Sequential Composition for Relaxed Memory: Pomsets with Predicate Transformers

Alan Jeffrey* and James Riely†
*Roblox
†DePaul University

***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 paper show how they can be combined.**

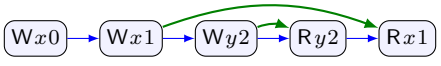
1. Introduction

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:

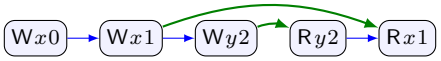
$$x{:=}0; x{:=}1; y{:=}2; r{:=}y; s{:=}x$$



(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, leading to a *sequentially consistent* semantics [14], in which *program order* inside a thread implies a total *causal order* between read and write events, for example:

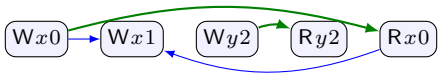
$$x{:=}0; x{:=}1; y{:=}2 \parallel r{:=}y; s{:=}x$$



Unfortunately, this model does not compile efficiently to commodity hardware, resulting in a 37–73% increase in CPU time on ARM [15] and, hence, in 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.

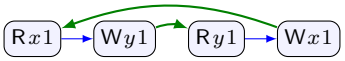
This has led to *relaxed memory models*, in which the requirement of sequential consistency is weakened to only apply *per-location* and not globally over the whole program. This allows executions which are inconsistent with program order, such as:

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



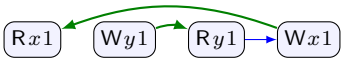
In such models, the causal order between events is important, and includes control and data dependencies, to avoid paradoxical "out of thin air" examples such as:

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



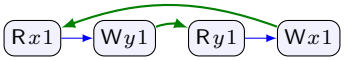
This candidate execution forms a cycle in causal order, so is disallowed, but this depends crucially on the control dependency from $(\mathsf{R}x1)$ to $(\mathsf{W}y1)$, and the data dependency from $(\mathsf{R}y1)$ to $(\mathsf{W}x1)$. If either is missing, then this execution is acyclic and hence allowed.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

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

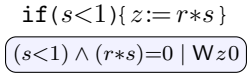


An optimizing compiler might lift the assignment $y{:=}1$ out of the conditional, thus removing the control dependency.

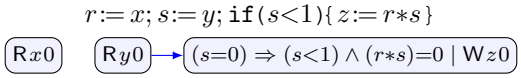
Prominent solutions to the problem of dependency calculation include:

- *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, 5],
- *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].

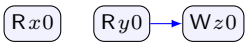
In this paper, we will focus on logical models, as those are compositional, and align well with existing models of sequential composition. The heart of the model of [12] is to add logical preconditions to events, which are introduced by store actions (modeling data dependencies) and conditionals (modeling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

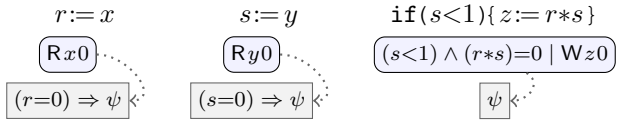
1.2. Sequential composition

Our approach follows that of weakest precondition semantics of Dijkstra [6], 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.

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

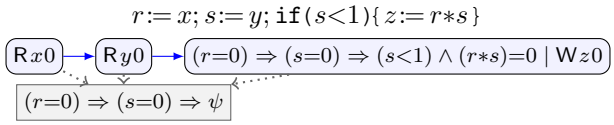
The rule we are most interested in is the one for sequential composition, which maps sequential composition of programs to function composition of predicate transformers.

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:



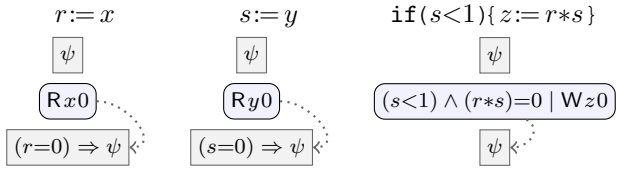
In the rightmost program above, the write to z affects the shared store, not the local state of the thread, therefore we assign it the identity transformer.

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:



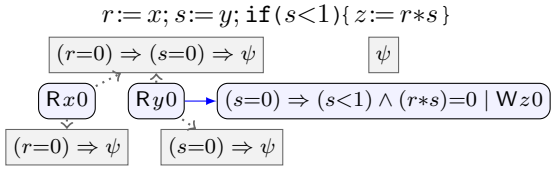
This model works for the sequentially consistent case, but needs to be weakened for the relaxed case. The key observation of this paper is that rather than working with one predicate transformer, we should work with a *family* of predicate transformers, indexed by sets of events.

For example, for single-event pomsets, there are two predicate transformers, since there are two subsets of any one-element set. The *independent* transformer is indexed by the empty set, whereas the *dependent* transformer is indexed by the singleton. We visualize this by including more than one transformed predicate, with an edge leading to the dependent one. For example:

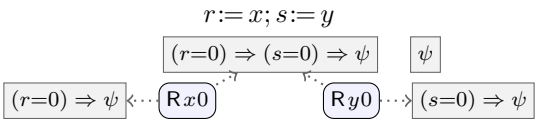


The model of sequential composition then picks which predicate transformer to apply to an event's precondition by picking the one indexed by all the events before it in causal order.

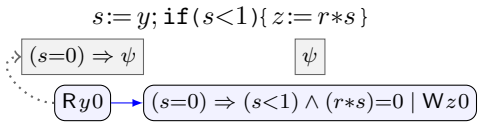
For example, we can recover the expected semantics for the above example by choosing the predicate transformer which is independent of $(\mathsf{R}x0)$ but dependent on $(\mathsf{R}y0)$, which is the transformer which maps ψ to $(s{=}0) \Rightarrow \psi$.



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 [6].

- §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, and
- §5 show how to address common litmus tests.

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 as in [7, 8]. 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, parallel composition, 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 quiescence symbols Q_{sc}, $\mathsf{Q}^x_{\mathsf{ro}}$, $\mathsf{Q}^x_{\mathsf{wo}}$ (used in §3), the downgrade symbols $\downarrow^x$ (used in §4.1), and the write symbol W (used in §4.2),
- 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 $\mathcal{S}_\mathcal{E}$ do not appear in programs: $S[N/s_e] = S$.

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. 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 $(\forall e \in E)(\forall x \in \mathcal{X})$.

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 $(\exists E_1)\ (\exists P_2 \in \mathcal{P}_2)$

1) $E = (E_1 \cup E_2)$,
2) if $d, e \in E_1$ then $d = e$,
3) if $e \in E_1$ 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:

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

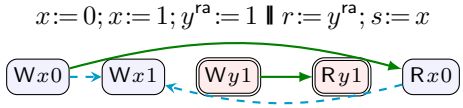
In this semantics, both `skip` and 0 map to the empty pomset. Parallel composition is disjoint union, inheriting labeling and order from the two sides. Prefixing may add a new action (on the left) to an existing pomset (on the right), inheriting labeling and order from the right.

It is worth noting that if $\leftrightarrow$ is taken to be the empty relation, then fulfilled pomsets of Def 1 correspond to sequentially consistent executions up to mumbling [4].

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 32.

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,
- $e \longrightarrow d$ 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*,
- $e \longrightarrow d$ arises from control/data/address *dependency*,
- $e \Longrightarrow d$ 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].

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

Let PomPre be the set of all pomsets with preconditions. The function $TOP : 2^{\mathsf{PomPre}} \to 2^{\mathsf{Pom}}$ embeds sets of pomsets with preconditions into sets of pomsets. It also substitutes formulae for quiescence and write symbols, for use in §3-4. In these "top-level" pomsets, every read is fulfilled and every precondition is a tautology.

Def 11. Let $[\phi/\mathsf{Q}]$ substitute all quiescence symbols by ϕ.

Def 12. If $P \in TOP(\mathcal{P})$ 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 write, $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}][\mathsf{tt}/\mathsf{W}]$ is a tautology,
5) if $\lambda_1(e)$ is a read, $\kappa_1(e)[\mathsf{tt}/\mathsf{Q}][\mathsf{ff}/\mathsf{W}]$ is a tautology and e is fulfilled (Def 2).

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 $(r{:=}x^\mu; S)$.

Def 13. If $P \in NIL$ 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 $P \in IF(\phi, \mathcal{P}_1, \mathcal{P}_2)$ 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 $P \in ST(x, M, \mathcal{P}_2)$ 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 $M{=}v \vee \kappa_2(e)$.

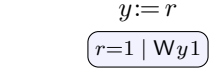
If $P \in LD(r, x, \mathcal{P}_2)$ then $(\exists P_2 \in \mathcal{P}_2)\ (\exists v \in \mathcal{V})$

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 $\kappa_2(e)$.

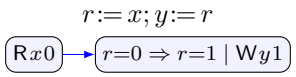
Def 14. For a language fragment, the semantics is:

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

Ex 15. 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 $v = 1$:

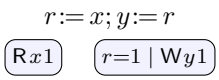


The value chosen for the read may be different from that chosen for the write:

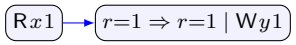


In this case, the pomset's preconditions depend on a bound register, so cannot contribute to a top-level pomset.

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,

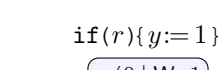


and the dependent case:



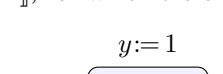
Since $r{=}1 \Rightarrow r{=}1$ is a tautology, this can be part of a top-level pomset.

Ex 16. 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:

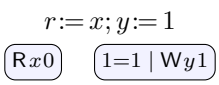


The rest of the reasoning is the same as Ex 15.

Ex 17. A simple example of an independency is a pomset $P \in [\![r{:=}x; y{:=}1]\!]$, for which there must be:



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



Ex 18. 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:

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:

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

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

2.4. 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.

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

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

As discussed in §1, 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 $d < e$.

Def 20. 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 $\tau^D(\psi)$.

Def 21. A *pomset with predicate transformers* is a pomset with preconditions (Def 13), together with a family of predicate transformers for E.

Let $\mathsf{PomTrans}$ be the set of all pomsets with predicate transformers. We can covert between pomsets with preconditions and pomsets with predicate transformers. In one direction, $\mathit{THRD} : 2^{\mathsf{PomTrans}} \to 2^{\mathsf{PomPre}}$ drops predicate transformers, and in the other, $\mathit{FORK} : 2^{\mathsf{PomPre}} \to 2^{\mathsf{PomTrans}}$ adopts the identity transformer.

Def 22. If $P \in \mathit{THRD}(\mathcal{P})$ 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) $\kappa(e)$ implies $\kappa_1(e)$.

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

1) $E = E_1$,
2) $\lambda(e) = \lambda_1(e)$,
3) if $d \leq_1 e$ then $d \leq e$,
4) $\kappa(e)$ implies $\kappa_1(e)$,
5) $\tau^D(\psi)$ implies ψ.

We model thread groups as sets of pomsets with preconditions, as in §2.3.

