

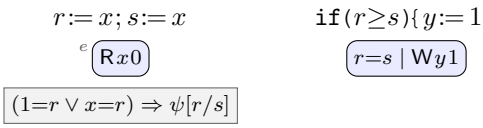
Def 51 (ALPHA)**.** Update Def 23 to:

L4) $\tau^D(\psi)$ implies $v{=}s_e \Rightarrow \psi[s_e/r]$,
L5) $(\forall s)\ \tau^C(\psi)$ implies $\psi[s/r]$.

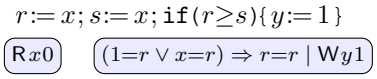
Ex 52. Revisiting Ex 50, eliding irrelevant transformers and choosing $s_e = r$:



Coalescing and composing:



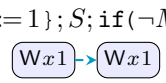
Composing:



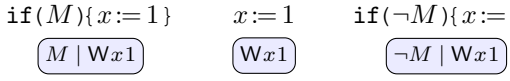
The precondition of $(\mathsf{W}y1)$ is a tautology, as required.

5.3. If-Closure (IF)

Ex 53. If $S = (x{:=}1)$, then Def 23 does *not* allow:

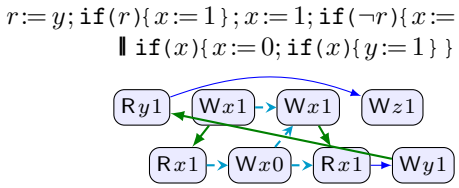


However, if $S = (\texttt{if}(\neg M)\{x{:=}1\};\texttt{if}(M)\{x{:=}1\})$, then it *does* allow the execution. Looking at the initial program:



The difficulty is that the middle action can coalesce either with the right action, or the left, but not both. Thus, we are stuck with some non-tautological precondition. Our solution is to allow a pomset to contain many events for a single action, as long as the events have disjoint preconditions.

This is not simply a theoretical question; it is observable.



Def 54 (ALPHA/IF)**.** Update Def 23 to:

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

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

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

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

Ex 55. Revisiting Ex 53, we can split the middle command:

Coalescing events gives the desired result.

5.4. Address Calculation (ADDR)

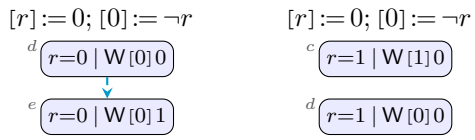
Def 56 (ADDR)**.** Update Def 23 to existentially quantify over ℓ in *STORE* and *LOAD*:

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

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

S4) $(\forall k)\ \tau^D(\psi)$ implies $L{=}k \Rightarrow \psi$,
S5) $(\forall k)\ \tau^C(\psi)$ implies $L{=}k \Rightarrow \psi$,
L4) $(\forall k)\ \tau^D(\psi)$ implies $L{=}k \Rightarrow v{=}r \Rightarrow \psi$,
L5) $(\forall k)\ \tau^C(\psi)$ implies $L{=}k \Rightarrow \psi$.

Ex 57. punning badly: Consider that $[\![[r] := 0; [0] := \neg r]\!]$ includes both of the following pomsets



Thus, the disjunction closure also includes both of the following:

In this example, the d events that coalesce come from inconsistent executions. This is possible because the d events originate from different commands.

6. Discussion

6.1. Relation to Traditional Predicate Transformers

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

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

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

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

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

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

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

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

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

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

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

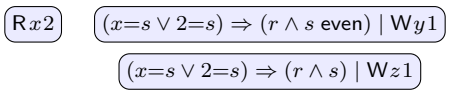
Prop 2. *If $P \in [\![S]\!]$ is top-level and quiescent then $\tau^E(\phi)$ implies $wp_S(\phi)$.*

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

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

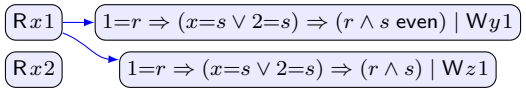
[I have a note: TC1: Track local state ???]

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

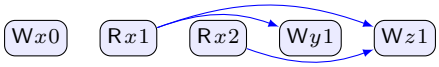


Without substitution:

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

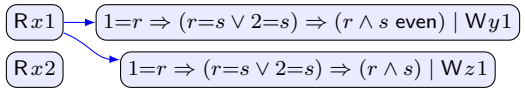


Prepending $x{:=}0$



With the substitution $[r/x]$:

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



Prepending $x{:=}0$

6.3. Fork-Join

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

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

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

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

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

Def 58. A *pomset with preconditions and termination* is a pomset with preconditions together with a predicate ✓.

Def 59.
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 12,
9) ✓ implies $\checkmark_1 \wedge \checkmark_2$.

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

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

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

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

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

We can then encode coherence as follows.

