

Sequential Composition for Relaxed Memory: Pomsets with Predicate Transformers

Alan Jeffrey* and James Riely†

*Roblox

†DePaul University

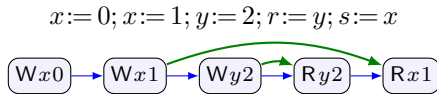
Abstract—This paper presents the first compositional definition of sequential composition that applies to a relaxed memory model weak enough to allow efficient implementation on Arm. We extend the denotational model of pomsets with preconditions with predicate transformers. Previous work has shown that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

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.

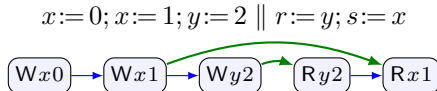
1.1. Memory

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:



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

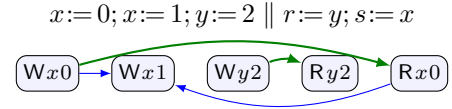
This model naturally extends to the case of shared-memory concurrency, leading to a sequentially consistent semantics [22], in which program order inside a thread implies a total causal order between read and write events, for example:



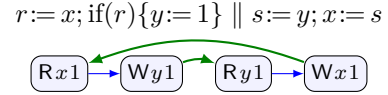
Unfortunately, this model does not compile efficiently to commodity hardware, resulting in a 37–73% increase in CPU time on ARM [24] and, hence, in power consumption. Developers of software and compilers have therefore been faced with a difficult trade-off, between an elegant model of memory, and its impact on resource

usage (such as size of data centers, electricity bills and carbon footprint). Unsurprisingly, many have chosen to prioritize efficiency over elegance.

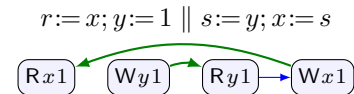
This has led to relaxed memory models, in which the requirement of sequential consistency is weakened to only apply per-location and not globally over the whole program. This allows executions which are inconsistent with program order, such as:



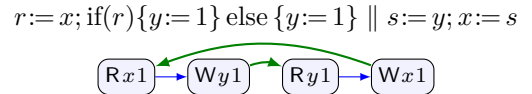
In such models, the causal order between events is important, and includes control and data dependencies, to avoid paradoxical “out of thin air” examples such as:



This candidate execution forms a cycle in causal order, so is disallowed, but this depends crucially on the control dependency from $(Rx1)$ to $(Wy1)$, and the data dependency from $(Ry1)$ to $(Wx1)$. If either is missing, then this execution is acyclic and hence allowed. For example dropping the control dependency results in:



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 $(Rx1)$ to $(Wy1)$, and therefore a cycle in, the candidate execution:



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 [25] and related models [16, 19, 7],
- rewriting methods, which give an operational model up to syntactic rewrites, such as [27], and
- logical methods, such as the pomsets with preconditions model of [17].

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 [17] is to add logical preconditions to events, which are introduced by store actions (modeling data dependencies) and conditionals (modeling control dependencies):

$$\frac{\text{if}(s < 1)\{z := r * s\}}{(s < 1) \wedge (r * s) = 0 \mid Wz0}$$

Preconditions are discharged by being ordered after a read:

$$\frac{r := x; s := y; \text{if}(s < 1)\{z := r * s\}}{\text{Rx0} \rightarrow \text{Ry0} \rightarrow (s=0 \Rightarrow (s < 1) \wedge (r * s) = 0 \mid Wz0)}$$

Note that there is dependency order from (Ry0) to (Wz0) so the precondition for (Wz0) only has to be satisfied assuming the hypothesis ($s=0$). There is no matching order from (Rx0) to (Wz0) which is why we do not assume the hypothesis ($r=0$). Nonetheless, the precondition on (Wz0) is a tautology, and so can be elided in the diagram:

$$\text{Rx0} \rightarrow \text{Ry0} \rightarrow \text{Wz0}$$

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.