Def 23. Adopting *NIL* and $\|$ from Def 13, the semantics of thread groups is:

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

We model commands by adding predicate transformers to Def 13, in order to calculate weakest preconditions.

Def 24. 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 13,
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 4 (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 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 25. The semantics of commands is:

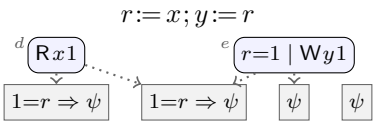
$$[\![\texttt{if}(M)\{\,S_1\,\}\,\texttt{else}\,\{\,S_2\,\}]\!] = \mathit{IF}(M{\neq}0,\, [\![S_1]\!],\, [\![S_2]\!])$$
$$[\![x^\mu{:=}\,M]\!] = \mathit{STORE}(x,\, M,\, \mu) \qquad [\![\texttt{abort}]\!] = \mathit{ABORT}$$
$$[\![r{:=}\,x^\mu]\!] = \mathit{LOAD}(r,\, x,\, \mu) \qquad [\![\texttt{skip}]\!]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

Most of these definitions are straightforward adaptations of §2.3, 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.

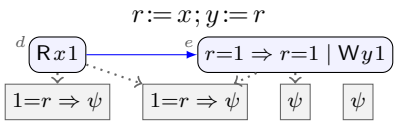
Ex 26. For read to write dependency, consider:

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:



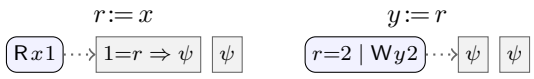
This pomset's preconditions depend on a bound register, so cannot contribute to a top-level pomset.

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 $(r{=}1 \Rightarrow \psi)$, this gives that $\kappa(e)$ is $(r{=}1 \Rightarrow \kappa_2(e))$, which is $(r{=}1 \Rightarrow r{=}1)$. This gives the pomsaet with predicate transformers:

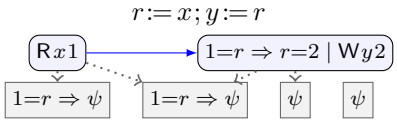


This pomset's preconditions do not depend on a bound register, so can contribute to a top-level pomset.

Ex 27. 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:



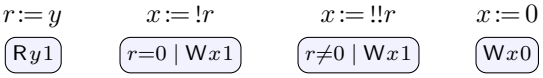
Ex 28. The predicate transformer we have chosen for L4 is different from the one used traditionally, which is written using substitution. Substitution is also used in [12]. Attempting to write the predicate transformers in this style we have:

L4) $\tau^D(\psi)$ implies $\psi[v/r]$,
L5) $\tau^C(\psi)$ implies $(\forall r)\psi$.

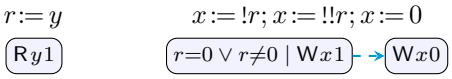
This phrasing of L5 says that ψ must be independent of r in order to appear in a top-level pomset. This choice for L5 is forced by Def 20, which states that the predicate transformer for a small subset of E must imply the transformer for a larger subset.

Sadly, this definition fails associativity.

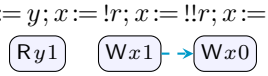
Consider the following, eliding transformers:



Associating to the right and merging:

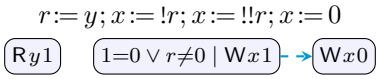


The precondition of $(\mathsf{W}x1)$ is a tautology, thus we have:



If, instead, we associate to the left:

Sequencing and merging:



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 24 in Agda. The proof requires that predicate transformers distribute through disjunction (Def 19). Since universal quantification does not distribute through disjunction, the attempt to define predicate transformers using substitution fails (in particular for L5.)

2.5. The Road Ahead

The final semantic functions for load, store, and thread initialization are given in Fig 1, at the end of the paper. In §3–5, we explain this definition by looking at its constituent parts, building on Def 24. In §3, we add *quiescence*, which encodes coherence, release-acquire access, and SC access. In §4, we add peculiarities that are necessary for efficient implementation on ARM8. In §5, 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 24, 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.

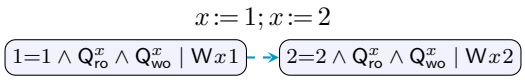
3. Quiescence

We introduce *quiescence*, which captures *coherence* and *synchronized access*. 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

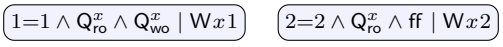
these logical symbols can be used to capture coherence and synchronization. This illustrates a feature of our model, which is that many features of weak memory can be cautured in the logic, not in the pomset model itself.

3.1. Coherence (CO)

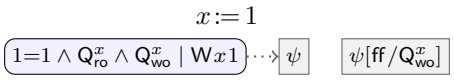
In the logic, the quiescence symbols are just uninterpreted formula, but the semantics uses them as preconditions, to ensure appropriate causal order. For example, *write-write coherence* enforces order between writes to the same location in the same thread. We model this by adding the precondition $(\mathsf{Q}^x_{\mathsf{ro}}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



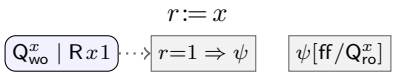
These symbols are left alone in the dependent case, but in the independent case we substitute ff for $\mathsf{Q}^x_{\mathsf{wo}}$:



This substitution is part of the predicate transformer:

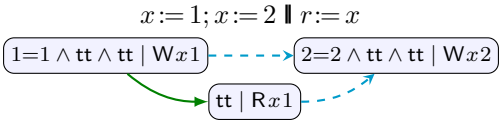


We treat read-write and write-read coherence similarly:



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 quiescence symbol with tt, so at top level there are no remaining quiescence symbols, for example:

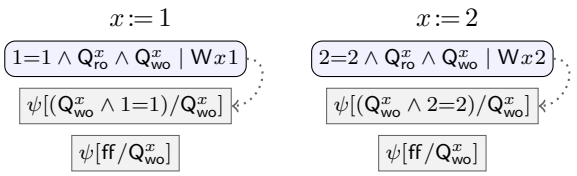


Def 29 (CO)**.** Update Def 24:

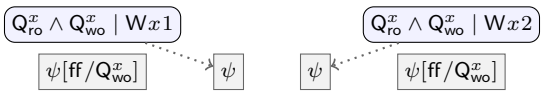
S3) $\kappa(e)$ implies $\mathsf{Q}^x_{\mathsf{ro}} \wedge \mathsf{Q}^x_{\mathsf{wo}} \wedge M{=}v$,
L3) $\kappa(e)$ implies $\mathsf{Q}^x_{\mathsf{wo}}$,

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 $\psi[\mathsf{ff}/\mathsf{Q}^x_{\mathsf{ro}}]$.

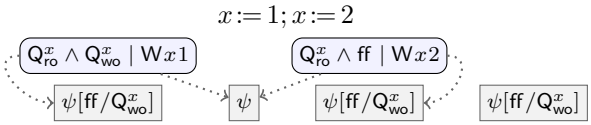
Ex 30. Def 29 enforces coherence. Consider:



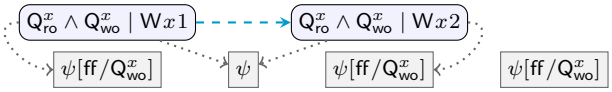
Simplifying, we have:



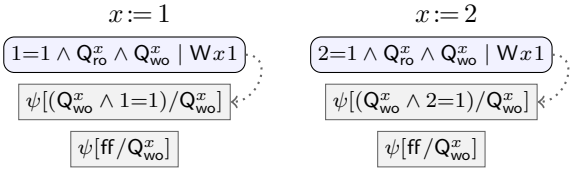
If we attempt to put these together unordered, the precondition of $(\mathsf{W}x2)$ becomes unsatisfiable:



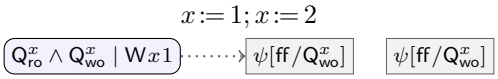
In order to get a satisfiable precondition for $(\mathsf{W}x2)$, we must introduce order:



Ex 31. 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.



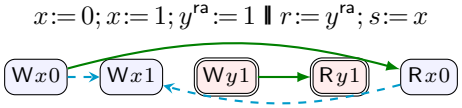
Merging the actions, since $2{=}1$ is unsatisfiable, we have:



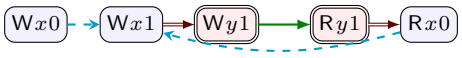
This is what we would hope: that the program $x:=1; x:=2$ should only be quiescent if there is a $(\mathsf{W}x2)$ event.

3.2. Synchronized Access (SYNC)

Ex 32. The publication idiom requires that we disallow the execution below, which is allowed by Def 29.



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



Def 33. Let $\mathsf{Q}^*_{\mathsf{ro}} = \bigwedge_y \mathsf{Q}^y_{\mathsf{ro}}$, and similarly for $\mathsf{Q}^*_{\mathsf{wo}}$. Let formulae $\mathsf{Q}^{\mathsf{S}x}_\mu$ and $\mathsf{Q}^{\mathsf{L}x}_\mu$ be defined:

$$\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}} = \mathsf{Q}^x_{\mathsf{ro}} \wedge \mathsf{Q}^x_{\mathsf{wo}} \qquad \mathsf{Q}^{\mathsf{L}x}_{\mathsf{rlx}} = \mathsf{Q}^x_{\mathsf{wo}}$$
$$\mathsf{Q}^{\mathsf{S}x}_{\mathsf{ra}} = \mathsf{Q}^*_{\mathsf{ro}} \wedge \mathsf{Q}^*_{\mathsf{wo}} \qquad \mathsf{Q}^{\mathsf{L}x}_{\mathsf{ra}} = \mathsf{Q}^x_{\mathsf{wo}}$$
$$\mathsf{Q}^{\mathsf{S}x}_{\mathsf{sc}} = \mathsf{Q}^*_{\mathsf{ro}} \wedge \mathsf{Q}^*_{\mathsf{wo}} \wedge \mathsf{Q}_{\mathsf{sc}} \qquad \mathsf{Q}^{\mathsf{L}x}_{\mathsf{sc}} = \mathsf{Q}^x_{\mathsf{wo}} \wedge \mathsf{Q}_{\mathsf{sc}}$$

Let $[\phi/\mathsf{Q}^*_{\mathsf{ro}}]$ substitute ϕ for every $\mathsf{Q}^y_{\mathsf{ro}}$, and similarly for $\mathsf{Q}^*_{\mathsf{wo}}$. Let substitutions $[\phi/\mathsf{Q}^{\mathsf{S}x}_\mu]$ and $[\phi/\mathsf{Q}^{\mathsf{L}x}_\mu]$ be defined:

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

Def 34 (CO/SYNC)**.** Update Def 24 to:

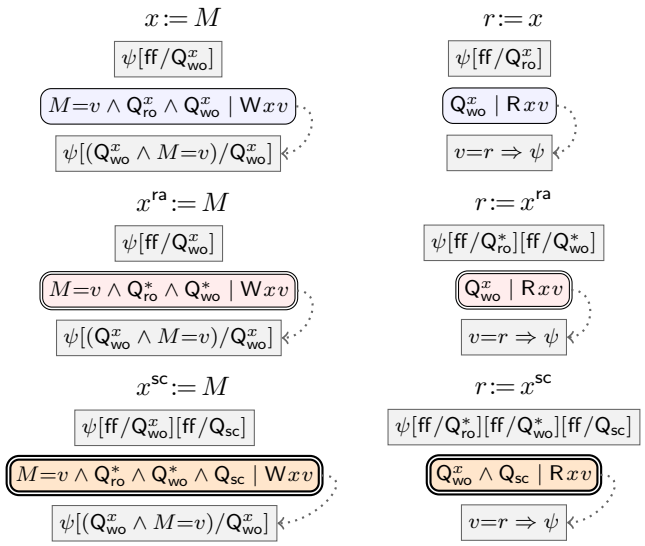
S3) $\kappa(e)$ implies $M{=}v \wedge \mathsf{Q}^{\mathsf{S}x}_{\mu}$,
L3) $\kappa(e)$ implies $\mathsf{Q}^{\mathsf{L}x}_{\mu}$,

S4) $\tau^{D}(\psi)$ implies $\psi[(\mathsf{Q}^{x}_{\mathsf{wo}} \wedge M{=}v)/\mathsf{Q}^{x}_{\mathsf{wo}}]$,
S5) $\tau^{C}(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_{\mu}]$,
L4) $\tau^{D}(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^{C}(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_{\mu}]$.