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

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

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

$$\mathsf{Q}^{\mathsf{W}}_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{sc}} \qquad \mathsf{Q}^{\mathsf{W}}_{\mathsf{ra}} = \mathsf{Q}_{\mathsf{ra}} \qquad \mathsf{Q}^{\mathsf{W}}_{\mathsf{rlx}} = \mathsf{Q}^{x}_{\mathsf{rw}}$$
$$\mathsf{Q}^{\mathsf{R}}_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{sc}} \qquad \mathsf{Q}^{\mathsf{R}}_{\mathsf{ra}} = \mathsf{Q}^{x}_{\mathsf{wo}} \qquad \mathsf{Q}^{\mathsf{R}}_{\mathsf{rlx}} = \mathsf{Q}^{x}_{\mathsf{wo}}$$
$$\mathsf{ds}^{x}_{\mathsf{sc}}\psi = \psi[\mathsf{ff}/{\downarrow^*}] \qquad \mathsf{ds}^{x}_{\mathsf{ra}}\psi = \psi[\mathsf{ff}/{\downarrow^*}] \qquad \mathsf{ds}^{x}_{\mathsf{rlx}}\psi = \psi[\mathsf{tt}/{\downarrow^x}]$$
$$\mathsf{dl}^{x}_{\mathsf{sc}} = {\downarrow^x} \qquad \mathsf{dl}^{x}_{\mathsf{ra}} = {\downarrow^x} \qquad \mathsf{dl}^{x}_{\mathsf{rlx}} = \mathsf{tt}$$

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

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

S5) $\tau^{\emptyset}(\psi)$ implies $\neg \mathsf{Q}_{\mathsf{ra}} \wedge \mathsf{ds}^{x}_{\mu}\psi[M/x]$

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

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

6.4. Must Allow Inconsistent Preconditions

See examples in §5.3.

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

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

6.5. Skolemization

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

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

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

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

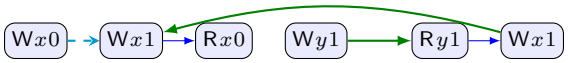
[Skolemization ensures disjunction closure, which is necessary for associativity. Show example.]

6.6. Reads Update Local State

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

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

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



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

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

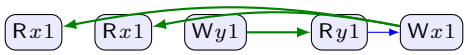
6.7. Parallel Composition

In [12, §2.4], parallel composition is defined allowing coalescing of events. Here we have forbidden coalescing. This difference appears to be arbitrary. In [12], however, there is a mistake in the handling of termination actions. The predicates should be joined using $\wedge$, not $\vee$.

6.8. Redundant Read Elimination

Requires indexing to resolve nondeterminism.

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



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

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

Combining the transformers, we have

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

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

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

which is not a tautology.

Same problem occurs [12], where we have:

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

Combining the transformers, we have

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

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

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

which is not a tautology.

The semantics here allows this by coalescing:

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

6.9. Redundant Read Elimination

In [12, §2.6] the semantics of read is defined as follows:

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

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

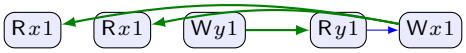
(P4C) If d reads v from x then either $e = d$ or $\kappa'(e)$ implies $\kappa(e)[v/x]$.
(P5A) If d reads and e writes then either $\kappa'(e)$ implies $\kappa(e)$ or $d \leq' e$.

We have discovered two issues with this definition.

The first issue concerns the substitution $[x/r]$. It should be $[r/x]$. We noticed this error while developing the alternative characterization presented here. The error causes redundant read elimination to fail in [12]. 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 [12]: $(\mathsf{W}y1)$ has precondition $r{=}s$ in $[\![\texttt{if}(r{=}s)\{y{:=}1\}]\!]$. Given the lack of order in the execution, the precondition of $(\mathsf{W}y1)$ must entail $r{=}1 \wedge r{=}x$ in $[\![s{:=}x;\texttt{if}(r{=}s)\{y{:=}1\}]\!]$. P4C imposes $r{=}1$, and P5A imposes $r{=}x$. Adding the second read, the precondition of $(\mathsf{W}y1)$ must entail both $1{=}1\wedge 1{=}x$ and also $x{=}1 \wedge x{=}x$. This can be simplified to $x{=}1$. This leaves a requirement that must be satisfied by a preceding write. Since the preceding write is the initialization to 0, the requirement cannot be satisfied, and the execution is impossible.[1]

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

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

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

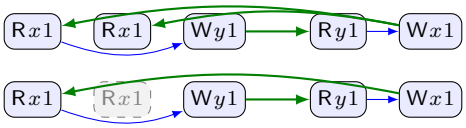
We can then reason with TC2 as follows: $(\mathsf{W}y1)$ has precondition $r{=}s$ in $[\![\texttt{if}(r{=}s)\{y{:=}1\}]\!]$. To avoid introducing order in the execution, the precondition of $(\mathsf{W}y1)$ must entail $r{=}1\wedge r{=}s$ in $[\![s{:=}x;\texttt{if}(r{=}s)\{y{:=}1\}]\!]$. P4C imposes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