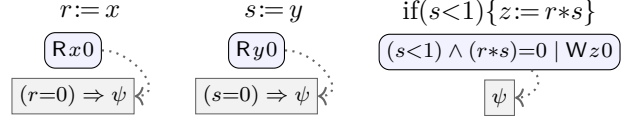
1.2. Sequential composition

Our approach follows that of weakest precondition semantics of Dijkstra [9], which provides an alternative characterization of Hoare logic [14] by mapping postconditions to preconditions. We recall the definition of $\text{wp}_S(\psi)$ for loop-free code below.

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

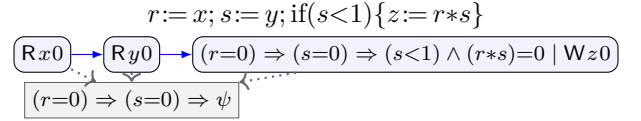
The rule we are most interested in is the one for sequential composition, which maps sequential composition of programs to function composition of predicate transformers.

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:



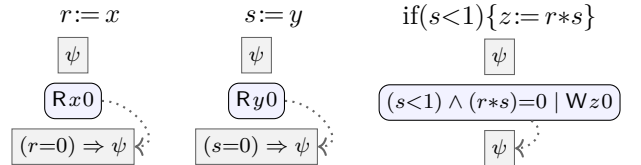
In the rightmost program above, the write to z affects the shared store, not the local state of the thread, therefore we assign it the identity transformer.

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:



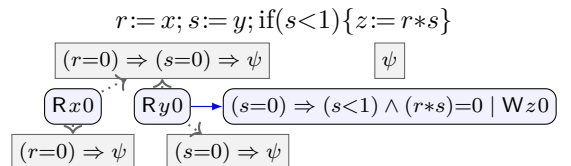
This model works for the sequentially consistent case, but needs to be weakened for the relaxed case. The key observation of this paper is that rather than working with one predicate transformer, we should work with a family of predicate transformers, indexed by sets of events.

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:

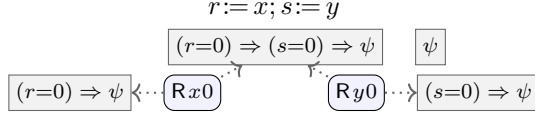


The model of sequential composition then picks which predicate transformer to apply to an event's precondition by picking the one indexed by all the events before it in causal order.

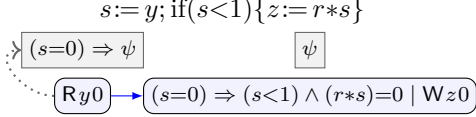
For example, we can recover the expected semantics for the above example by choosing the predicate transformer which is independent of (Rx0) but dependent on (Ry0), which is the transformer which maps ψ to $(s=0) \Rightarrow \psi$.



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

In this paper, we show how pomsets with preconditions [17] can be combined with predicate transformers [9] to create a compositional semantics for sequential composition.

- §3 presents the basic model, with few features required of the logic of preconditions, but a resulting lack of fidelity to existing models,
- §4 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),
- §5 adds the features required for efficient compilation to modern architectures: downgrading some synchronized accesses to relaxed, and removing read-read dependencies, and
- §6 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 [17] apply to our model, including compositional reasoning for temporal safety properties and local drf-sc as in [10, 11]. In §5, we provide an alternative proof strategy for efficient compilation to arm8, which improves upon that of [17] by using a recent alternative characterization of arm8.

2. Related Work

- Models with strong dependencies that disallow compiler optimizations and efficient implementation on arm8: [22, 21, 10, 21, 18, 5, 20].
- Models with weak dependencies that allow some forms of thin-air reasoning: [25, 15, 19, 7]
- Models that do not calculate dependencies: [4, 32, 33, 8].

Like us, Kavanagh and Brookes [20] use pomsets to create a compositional account of sequential composition. However, their model requires a fence after every read on arm8, and uses syntactic dependencies, thus invalidating many compiler optimizations.