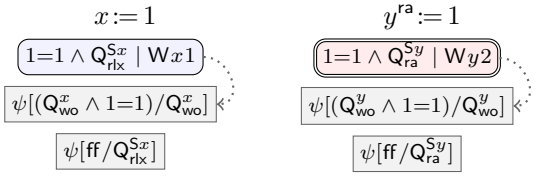
The quiescence formulae indicate what must precede an event. For example, all preceding accesses must be ordered before a releasing write, whereas only writes on x must be ordered before a releasing read on x.

The quiescence substitutions update quiescence symbols in subsequent code. For subsequent independent code, *S5* and *L5* substitute false. In top-level pomsets, *TOP* substitutes true (Def 12). For example, we substitute ff for $\mathsf{Q}^{\mathsf{S}x}_{\mathsf{ra}}$ in the independent case for a releasing write; this ensures that subsequent writes to x follow the releasing write in top-level pomsets. Similarly, we substitute ff for $\mathsf{Q}^{\mathsf{L}x}_{\mathsf{ra}}$ in the independent case for an acquiring write; this ensures that all subsequent accesses follow the acquiring read in top-level pomsets.

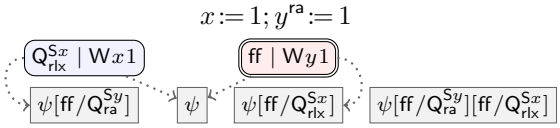
Ex 35. The following pomsets show the effect of quiescence for each access mode.



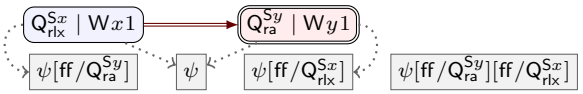
Ex 36. Def 29 enforces publication. Consider:



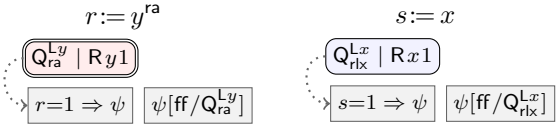
Since $\mathsf{Q}^{\mathsf{S}y}_{\mathsf{ra}}[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_{\mathsf{rlx}}]$ is ff, composing these without order simplifies to:



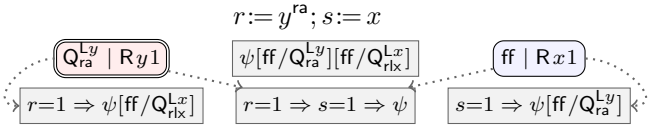
In order to get a satisfiable precondition for $(\mathsf{W}y1)$, we must introduce order:



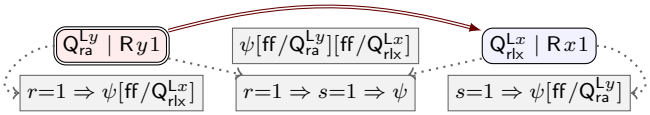
Ex 37. Def 29 enforces subscription. Consider:



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



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

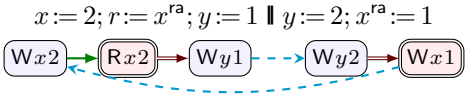


4. Efficient Implementation on ARMv8

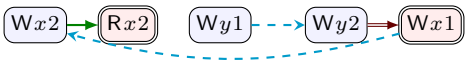
We discuss ARM8 using *external global completion* (EGC) [1] [3, §B2.3.6] which is very close to our model.

4.1. Downgraded Reads (DGR)

Ex 38. The following example is from Podkopaev [20]. It 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.



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 $\downarrow^{x}$.

Def 39. Let $[\phi/\downarrow^{*}]$ substitute ϕ for every $\downarrow^{y}$.
Let formula $\downarrow^{x}_{\mu}$ and substitution $[\mu/\downarrow^{x}]$ be defined:

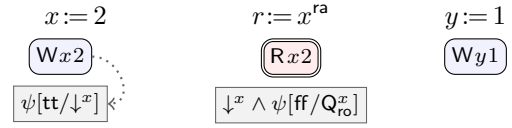
$$\downarrow^{x}_{\mathsf{rlx}} = \mathsf{tt} \qquad [\mathsf{rlx}/\downarrow^{x}] = [\mathsf{tt}/\downarrow^{x}]$$
$$\downarrow^{x}_{\mathsf{ra}} = \downarrow^{x} \qquad [\mathsf{ra}/\downarrow^{x}] = [\mathsf{ff}/\downarrow^{*}]$$
$$\downarrow^{x}_{\mathsf{sc}} = \downarrow^{x} \qquad [\mathsf{sc}/\downarrow^{x}] = [\mathsf{ff}/\downarrow^{*}]$$

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

S4) $\tau^D(\psi)$ implies $\psi[\mu/{\downarrow^x}][(\mathsf{Q}^x_{\mathsf{wo}} \wedge M{=}v)/\mathsf{Q}^x_{\mathsf{wo}}]$,
S5) $\tau^C(\psi)$ implies $\psi[\mu/{\downarrow^x}][\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_\mu]$,
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies ${\downarrow^x_\mu} \wedge \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_\mu]$.

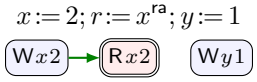
Load actions that require downgrading introduce $\downarrow^x$. Relaxed stores on x substitute true for $\downarrow^x$, whereas synchronizing stores substitute false for $\downarrow^x$.

Ex 41. Revisiting Ex 38 and eliding irrelevant transformers:

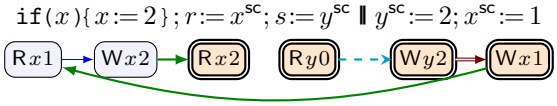


Associating right:

Composing, we have, as desired:



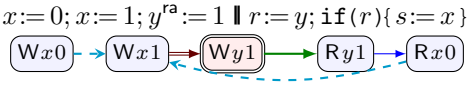
Ex 42. One might worry that our model is too permissive for sc access, but ARM8 itself allows some very counterintuitive results for sc access. In the following execution we elide the initializing write $(\mathsf{W}y0)$.



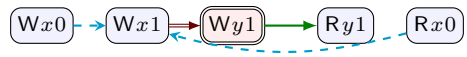
Under EGC, this is explained by dropping the order $(\mathsf{R}x2) \longrightarrow (\mathsf{R}y0)$, because $(\mathsf{R}x2)$ is fulfilled by a relaxed write in the same thread.

4.2. Removing Read-Read dependencies (RRD)

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



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.

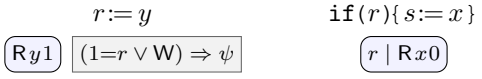
Def 44 (RRD)**.** Update Def 24 to:

L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $(v{=}r \vee \mathsf{W}) \Rightarrow \psi$, when $E \neq \emptyset$,
L6) $\tau^B(\psi)$ implies ψ, when $E = \emptyset$.

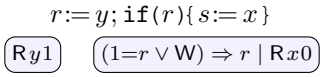
L4 is the *dependent* case, L5 the *independent* case, and L6 the *empty* case. Recall the substitutions performed by *TOP* (Def 12). For writes, L5 is the same as L6 (since W is tt). For reads, L5 is the same as L4 (since W is ff).

One reading of L5 is that when satisfying a precondition ϕ it is safe to use the value of the read, as long as the action is not a write. If the pomset is empty, however, there is no value v to use. Therefore the best we can do is to emulate skip, as in L6.

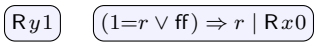
Ex 45. Revisiting Ex 43 and eliding irrelevant transformers:



Composing sequentially:



At top-level, *TOP* yields:



The precondition of $(\mathsf{R}x0)$ is a tautology, as required.

4.3. Full semantics for ARM

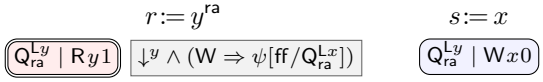
Def 46 combines all of the features of §3–4.

Def 46 (CO/SYNC/DGR/RRD)**.** Let $[\phi/\mathsf{Q}]$ be the substitution $[\phi/\mathsf{Q}^*_{\mathsf{ro}}][\phi/\mathsf{Q}^*_{\mathsf{wo}}][\phi/\mathsf{Q}_{\mathsf{sc}}]$. Update Def 24 to:

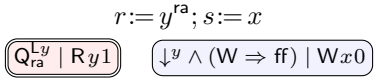
S3) $\kappa(e)$ implies $\mathsf{Q}^{\mathsf{S}x}_\mu \wedge M{=}v$,
L3) $\kappa(e)$ implies $\mathsf{Q}^{\mathsf{L}x}_\mu$.

S4) $\tau^D(\psi)$ implies $\psi[(\mathsf{Q}^x_{\mathsf{wo}} \wedge M{=}v)/\mathsf{Q}^x_{\mathsf{wo}}][\mu/{\downarrow^x}]$
S5) $\tau^C(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}x}_\mu][\mu/{\downarrow^x}]$
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies ${\downarrow^x_\mu} \wedge ((v{=}r \vee \mathsf{W}) \Rightarrow \psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_\mu])$.

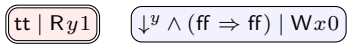
Ex 47. RRD does not adversely affect subscription (Ex 37).