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

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

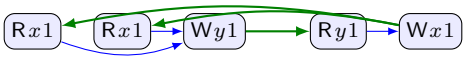
It is interesting to note that the substitution is $[x/r]$ on eliminated reads, and $[r/x]$ on non-eliminated reads. Intuitively, the subsequent value of x is fixed by an explicit read, but not for an eliminated read. In the latter case, the value is fixed by some preceding action. The preceding action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

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



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

1. In [12] we ignore the middle terms, mistakenly simplifying this to $1{=}1 \wedge x{=}x$. Correcting the error, the attempted execution is:

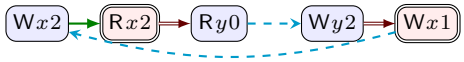


6.10. Internal Acquiring Reads

Our solution allows executions that are not allowed under ARM8 since we do not insist that the local relaxed write is actually read from. This may seem counterintuitive, but we don't see a local way to be more precise.

The second issue concerns acquiring reads. Shortly after publication, Podkopaev [20] noticed a shortcoming of the implementation on ARM8 in [12, §7]. The proof given there assumes that all internal reads can be dropped. However, this is not the case for acquiring reds. For example, [12] disallows the following execution, which is allowed by ARM8 and TSO.

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

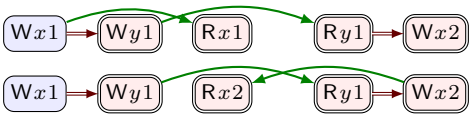


The solution we have adopted is to allow an acquiring read to be downgraded to a relaxed read when it is preceded (sequentially) by a relaxed write that could fulfill it. Backporting this solution to [12] requires that we add access predicates to the logic and allow

6.11. Triangular Races

The notion of data-race is incorrect in [12].

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



Bug is in [8, Lemma A.4]. It assumes that $(\mathsf{R}x1)$ and $(\mathsf{W}x2)$ are racing in the first execution because they are not ordered by happens-before. But this is false since neither is plain.

In addition, the ARM8 implementation result given here does not rely on read elimination. Instead we use a recent alternative characterization of ARM8 [1, 4, 3].

7. Outro

References

[1] J. Alglave. This commit adds three alternative formulations of the arm model, both for non-mixed and mixed size accesses. https://github.com/herd/herdtools7/commit/685ee4, June 2020.

[2] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014. ISSN 0164-0925. doi: 10.1145/2627752. URL http://doi.acm.org/10.1145/2627752.

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

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

[5] S. D. Brookes. Full abstraction for a shared-variable parallel language. *Inf. Comput.*, 127(2):145–163, 1996. doi: 10.1006/inco.1996.0056. URL https://doi.org/10.1006/inco.1996.0056.

[6] S. Chakraborty and V. Vafeiadis. Grounding thin-air reads with event structures. *PACMPL*, 3(POPL):70:1–70:28, 2019. doi: 10.1145/3290383. URL https://doi.org/10.1145/3290383.

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

[8] 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.

[9] J. L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61(2):199–224, 1988. ISSN 0304-3975. doi: 10.1016/0304-3975(88)90124-7. URL http://www.sciencedirect.com/science/article/pii/0304397588901247.

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

[11] R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 307–326, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11956-5, 978-3-642-11956-9. doi: 10.1007/978-3-642-11957-6_17. URL http://dx.doi.org/10.1007/978-3-642-11957-6_17.

[12] 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.

[13] J. Kang, C. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In G. Castagna and A. D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 175–189. ACM, 2017. URL http://dl.acm.org/citation.cfm?id=3009850.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979. ISSN 0018-9340. doi: 10.1109/TC.1979.1675439. URL https://doi.org/10.1109/TC.1979.1675439.

[15] L. Liu, T. Millstein, and M. Musuvathi. Accelerating sequential consistency for java with speculative compilation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 16–30, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314611. URL http://doi.acm.org/10.1145/3314221.3314611.

[16] J. Manson, W. Pugh, and S. V. Adve. The java memory model. *SIGPLAN Not.*, 40(1):378–391, Jan. 2005. ISSN 0362-1340. doi: 10.1145/1047659.1040336. URL http://doi.acm.org/10.1145/1047659.1040336.

[17] A. W. Mazurkiewicz. Introduction to trace theory. In V. Diekert and G. Rozenberg, editors, *The Book of Traces*, pages 3–41. World Scientific, 1995. doi: 10.1142/9789814261456_0001. URL https://doi.org/10.1142/9789814261456_0001.

[18] J. Pichon-Pharabod and P. Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 622–633, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837616. URL http://doi.acm.org/10.1145/2837614.2837616.

[19] G. D. Plotkin and V. R. Pratt. Teams can see pomsets. In D. A. Peled, V. R. Pratt, and G. J. Holzmann, editors, *Partial Order Methods in Verification, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, July 24-26, 1996*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 117–128. DIMACS/AMS, 1996. doi: 10.1090/dimacs/029/07. URL https://doi.org/10.1090/dimacs/029/07.

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

[21] W. Pugh. Causality test cases, 2004. URL https://perma.cc/PJT9-XS8Z.