3. 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) [13, 28], 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.

3.1. 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 $[v]$. 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.

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

$S ::= \text{skip} \mid r := M \mid r := [L]_{\sigma}^{\mu} \mid [L]_{\sigma}^{\mu} := M \mid F_{\sigma}^{\nu} \mid \text{if}(M)\{S_1\} \text{ else } \{S_2\} \mid r := \text{CAS}([L]_{\sigma}^{\mu_1, \mu_2}, M, N) \mid r := \text{FADD}([L]_{\sigma}^{\mu_1, \mu_2}, M) \mid r := \text{EXCH}$

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

Commands, aka statements, S , include memory accesses at a given mode, as well as the usual structural constructs. Following [12], \parallel denotes parallel composition, preserving thread state on the right after a join. In examples and sublanguages without join, we use the symmetric \parallel operator.

The semantics is built from the following.

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

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

We require that:

- actions include writes (Wxv) and reads (Rxv),
- formulae include equalities ($M=N$) and ($x=M$),
- formulae include quiescence symbols Q_{sc} , Q_{ro}^x , Q_{wo}^x (used in §4), the downgrade symbols \downarrow^x (used in §5.1), and the write symbol W (used in §5.2),

- formulae are closed under negation, conjunction, disjunction, and substitutions $[M/r]$, $[M/x]$, and $[\phi/s]$ for each symbol s ,
- there is an entailment relation \models between formulae,
- \models 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 §6.1, 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 $\phi \models \psi$. We say ϕ is a tautology if $\text{tt} \models \phi$. We say ϕ is unsatisfiable if $\phi \models \text{ff}$.

Throughout §3–5 we additionally require that

- each register is assigned at most once in a program.
- In §6, 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: $S[N/s_e] = S$.

Def 1. Reorderability relations. Coherence-after and synchronization.

$$\begin{aligned} \times_{\text{ca}} &= \{(Wx, Wy) \mid x \neq y\} \cup \{(Rx, Wy) \mid x \neq y\} \cup \{(Wx, Ry) \mid x \neq y\} \cup \{(Rx, Ry) \mid x \neq y\} \\ \times_{\text{sync}} &= \{(W, R)\} \cup \{(W, W)\} \cup \{(F^{\text{rel}}, R)\} \\ &\quad \cup \{(R, W)\} \cup \{(R, R)\} \cup \{(W, F^{\text{acq}})\} \cup \{(F^{\text{rel}}, F^{\text{acq}})\} \\ \times &= \times_{\text{sync}} \cap \times_{\text{ca}} \end{aligned}$$

Action (Wxv) matches (Rxw) when $v = w$. Action (Wxv) blocks (Rxw) , for any v, w .

Actions (W) and $(F^{\nu \neq \text{acq}})$ are release actions.

3.2. 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 2. A pomset over \mathcal{A} is a tuple (E, \leq, λ) where

- $E \subseteq \mathcal{E}$ is a set of events,
- $\leq \subseteq (E \times E)$ is the causality partial order,
- $\lambda : E \rightarrow \mathcal{A}$ 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 3. Action (Wxv) matches (Rxw) when $v = w$. Action (Wxv) blocks (Rxw) , for any v, w .

A read event e is fulfilled if there is a $d \leq e$ which matches it and, for any c which can block e , either $c \leq d$ or $e \leq c$.

We introduce reorderability [26] in order to provide examples with coherence in this subsection. In §4 we show that coherence can be encoded in the logic, making reorderability unnecessary.

Def 4. Actions a and b are reorderable ($a \bowtie b$) if either both are reads or they are accesses to different locations. Formally $\bowtie = \{(Rxv, Ryw)\} \cup \{(Rxv, Wyw), (Wxv, Ryw), (Wxv, Wyw) \mid x \neq y\}$.

Actions that are not reorderable are in conflict.

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

Def 5. If $P \in \text{NIL}$ then $E = \emptyset$.