Since $\mathsf{Q}^{\mathsf{L}y}_{\mathsf{ra}}[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}x}_{\mathsf{ra}}]$ is false, we have:



Applying *TOP* yields:



Although $(\mathsf{ff} \Rightarrow \mathsf{ff})$ is a tautology, $\downarrow^y$ is not. Thus, the pomset can only contribute to a top-level pomset if the thread is preceded by a relaxed write to y, allowing $(\mathsf{R}y1)$ to be downgraded to a relaxed read.

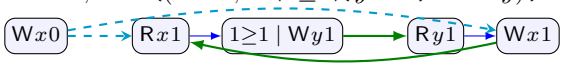
Every ARM8 execution is allowed by Def 46. The proof of this fact is simplified by the recent characterization of ARM8 in terms of *external global completion* (EGC) [3, §B2.3.6]. Under EGC, an ARM8 execution is a linearization of per-location program order and a subset of local-order. Every such linearization is also a valid pomset under Def 46.

5. Other Features

5.1. Local Invariant Reasoning (LIR)

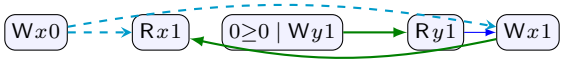
Ex 48. JMM causality Test Case 1 [21] states the following execution should be allowed "since interthread compiler analysis could determine that x and y are always nonnegative, allowing simplification of $r{\geq}0$ to true, and allowing write $y{:=}1$ to be moved early."

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



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:

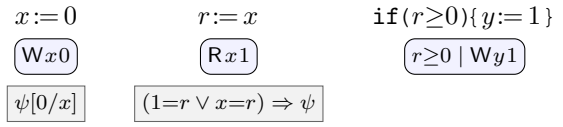


Def 49 (LIR)**.** Update Def 24 to:

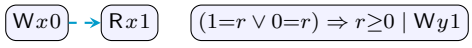
S4) $\tau^D(\psi)$ implies $\psi[M/x]$,
S5) $\tau^C(\psi)$ implies $\psi[M/x]$,
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $(v{=}r \vee x{=}r) \Rightarrow \psi$, when $E \neq \emptyset$,
L6) $\tau^B(\psi)$ implies ψ, when $E = \emptyset$.

L5 introduces memory references. It states that to be independent of the read, we must 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. As in Def 44, we include L6 to provide a predicate transformer for the empty pomset.

Ex 50. Revisiting Ex 48 and eliding irrelevant transformers:



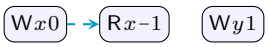
$$x{:=}0; r{:=}x; \texttt{if}(r{\geq}0)\{y{:=}1\}$$



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

If L5 required only that $x{=}r \Rightarrow \psi$, then the following execution would be allowed:

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



But this would violate the expected local invariant: that all values seen for x are nonnegative.

It is worth emphasizing that this reasoning is local, and therefore unaffected by the introduction of additional threads, as in Test Case 9 [21].

Some care is necessary when combining LIR Def 49 and RRD Def 44. The proper form for L5 is:

L5) $\tau^C(\psi)$ implies $(v{=}r \vee (\mathsf{W} \wedge x{=}r)) \Rightarrow \psi$.

When W is true, this is unchanged from L5 in Def 49. When W is false, it is the same as L4 in Def 49.

One must also be careful when combing LIR with RMW operations, such as `CAS`. For these operations, LIR is unsound. Thus reads in RMWs should always use L6 for the independent case.

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 51. 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."

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

This execution is not allowed under Def 49, since the precondition of $(\mathsf{W}y1)$ in the independent case is

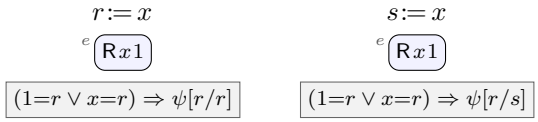
$$(r{=}1 \vee r{=}x) \Rightarrow (s{=}1 \vee s{=}r) \Rightarrow (r{=}s),$$

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:

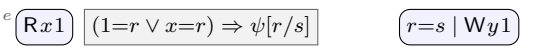
Def 52 (ALPHA)**.** Update Def 24 to:

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

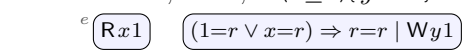
Ex 53. Revisiting Ex 51 and choosing $s_e = r$:



Coalescing and composing:



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



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

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) $\lambda(e) = \mathsf{W} x v_e$,
S3) $\kappa(e)$ implies $\theta_e \wedge \mathsf{Q}_\mu^{\mathsf{S}x} \wedge M{=}v_e$,
S4) $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/x][\mu/{\downarrow^x}][(\mathsf{Q}^x_{\mathsf{wo}} \wedge M{=}v_e)/\mathsf{Q}^x_{\mathsf{wo}}]$,
S5) $(\forall e \in E \setminus D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/x][\mu/{\downarrow^x}][\mathsf{ff}/\mathsf{Q}_\mu^{\mathsf{S}x}]$,
S6) $\tau^D(\psi)$ implies $(\nexists e \in E \mid \theta_e) \Rightarrow \psi[M/x][\mu/{\downarrow^x}][\mathsf{ff}/\mathsf{Q}_\mu^{\mathsf{S}x}]$.

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

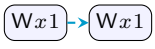
L1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
L2) $\lambda(e) = \mathsf{R} x v_e$,
L3) $\kappa(e)$ implies $\theta_e \wedge \mathsf{Q}_\mu^{\mathsf{L}x}$,
L4) $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow v_e{=}s_e \Rightarrow \psi[s_e/r]$,
L5) $(\forall e \in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

Figure 1: Full Semantics of Loads and Stores (See Def 11 for Q, Def 33 for $\mathsf{Q}_\mu^{\mathsf{S}x}$, $\mathsf{Q}_\mu^{\mathsf{L}x}$, and Def 39 for ${\downarrow_\mu^x}$)

5.3. If-Closure (IF)

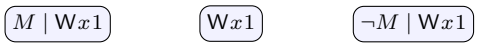
Ex 54. If $S = (x{:=}1)$, then Def 24 does *not* allow:

$$\texttt{if}(M)\{x{:=}1\}; S; \texttt{if}(\neg M)\{x{:=}1\}$$



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:

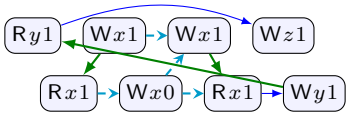
$$\texttt{if}(M)\{x{:=}1\} \qquad x{:=}1 \qquad \texttt{if}(\neg M)\{x{:=}1\}$$



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. For example, Def 24 does not allow the following.

$$r{:=}y; \texttt{if}(r)\{x{:=}1\}; x{:=}1; \texttt{if}(\neg r)\{x{:=}1\}; z{:=}r$$
$$\Vert\ \texttt{if}(x)\{x{:=}0; \texttt{if}(x)\{y{:=}1\}\}$$



Def 55 (ALPHA/IF)**.** Update Def 24 to:
If $P \in \mathit{STORE}(x, M, \mu)$ then $(\exists v : E \to \mathcal{V})\ (\exists \theta : E \to \Phi)$

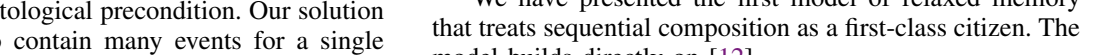
S1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
S2) $\lambda(e) = \mathsf{W} x v_e$,
S3) $\kappa(e)$ implies $\theta_e \wedge M{=}v$,
S4) $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi$,
S5) $\tau^C(\psi)$ implies $(\nexists e \in E \cap C \mid \theta_e) \Rightarrow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

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) $\lambda(e) = \mathsf{R} x v_e$,
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 56. Revisiting Ex 54, we can split the middle command:

$$\texttt{if}(M)\{x{:=}1\} \qquad x{:=}1 \qquad \texttt{if}(\neg M)\{x{:=}1\}$$



Coalescing events gives the desired result.

These examples show that we must allow inconsistent predicates in a single pomset, unlike [12].

6. Conclusions

We have presented the first model of relaxed memory that treats sequential composition as a first-class citizen. The model builds directly on [12].

For sequential composition, parallel composition and the conditional, we believe that the definition is *natural*, even *canonical*. For stores and loads, instead, the definition in Fig 1 is a Frankenstein's monster of features. This complexity is *essential*, however, not just an accident of our poor choices. Relaxed memory models must please many audiences: compiler writers want one thing, hardware designers another, and programmers yet another still. The result is inevitably full of compromise.

Given that *complexity* cannot be eliminated from relaxed memory models, the best one can do is attempt to understand its causes. We have broken the problem into seven manageable pieces, discussed throughout §3–5. Def 46 summarizes all the features necessary for efficient implementation on ARM8. We discuss address calculation, read-modify-write operations and fences in the appendix.

Logic is the thread that sews these features together.

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

[4] 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.

[5] 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.

[6] 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.

[7] S. Dolan, K. Sivaramakrishnan, and A. Madhavapeddy. Bounding data races in space and time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 242–255, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192421. URL http://doi.acm.org/10.1145/3192366.3192421.

[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.

Appendix A. Additional Features

A.1. Address Calculation (ADDR)

Def 58 (ADDR)**.** Update Def 24 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 $L{=}\ell$.

L4) $\tau^D(\psi)$ implies $(L{=}\ell \Rightarrow v{=}r) \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies ψ.

Fig 3 describes the full semantics with address calculation.

Ex 59. Def 58 is naive with respect to merging events. Consider the following example from [12]:

$[r] := 0;\ [0] := !r$ $\qquad$ $[r] := 0;\ [0] := !r$

$^c(r{=}1 \mid \mathsf{W}[1]0)$ $^d(r{=}1 \mid \mathsf{W}[0]0)$ $\qquad$ $^d(r{=}0 \mid \mathsf{W}[0]0) \to {}^e(r{=}0 \mid \mathsf{W}[0]1)$

Merging, we have:

$\texttt{if}(M)\{\,[r] := 0;\ [0] := !r\,\}\ \texttt{else}\ \{\,[r] := 0;\ [0] := !r\,\}$

$^c(r{=}1 \mid \mathsf{W}[1]0)$ $^d(r{=}0 \vee r{=}1 \mid \mathsf{W}[0]0) \to {}^e(r{=}0 \mid \mathsf{W}[0]1)$

The precondition of $\mathsf{W}[0]0$ is a tautology; however, this is not possible for $([r] := 0;\ [0] := !r)$ alone, using Def 58. The full semantics, given in Fig 3, enables this execution using if-closure. The individual commands have the pomsets:

$[r] := 0$ $\qquad$ $[0] := !r$

$^c(r{=}1 \mid \mathsf{W}[1]0)$ $\qquad$ $^d(r{=}1 \mid \mathsf{W}[0]0)$

$^d(r{=}0 \mid \mathsf{W}[0]0)$ $\qquad$ $^e(r{=}0 \mid \mathsf{W}[0]1)$

Sequencing and merging, we have:

$[r] := 0;\ [0] := !r$

$^c(r{=}1 \mid \mathsf{W}[1]0)$ $^d(r{=}0 \vee r{=}1 \mid \mathsf{W}[0]0) \to {}^e(r{=}0 \mid \mathsf{W}[0]1)$

The precondition of $(\mathsf{W}[0]0)$ is a tautology, as required.

A.2. Read-Modify-Write Operations (RMW)

[Todo]

A.3. Fence Operations (FENCE)

[Todo]

Appendix B. Discussion

B.1. Closure properties

We would like the semantics tp be closed with respect to *augments* and *downsets*.

Augments include more order and stronger formulae; in examples, we typically consider pomsets that are augment-minimal. One intuitive reading of augment closure is that adding order can only cause preconditions to weaken.

Def 60. P_2 is an *augment* of P_1 if

1) $E_2 = E_1$,
2) $\lambda_2(e) = \lambda_1(e)$,
3) $\kappa_2(e)$ implies $\kappa_1(e)$,
4) $\tau_2^D(e)$ implies $\tau_1^D(e)$,
5) if $d \leq_2 e$ then $d \leq_1 e$.

Prop 61. *If* $P_1 \in [\![S]\!]$ *and* P_2 *augments* P_1 *then* $P_2 \in [\![S]\!]$.

Downsets include a subset of initial events, similar to *prefixes* for strings.

Def 62. P_2 is an *downset* of P_1 if

1) $E_2 \subseteq E_1$,
2) $(\forall e \in E_2)\ \lambda_2(e) = \lambda_1(e)$,
3) $(\forall e \in E_2)\ \kappa_2(e) = \kappa_1(e)$,
4) $(\forall e \in E_2)\ \tau_2^D(e) = \tau_1^D(e)$,
5) $(\forall d \in E_2)\ (\forall e \in E_2)\ d \leq_2 e$ if and only if $d \leq_1 e$,
6) $(\forall d \in E_1)\ (\forall e \in E_2)$ if $d \leq_1 e$ then $d \in E_2$.

Ex 63. The semantics is not downset closed, due to the lack of read-read dependencies. Consider

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

$(\mathsf{R}x0)$ $\qquad$ $(\mathsf{R}y0)$

The semantics of this program includes the singleton pomset $(\mathsf{R}x0)$, but not the singleton pomset $(\mathsf{R}y0)$. To get $(\mathsf{R}x0)$, we combine:

$r := x$ $\qquad$ $\texttt{if}(!r)\{\,s := y\,\}$

$(\mathsf{R}x0)$ $\qquad$ $\emptyset$

Attempting to get $(\mathsf{R}y0)$, we instead get:

$r := x$ $\qquad$ $\texttt{if}(!r)\{\,s := y\,\}$

$\emptyset$ $\qquad$ $(r{=}0 \mid \mathsf{R}y0)$

Since r appears only once in the program, this pomset cannot contribute to a top-level pomset.

In Def 44, these cases are defined:

L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $(v{=}r \vee \mathsf{W}) \Rightarrow \psi$,
L6) $\tau^B(\psi)$ implies ψ, when $E = \emptyset$.

For writes, L5 is the same as L6. For reads, L5 is the same as L4. When attempting to prove downset closure, the lemma that fails states that L5 should behave like L6: if you have two events in program order, but unordered in the pomset, then the predicate transformer for the empty set case and the independent case of the first event do the same thing to precondition of the second event, modulo quiescence.

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 $\theta_e \wedge \mathsf{Q}_\mu^{\mathsf{S}[\ell_e]} \wedge L{=}\ell_e \wedge M{=}v_e$,
S4) $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/[\ell_e]][\mu/{\downarrow}^{[\ell_e]}][(\mathsf{Q}_{\mathsf{wo}}^{[\ell_e]} \wedge M{=}v_e \wedge L{=}\ell_e)/\mathsf{Q}_{\mathsf{wo}}^{[\ell_e]}]$,
S5) $(\forall e \in E \setminus D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/[\ell_e]][\mu/{\downarrow}^{[\ell_e]}][\mathsf{ff}/\mathsf{Q}_\mu^{\mathsf{S}[\ell_e]}]$,
S6) $(\forall k)\ \tau^D(\psi)$ implies $(\nexists e \in E \mid \theta_e) \Rightarrow (L{=}k) \Rightarrow \psi[M/[k]][\mu/{\downarrow}^{[k]}][\mathsf{ff}/\mathsf{Q}_\mu^{\mathsf{S}[k]}]$.

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 $\theta_e \wedge \mathsf{Q}_\mu^{\mathsf{L}[\ell_e]} \wedge L{=}\ell_e$,
L4) $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow (L{=}\ell_e \Rightarrow v_e{=}s_e) \Rightarrow \psi[s_e/r]$,
L5) $(\forall e \in E \setminus D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow {\downarrow}_\mu^{[\ell_e]} \wedge \big(\langle (L{=}\ell_e \Rightarrow v_e{=}s_e) \vee (\mathsf{W} \wedge (L{=}\ell_e \Rightarrow [\ell_e]{=}s_e))\rangle \Rightarrow \psi[s_e/r][\mathsf{ff}/\mathsf{Q}_\mu^{\mathsf{L}[\ell_e]}]\big)$,
L6) $(\forall k)(\forall s)\ \tau^D(\psi)$ implies $(\nexists e \in E \mid \theta_e) \Rightarrow (L{=}k) \Rightarrow ({\downarrow}_\mu^{[k]} \wedge \psi[s/r][\mathsf{ff}/\mathsf{Q}_\mu^{\mathsf{L}[k]}])$.

Figure 3: Full Semantics of Loads and Stores (See Def 11 for Q, Def 33 for $\mathsf{Q}_\mu^{\mathsf{S}x}$, $\mathsf{Q}_\mu^{\mathsf{L}x}$, and Def 39 for ${\downarrow}_\mu^x$)

B.2. Comparison with Weakest Preconditions

We compare traditional transformers to the dependent-case transformers of Def 49; thus we consider only totally ordered executions. Because we only consider the dependent case, we drop the superscript E on τ^E throughout this section. We also assume that each register appears at most once in a program, as we did throughout §2–4.

Because of augment closure, we are not interested in isolating the *weakest* precondition. Thus we think of transformers as Hoare triples. In addition, all programs in our language are strongly normalizing, so we need not distinguish strong and weak correctness. In this setting, the Hoare triple $\{\phi\}\ S\ \{\psi\}$ holds exactly when $\phi \Rightarrow \mathit{wp}_S(\psi)$.

Hoare triples do not distinguish thread-local variables from shared variables. Thus, the assignment rule applies to all types of storage. The rules can be written as follows:

$$\begin{aligned}\mathit{wp}_{x:=M}(\psi) &= \psi[M/x]\\ \mathit{wp}_{r:=M}(\psi) &= \psi[M/r]\\ \mathit{wp}_{r:=x}(\psi) &= x{=}r \Rightarrow \psi\end{aligned}$$

Here we have chosen an alternative formulation for the read rule, which is equivalent the more traditional $\psi[x/r]$, as long as registers occur at most once in a program. In Def 49, the transformers for the dependent case are as follows:

$$\begin{aligned}\tau_{x:=M}(\psi) &= \psi[M/x]\\ \tau_{r:=M}(\psi) &= \psi[M/r]\\ \tau_{r:=x}(\psi) &= v{=}r \Rightarrow \psi \qquad \text{where } \lambda(e) = \mathsf{R}\,x\,v\end{aligned}$$

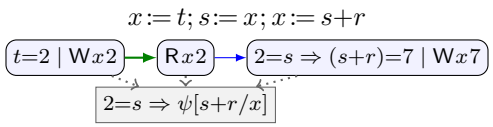
Only the read rule differs from the traditional one.

For programs where every register is bound and every read is fulfilled, our dependent transformers are the same as the traditional ones. In our semantics, thus, we only consider totally-ordered executions where every read could be fulfilled by prepending some writes. For example, we ignore pomsets of $x{:=}2; r{:=}x$ that read 1 for x.

For example, let S_i be defined:

$$\begin{aligned}S_1 &= s{:=}x; x{:=}s{+}r\\ S_2 &= x{:=}t; S_1\\ S_3 &= t{:=}2; r{:=}5; S_2\end{aligned}$$

The following pomset appears in the semantics of S_2. A pomset for S_3 can be derived by substituting $[2/t, 5/r]$. A pomset for S_1 can be derived by eliminating the initial write.

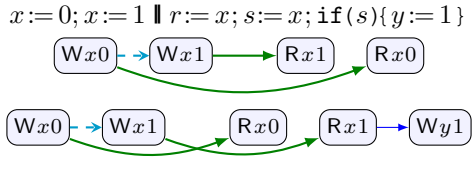


The predicate transformers are:

$$\begin{aligned}\mathit{wp}_{S_1}(\psi) &= x{=}s \Rightarrow \psi[s{+}r/x] \quad & \tau_{S_1}(\psi) &= 2{=}s \Rightarrow \psi[s{+}r/x]\\ \mathit{wp}_{S_2}(\psi) &= t{=}s \Rightarrow \psi[s{+}r/x] \quad & \tau_{S_2}(\psi) &= 2{=}s \Rightarrow \psi[s{+}r/x]\\ \mathit{wp}_{S_3}(\psi) &= 2{=}s \Rightarrow \psi[s{+}5/x] \quad & \tau_{S_3}(\psi) &= 2{=}s \Rightarrow \psi[s{+}5/x]\end{aligned}$$

B.3. Completed Pomsets and Fork

It is sometimes useful to distinguish *terminated* or *completed* executions from partial executions. For example in $[\![x{:=}1; y{:=}1]\!]$, we expect completed executions to include two write actions. Note that this is different from being downset-maximal.



(1)

(2)

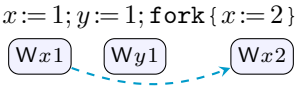
(1) is a downset of (2), but both are completed.

For pomsets with predicate transformers, we identify *completion* with *quiescence.*

Def 64. A pomset with predicate transformers P is *completed* if, for every quiescence symbol s, $\tau^E(\mathsf{s})$ implies s.

For example, there are no pomsets in $[\![\texttt{abort}]\!]$ that are completed, whereas the augment-minimal pomset of $[\![\texttt{skip}]\!]$ (which has the identity transformer) is completed.

The `fork` operation is asynchronous: In $[\![S_1; S_2 \texttt{fork}\{G\}; S_3]\!]$, the only order enforced between $[\![S_1; S_2; S_3]\!]$ and $[\![G]\!]$ comes from quiescence preconditions in $[\![G]\!]$; the transformer of $[\![\texttt{fork}\{G\}]\!]$ is the identity transformer, thus $[\![G]\!]$ runs concurrently with $[\![S_3]\!]$. In addition, if S_2 includes no releases and the locations of S_2 are disjoint from those of G, then $[\![G]\!]$ run concurrently with $[\![S_2; S_3]\!]$.



B.4. Fork-Join

In this subsection, we model a variant of our language that removes the asynchronous `fork` operation and adds a synchronous `fork-join`.

$$S ::= \texttt{abort} \mid \texttt{skip} \mid r := M \mid r := [L]^{\mu} \mid [L]^{\mu} := M \mid \texttt{fork}\{G\}; \texttt{join} \mid S_1; S_2 \mid \texttt{if}(M)\{S_1\}\texttt{else}\{S_2\}$$

In $(S_1; \texttt{fork}\{G\}; \texttt{join}; S_2)$, S_1 must complete before G begins, and threads in G must complete before S_2 begins. Thus $(\texttt{fork}\{r := x\}; \texttt{join})$ acts like a full fence. As modeled here, however, if G is empty, no order is imposed between S_1 and S_2. Thus $[\![\texttt{fork}\{\texttt{skip}\}; \texttt{join}]\!] = [\![\texttt{skip}]\!]$.

To model `fork-join`, we give the semantics of thread groups using pomsets with preconditions *and termination*.

Def 65. A *pomset with preconditions and termination* is a pomset with preconditions (Def 13) together with a termination predicate (notation ✓).

Def 66. If $P \in \mathit{THRD}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$

1–4) as for *THRD* in Def 22,
5) if ✓ then P is completed (Def 64).

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 Def 13,
9) ✓ implies $\checkmark_1 \wedge \checkmark_2$.

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

1–3) as for *FORK* in Def 22,
4) $\kappa(e)$ implies $\kappa_1(e)$,
5) $\kappa(e)$ implies s, for every quiescence symbol s,
6) $\tau^D(\psi)$ implies ψ, if $D = E$ and $\checkmark_1$,
7) $\tau^D(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}]$, otherwise.

Def 67. Update Def 25 to include:

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

We embed pomsets with predicate transformers into pomsets with preconditions and termination using completion. The rules for thread groups keep track of the termination predicate. As noted in §B.3, every pomset in $[\![\texttt{fork}\{G\}]\!]$ is completed. In contrast, a pomset in $[\![\texttt{fork}\{G\}; \texttt{join}]\!]$ is completed only if every thread in G is completed.

Top-level thread groups do not need quiescence symbols; thus, *THRD* removes all quiescence symbols by substitution. However, $\mathit{FORKJOIN}(\mathcal{P})$ adds every possible quiescence symbol as a precondition to the events of $\mathcal{P}$. For example, the preconditions of $[\![S \parallel 0]\!]$ do not contain quiescence symbols. Instead, the preconditions of $[\![\texttt{fork}\{S \parallel 0\}; \texttt{join}]\!]$ are saturated with them. As a result, in completed top-level pomsets of $[\![S_1; \texttt{fork}\{G\}; \texttt{join}]\!]$, all of the events from $[\![S_1]\!]$ must precede those of $[\![G]\!]$.

A similar thing happens with predicate transformers. Thread groups in $[\![S \parallel 0]\!]$ do not contain predicate transformers. Instead, all of the independent predicate transformers of $[\![\texttt{fork}\{S \parallel 0\}; \texttt{join}]\!]$ take ψ to $\psi[\mathsf{ff}/\mathsf{Q}]$. As a result, in completed top-level pomsets of $[\![\texttt{fork}\{G\}; \texttt{join}; S_2]\!]$, all of the events from $[\![G]\!]$ must precede those of $[\![S_2]\!]$.

B.5. Using Independency for Coherence and Synchronization

In §3.2, we encoded coherence and synchronized access using quiescence symbols. Building on the language with `fork-join`, it is possible to model these using independency (§2.2), rather than encoding them in the logic.

To capture completion, we use a single quiescence symbol: Q.

Update actions to include access modes: $(\mathsf{W}^{\mu} x v)$ and $(\mathsf{R}^{\mu} x v)$. Independency of two sequential actions is determined, in part, by the modes of the two actions, capturing synchronization:

	2nd					
1st	$\mathsf{R}^{\mathsf{rlx}}$	R^{ra}	R^{sc}	$\mathsf{W}^{\mathsf{rlx}}$	W^{ra}	W^{sc}
$\mathsf{R}^{\mathsf{rlx}}$	✓	✓	✓	✓	✗	✗
R^{ra}	✗	✗	✗	✗	✗	✗
R^{sc}	✗	✗	✗	✗	✗	✗
$\mathsf{W}^{\mathsf{rlx}}$	✓	✓	✓	✓	✗	✗
W^{ra}	✓	✓	✓	✓	✗	✗
W^{sc}	✓	✓	✗	✓	✗	✗

Def 68. Including coherence, *independency* is defined:

$$\begin{aligned}
\overset{\mathsf{ra}}{\leftrightarrow} &= \{(\mathsf{R}^{\mu}, \mathsf{R}^{\nu}) \mid \mu = \mathsf{rlx}\} \\
&\cup \{(\mathsf{W}^{\mu}, \mathsf{R}^{\nu}) \mid \mu \neq \mathsf{sc} \vee \nu \neq \mathsf{sc}\} \\
&\cup \{(\mathsf{R}^{\mu}, \mathsf{W}^{\nu}) \mid \mu = \mathsf{rlx} \wedge \nu = \mathsf{rlx}\} \\
&\cup \{(\mathsf{W}^{\mu}, \mathsf{W}^{\nu}) \mid \nu = \mathsf{rlx}\} \\
\overset{\mathsf{co}}{\leftrightarrow} &= \{(\mathsf{R}x, \mathsf{R}y)\} \\
&\cup \{(\mathsf{W}x, \mathsf{R}y) \mid x \neq y\} \\
&\cup \{(\mathsf{R}x, \mathsf{W}y) \mid x \neq y\} \\
&\cup \{(\mathsf{W}x, \mathsf{W}y) \mid x \neq y\} \\
\leftrightarrow &= \{(a, b) \mid a \overset{\mathsf{co}}{\leftrightarrow} b \wedge a \overset{\mathsf{ra}}{\leftrightarrow} b\}
\end{aligned}$$

Def 69. If $P \in (\mathcal{P}_1 ; \mathcal{P}_2)$ then $(\exists P_1 \in \mathcal{P}_1)\ (\exists P_2 \in \mathcal{P}_2)$

1–9) as for ; in Def 24,
10) if $d \in E_1$ and $e \in E_2$ either $d < e$ or $a \leftrightarrow \lambda_2(e)$.

Update the read and write rules of Def 24 to:

S4) $\tau^D(\psi)$ implies $\psi[(\mathsf{Q} \wedge M{=}v)/\mathsf{Q}]$,

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

Ex 70. The logic of quiescence is greatly simplified. Compare the following to Ex 35.

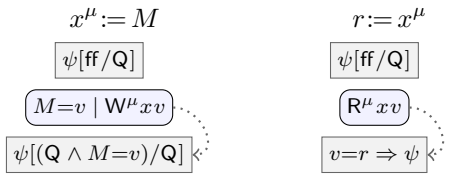


Fig 4 shows the resulting semantics of read and write, without address calculation. Fig 5 shows the resulting semantics with address calculation.

B.6. Substitutions

Recall the load rules from §5.1:

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, when $E = \emptyset$.

It is also possible to collapse x and r when doing a load:

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

Perhaps surprisingly, these two semantics are incomparable. Consider the following:

$$\mathtt{if}(r \wedge s\ \mathsf{even})\{y := 1\}\,;\mathtt{if}(r \wedge s)\{z := 1\}$$
$$r \wedge s\ \mathsf{even} \mid \mathsf{W}y1$$
$$r \wedge s \mid \mathsf{W}z1$$

Prepending $(s := x)$, we get the same result regardless of whether we substitute $[s/x]$, since x does not occur in either precondition. Here we show the independent case:

$$s := x\,;\mathtt{if}(r \wedge s\ \mathsf{even})\{y := 1\}\,;\mathtt{if}(r \wedge s)\{z := 1\}$$
$$(2{=}s \vee x{=}s) \Rightarrow (r \wedge s\ \mathsf{even}) \mid \mathsf{W}y1$$
$$\mathsf{R}x2 \qquad (2{=}s \vee x{=}s) \Rightarrow (r \wedge s) \mid \mathsf{W}z1$$

Prepending $(r := x)$, we now get different results since the preconditions mention x. Without substitution:

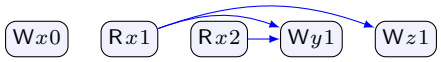
$$r := x\,; s := x\,;\mathtt{if}(r \wedge s\ \mathsf{even})\{y := 1\}\,;\mathtt{if}(r \wedge s)\{z := 1\}$$
$$\mathsf{R}x1 \to 1{=}r \Rightarrow (2{=}s \vee x{=}s) \Rightarrow (r \wedge s\ \mathsf{even}) \mid \mathsf{W}y1$$
$$\mathsf{R}x2 \to 1{=}r \Rightarrow (2{=}s \vee x{=}s) \Rightarrow (r \wedge s) \mid \mathsf{W}z1$$

Prepending $(x := 0)$, which substitutes $[0/x]$, the precondition of $(\mathsf{W}y1)$ becomes $(1{=}r \Rightarrow (2{=}s \vee 0{=}s) \Rightarrow (r \wedge s\ \mathsf{even}))$, which is a tautology, whereas the precondition of $\mathsf{W}z1$ becomes $(1{=}r \Rightarrow (2{=}s \vee 0{=}s) \Rightarrow (r \wedge s))$, which is not. In order to be top-level, $\mathsf{W}z1$ must depend on $\mathsf{R}x2$; in this case the precondition becomes $(1{=}r \Rightarrow 2{=}s \Rightarrow (r \wedge s))$, which is a tautology.

The situation reverses with the substitution $[r/x]$:

$$r := x\,; s := x\,;\mathtt{if}(r \wedge s\ \mathsf{even})\{y := 1\}\,;\mathtt{if}(r \wedge s)\{z := 1\}$$
$$\mathsf{R}x1 \to 1{=}r \Rightarrow (2{=}s \vee r{=}s) \Rightarrow (r \wedge s\ \mathsf{even}) \mid \mathsf{W}y1$$
$$\mathsf{R}x2 \to 1{=}r \Rightarrow (2{=}s \vee r{=}s) \Rightarrow (r \wedge s) \mid \mathsf{W}z1$$

Prepending $(x := 0)$:



The dependency has changed from $(\mathsf{R}x2) \to (\mathsf{W}z1)$ to $(\mathsf{R}x2) \to (\mathsf{W}y1)$. The resulting sets of pomsets are incomparable.

Thinking in terms of hardware, the difference is whether reads update the cache, thus clobbering preceding writes. With $[r/x]$, reads clobber the cache, whereas without the substitution, they do not. Since most caches work this way, the model with $[r/x]$ is likely preferred for modeling hardware. In a software model, however, we see no reason to prefer one of these over the other.

Appendix C. Differences with OOPSLA

Substitution. [12] uses substitution rather than Skolemizing. Indeed our use of Skolemization is motivated by disjunction closure for predicate transformers, which do not appear in [12]; see §2.4.

In §5.1, we give the semantics of load for nonempty pomsets as:

L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $(v{=}r \vee x{=}r) \Rightarrow \psi$.

In [12], the definition is roughly as follows:

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

These substitutions collapse x and r, allowing local invariant reasoning, as in §5.1. Without Skolemizing it is necessary to substitute $[x/r]$, since the reverse substitution $[r/x]$ is useless when r is bound.

Removing the substitution of $[x/r]$ in the independent case has a small technical advantage: we no longer require *extended* expressions (which include memory references), since substitutions no longer introduce memory references.

The substitution $[x/r]$ does not work with Skolemization, even for the dependent case, since we lose the unique marker for each read. In effect, this forces the reads to the same values. To be concrete, the candidate definition would modify L4 to be:

L4) $\tau^D(\psi)$ implies $v{=}x \Rightarrow \psi[x/r]$.

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) $\lambda(e) = \mathsf{W} x v_e$,
S3) $\kappa(e)$ implies $\theta_e \wedge M{=}v_e$,
S4) $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/x][\mu/{\downarrow^x}][(\mathsf{Q} \wedge M{=}v_e)/\mathsf{Q}]$,
S5) $(\forall e \in E \setminus D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/x][\mu/{\downarrow^x}][\mathsf{ff}/\mathsf{Q}]$,
S6) $\tau^D(\psi)$ implies $(\nexists e \in E \mid \theta_e) \Rightarrow \psi[M/x][\mu/{\downarrow^x}][\mathsf{ff}/\mathsf{Q}]$.

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) $\lambda(e) = \mathsf{R} x v_e$,
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) $(\forall e \in E \setminus D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow {\downarrow_\mu^x} \wedge \big(\langle v_e{=}s_e \vee (\mathsf{W} \wedge x{=}s_e)\rangle \Rightarrow \psi[s_e/r][\mathsf{ff}/\mathsf{Q}]\big)$,
L6) $(\forall s)\ \tau^D(\psi)$ implies $(\nexists e \in E \mid \theta_e) \Rightarrow ({\downarrow_\mu^x} \wedge \psi[s/r][\mathsf{ff}/\mathsf{Q}])$.

Figure 4: Full Semantics of Loads and Stores without Address Calculation (See Def 39 for $\downarrow_\mu^x$)

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 $\theta_e \wedge L{=}\ell_e \wedge M{=}v_e$,
S4) $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/[\ell_e]][\mu/{\downarrow^{[\ell_e]}}][(\mathsf{Q} \wedge M{=}v_e \wedge L{=}\ell_e)/\mathsf{Q}]$,
S5) $(\forall e \in E \setminus D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/[\ell_e]][\mu/{\downarrow^{[\ell_e]}}][\mathsf{ff}/\mathsf{Q}]$,
S6) $(\forall k)\ \tau^D(\psi)$ implies $(\nexists e \in E \mid \theta_e) \Rightarrow (L{=}k) \Rightarrow \psi[M/[k]][\mu/{\downarrow^{[k]}}][\mathsf{ff}/\mathsf{Q}]$.

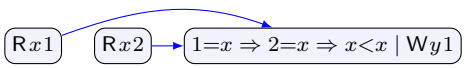
If $P \in \mathit{LOAD}(r,\, L,\, \mu)$ then $(\exists \ell : E \to \mathcal{V})\ (\exists v : E \to \mathcal{V})\ (\exists \theta : E \to \Phi)$

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 $\theta_e \wedge L{=}\ell_e$,
L4) $(\forall e \in E \cap D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow (L{=}\ell_e \Rightarrow v_e{=}s_e) \Rightarrow \psi[s_e/r]$,
L5) $(\forall e \in E \setminus D)\ \tau^D(\psi)$ implies $\theta_e \Rightarrow {\downarrow_\mu^{[\ell_e]}} \wedge \big(\langle (L{=}\ell_e \Rightarrow v_e{=}s_e) \vee (\mathsf{W} \wedge (L{=}\ell_e \Rightarrow [\ell_e]{=}s_e))\rangle \Rightarrow \psi[s_e/r][\mathsf{ff}/\mathsf{Q}]\big)$,
L6) $(\forall k)(\forall s)\ \tau^D(\psi)$ implies $(\nexists e \in E \mid \theta_e) \Rightarrow (L{=}k) \Rightarrow ({\downarrow_\mu^{[k]}} \wedge \psi[s/r][\mathsf{ff}/\mathsf{Q}])$.

Figure 5: Full Semantics of Loads and Stores with Address Calculation (See Def 39 for $\downarrow_\mu^x$)

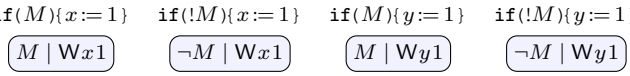
Using this definition, consider the following:

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

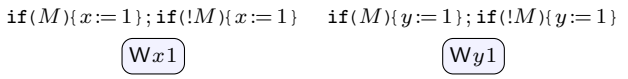


Although the execution seems reasonable, the precondition on the write is not a tautology.

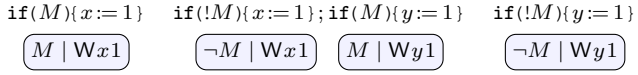
Consistency. [12] imposes *consistency*, which requires that for every pomset P, $\bigwedge_e \kappa(e)$ is satisfiable. Associativity requires that we allow pomsets with inconsistent preconditions. Consider a variant of Ex 54 from §5.3.



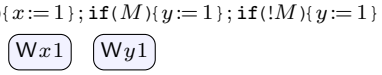
Associating left and right, we have:



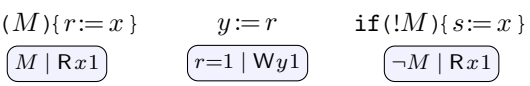
Associating into the middle, instead, we require:



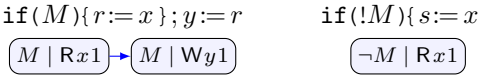
Joining left and right, we have:



Causal Strengthening. [12] imposes *causal strengthening*, which requires for every pomset P, if $d \leq e$ then $\kappa(e)$ implies $\kappa(d)$. Associativity requires that we allow pomsets without causal strengthening. Consider the following.

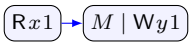


Associating left, with causal strengthening:



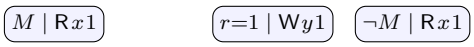
Finally, merging:

$$\texttt{if}(M)\{r:=x\};y:=r;\texttt{if}(!M)\{s:=x\}$$



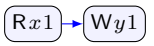
Instead, associating right:

$$\texttt{if}(M)\{r:=x\} \qquad y:=r;\texttt{if}(!M)\{s:=x\}$$



Merging:

$$\texttt{if}(M)\{r:=x\};y:=r;\texttt{if}(!M)\{s:=x\}$$



With causal strengthening, the precondition of $\mathsf{W}y1$ depends upon how we associate. This is not an issue in [12], which always associates to the right.

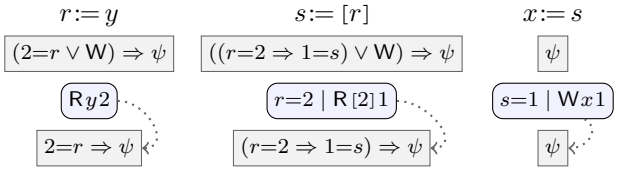
Causal Strengthening and Address Dependencies. In order to guarantee that address calculation does not introduce thin-air executions, the predicate transformer for address calculation must be chosen carefully. Combing Def 44 and Def 58 we have:

L4) $\tau^D(\psi)$ implies $(L{=}\ell \Rightarrow v{=}r) \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $((L{=}\ell \Rightarrow v{=}r) \vee \mathsf{W}) \Rightarrow \psi$.

Consider the following program, from [12, §5], where initially $x = 0$, $y = 0$, $[0] = 0$, $[1] = 2$, and $[2] = 1$. It should only be possible to read 0, disallowing the attempted execution below:

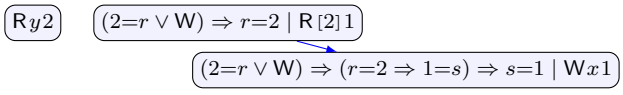
$$r:=y;s:=[r];x:=s \parallel r:=x;s:=[r];y:=s$$

Looking at the left thread:



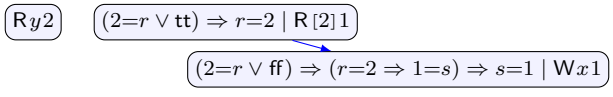
Composing, we have:

$$r:=y;s:=[r];x:=s$$



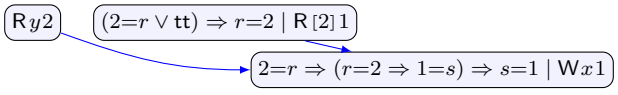
Substituting for W:

$$r:=y;s:=[r];x:=s$$



The precondition of $(\mathsf{R}[2]1)$ is a tautology, but the precondition of $(\mathsf{W}x1)$ is not. This forces a dependency:

$$r:=y;s:=[r];x:=s$$

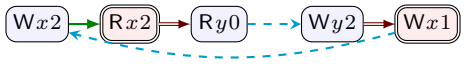


All the preconditions are now tautologies.

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 $\vee$.

Internal 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.

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

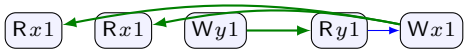


The solution we have adopted is to allow an acquiring read to be downgraded to a relaxed read when it is preceded (sequentially) by a relaxed write that could fulfill it. This solution allows executions that are not allowed under ARM8 since we do not insist that the local relaxed write is actually read from. This may seem counterintuitive, but we don't see a local way to be more precise.

As a result, we use a different proof strategy for ARM8 implementation, which does not rely on read elimination. The proof idea uses a recent alternative characterization of ARM8 [1, 3].

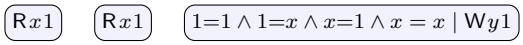
Redundant Read Elimination. Contrary to the claim, redundant read elimination fails for [12]. We discussed redundant read elimination in §5.2. Consider JMM Causality Test Case 2, which we discussed there.

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



Under the semantics of [12], we have

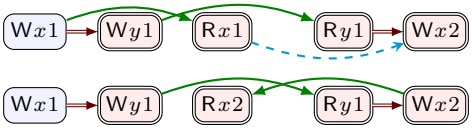
$$r:=x;s:=x;\texttt{if}(r{=}s)\{y:=1\}$$



The precondition of $(\mathsf{W}y1)$ is *not* a tautology, and therefore redundant read elimination fails. (It is a tautology in $r:=x;$ $s:=r;\texttt{if}(r{=}s)\{y:=1\}$.) In [12, §3.1], we incorrectly stated that the precondition of $(\mathsf{W}y1)$ was $1{=}1 \wedge x{=}x$.

A Note on Mixed-Mode Data Races. In preparing this paper, we came across the following example, which appears to invalidate Theorem 4.1 of [8].

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



(3)

(4)

The program is data-race free. The two executions shown are the only top-level executions that include $(\mathsf{W}x2)$.

Theorem 4.1 of [8] is stated by extending execution sequences. In the terminology of [8], a read is *L-weak* if it is sequentially stale. Let $\rho = (\mathsf{W}x1)(\mathsf{W}y1)(\mathsf{R}y1)(\mathsf{W}x2)$ be a sequence and $\alpha = (\mathsf{R}x1)$. ρ is L-sequential and α is L-weak in $\rho\alpha$. But there is no execution of this program that includes a data race, contradicting the theorem. The error seems to be in Lemma A.4 of [8], which states that if α is L-weak after an L-sequential ρ, then α must be in a data race. That is clearly false here, since $(\mathsf{R}x1)$ is stale, but the program is data race free.

In proving the SC-LDRF result in [12, §8], we noted that our proof technique is more robust than that of [8], because it limits the prefixes that must be considered. In (3), the induction hypothesis requires that we add $(\mathsf{R}x1)$ before $(\mathsf{W}x2)$ since $(\mathsf{R}x1) \dashrightarrow (\mathsf{W}x2)$. In particular,

is not a downset of (3), because $(\mathsf{R}x1) \dashrightarrow (\mathsf{W}x2)$. As we noted in [12, §8], this affects the inductive order in which we move across pomsets, but does not affect the set of pomsets that are considered. In particular,

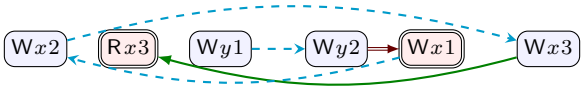
is a downset of (3).

Appendix D. More Stuff

D.1. Downgraded Reads

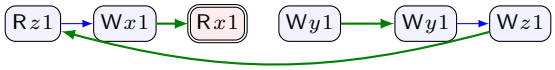
We allow downgrades in executions that are not be allowed by ARM8.

$$x := 2; r := x^{\mathsf{ra}}; y := 1 \parallel y := 2; x^{\mathsf{ra}} := 1 \parallel x := 3$$



ARM8 disallows this because the acquiring read is fulfilled by an external write.

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



ARM8 disallows this because data and control dependencies change acquiring read is fulfilled by an external write.

D.2. If Closure and Address Dependencies

An optimization (p/q are registers):

$$r := [p]\,; s := [q]$$

vs

$$r := [p]\,; \mathtt{if}(p{=}q)\{\,s := r\,\}\mathtt{else}\{\,s := [q]\,\}$$

$$r := \mathtt{new}; [r] := 42; s := [r]; x := r \parallel r := x; [r] := 7$$

If closure is at odds with Java Final field semantics. Do sequencing and if commute?

D.3. About ARM

Hypothesis: gcb cannot contradict (poloc minus RxR).

D.4. Left Par

Another possibility is to use left par and take register state from the left, but quiescence from both sides. This removes the syntax level of thread groups.

$$S \;::=\; \mathtt{abort} \mid \mathtt{skip} \mid r := M \mid r := [L]^{\mu} \mid [L]^{\mu} := M \mid S_1 \parallel S_2 \mid S_1; S_2 \mid \mathtt{if}(M)\{\,S_1\,\}\mathtt{else}\{\,S_2\,\}$$

[Todo.]

D.5. Using Independency for Coherence

It is also possible to use independency only to capture coherence, but the results are less interesting.

In the logic, we remove the symbols $\mathsf{Q}^x_{\mathsf{wo}}$ and $\mathsf{Q}^x_{\mathsf{ro}}$. Previously, we had given the semantics of ra access using $\mathsf{Q}^*_{\mathsf{wo}}$ and $\mathsf{Q}^*_{\mathsf{ro}}$, which were encoded using $\mathsf{Q}^x_{\mathsf{wo}}$ and $\mathsf{Q}^x_{\mathsf{ro}}$. With these gone, we introduce the quiescence symbol Q^* and $\mathsf{Q}_{\mathsf{acq}}$. Thus, the only quiescence symbols required are Q^*, $\mathsf{Q}_{\mathsf{acq}}$ and Q_{sc}. Fig 6 shows the difference with the semantics of §3.2.

Def 71. Let formulae $\mathsf{Q}^{\mathsf{S}}_{\mu}$ and $\mathsf{Q}^{\mathsf{L}}_{\mu}$ be defined:

$$\begin{array}{ll} \mathsf{Q}^{\mathsf{S}}_{\mathsf{rlx}} = \mathsf{Q}_{\mathsf{acq}} & \mathsf{Q}^{\mathsf{L}}_{\mathsf{rlx}} = \mathsf{Q}_{\mathsf{acq}} \\ \mathsf{Q}^{\mathsf{S}}_{\mathsf{ra}} = \mathsf{Q}_{\mathsf{acq}} \wedge \mathsf{Q}^* & \mathsf{Q}^{\mathsf{L}}_{\mathsf{ra}} = \mathsf{Q}_{\mathsf{acq}} \\ \mathsf{Q}^{\mathsf{S}}_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{acq}} \wedge \mathsf{Q}^* \wedge \mathsf{Q}_{\mathsf{sc}} & \mathsf{Q}^{\mathsf{L}}_{\mathsf{sc}} = \mathsf{Q}_{\mathsf{acq}} \wedge \mathsf{Q}_{\mathsf{sc}} \end{array}$$

Let substitutions $[\phi/\mathsf{Q}^{\mathsf{S}}_{\mu}]$ and $[\phi/\mathsf{Q}^{\mathsf{L}}_{\mu}]$ be defined:

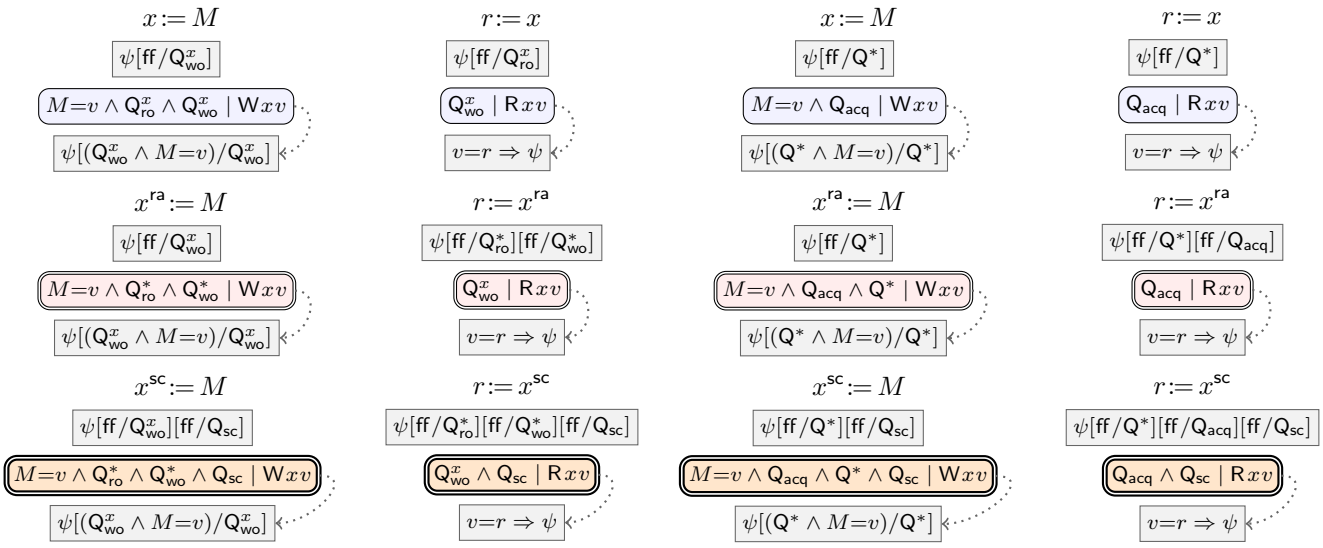
$$\begin{array}{ll} [\phi/\mathsf{Q}^{\mathsf{S}}_{\mathsf{rlx}}] = [\phi/\mathsf{Q}^*] & [\phi/\mathsf{Q}^{\mathsf{L}}_{\mathsf{rlx}}] = [\phi/\mathsf{Q}^*] \\ [\phi/\mathsf{Q}^{\mathsf{S}}_{\mathsf{ra}}] = [\phi/\mathsf{Q}^*] & [\phi/\mathsf{Q}^{\mathsf{L}}_{\mathsf{ra}}] = [\phi/\mathsf{Q}^*, \phi/\mathsf{Q}_{\mathsf{acq}}] \\ [\phi/\mathsf{Q}^{\mathsf{S}}_{\mathsf{sc}}] = [\phi/\mathsf{Q}^*, \phi/\mathsf{Q}_{\mathsf{sc}}] & [\phi/\mathsf{Q}^{\mathsf{L}}_{\mathsf{sc}}] = [\phi/\mathsf{Q}^*, \phi/\mathsf{Q}_{\mathsf{acq}}, \phi/\mathsf{Q}_{\mathsf{sc}}] \end{array}$$

Def 72. Update Def 24 to:

S3) $\kappa(e)$ implies $M{=}v \wedge \mathsf{Q}^{\mathsf{S}}_{\mu}$,
L3) $\kappa(e)$ implies $\mathsf{Q}^{\mathsf{L}}_{\mu}$,

S4) $\tau^D(\psi)$ implies $\psi[(\mathsf{Q}^* \wedge M{=}v)/\mathsf{Q}^*]$,
S5) $\tau^C(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{S}}_{\mu}]$,
L4) $\tau^D(\psi)$ implies $v{=}r \Rightarrow \psi$,
L5) $\tau^C(\psi)$ implies $\psi[\mathsf{ff}/\mathsf{Q}^{\mathsf{L}}_{\mu}]$.

The most interesting examples in Fig 6b concern ra access. Every independent transformer substitutes $[\mathsf{ff}/\mathsf{Q}^*]$. Q^* is a precondition for any releasing write e, ensuring that all preceding events must are ordered before e. Conversely, $\mathsf{Q}_{\mathsf{acq}}$ is a precondition of every event. The independent transformer for any acquiring read e substitutes $[\mathsf{ff}/\mathsf{Q}_{\mathsf{acq}}]$, ensuring that all following events must be ordered after e.



(a) Quiescence Examples (§3.2)

(b) Quiescence Examples (§D.5)

Figure 6: Quiescence Examples for Coherence

As before, the substitution in S4 ensures that left merges are not quiescent (Ex 31).

Item 10 of Def 69 ensures coherence. This definition is incompatible with asynchronous `fork` parallelism of Def 22, where we expect executions such as:

$$\texttt{fork}\{r := x\}; x := 1$$

$$\mathsf{R}x1 \longleftarrow \mathsf{W}x1$$

Item 10 would require $(\mathsf{R}x1) \dashrightarrow (\mathsf{W}x1)$, forbidding this.