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

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

If $P \in (a \rightarrow \mathcal{P}_2)$ then $(\exists E_1) (\exists P_2 \in \mathcal{P}_2)$

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

If $P \in \text{TOP}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$

- 1) $E = E_1$,
- 2) $\lambda(e) = \lambda_1(e)$,
- 3) if $d \leq_1 e$ then $d \leq e$,
- 4) if $\lambda_1(e)$ is a read then e is fulfilled (Def 3).

Def 6. For a language fragment, the semantics is:

$$\begin{aligned} \llbracket x^\mu := v; S \rrbracket &= (Wxv) \rightarrow \llbracket S \rrbracket & \llbracket \text{skip} \rrbracket &= \llbracket 0 \rrbracket = \text{NIL} \\ \llbracket r := x^\mu; S \rrbracket &= \bigcup_v (Rxv) \rightarrow \llbracket S \rrbracket & \llbracket S_1 \parallel S_2 \rrbracket &= \llbracket S_1 \rrbracket \parallel \llbracket S_2 \rrbracket \end{aligned}$$

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

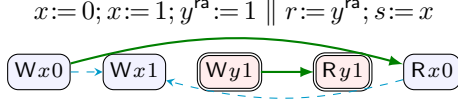
It is worth noting that if \bowtie is taken to be the empty relation, then top-level pomsets of Def 2 correspond to sequentially consistent executions up to mumbling [6].

Ex 7. 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 8. 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 (Wy1) publishes (Wx1) in Ex 30.

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

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

Def 9. \mathcal{P}_1 refines \mathcal{P}_2 if $\mathcal{P}_1 \subseteq \mathcal{P}_2$.

Ex 10. Ex 7 shows that $\llbracket x := 1 \rrbracket$ refines $\llbracket x := 1; x := 1 \rrbracket$.

3.3. 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 [17].

Def 11. A pomset with preconditions is a pomset (Def 2) together with $\kappa : E \rightarrow \Phi$.

Def 12. Let $[\phi/Q]$ substitute all quiescence symbols by ϕ .

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

Def 13. If $P \in \text{NIL}$ 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 5,

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

1–5) as for \parallel in Def 5 (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 $P \in \text{WR}(x, M, \mathcal{P}_2)$ then $(\exists P_2 \in \mathcal{P}_2) (\exists v \in \mathcal{V})$

1–6) as for $(Wxv) \rightarrow \mathcal{P}_2$ in Def 5,

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

1–6) as for $(Rxv) \rightarrow \mathcal{P}_2$ in Def 5,

7) if $e \in E_2 \setminus E_1$ then either

$\kappa(e)$ implies $r=v \Rightarrow \kappa_2(e)$ and $(\exists d \in E_1) d < e$, or
 $\kappa(e)$ implies $\kappa_2(e)$.

If $P \in \text{TOP}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$

1–4) as for TOP in Def 5,

5) if $\lambda_1(e)$ is a write, $\kappa_1(e)[\text{tt}/Q][\text{tt}/W]$ is a tautology,

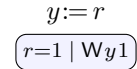
6) if $\lambda_1(e)$ is a read, $\kappa_1(e)[\text{tt}/Q][\text{ff}/W]$ is a tautology.

Let PomPre be the set of all pomsets with preconditions. The function $\text{TOP} : 2^{\text{PomPre}} \rightarrow 2^{\text{Pom}}$ embeds sets of pomsets with preconditions into sets of pomsets. It also substitutes formulae for quiescence and write symbols, for use in §4–5. In these “top-level” pomsets, every read is fulfilled and every precondition is a tautology.

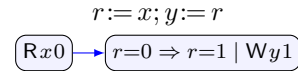
Def 14. For a language fragment, the semantics is:

$$\begin{aligned} \llbracket \text{if}(M)\{S_1\} \text{ else } \{S_2\} \rrbracket &= \text{IF}(M \neq 0, \llbracket S_1 \rrbracket, \llbracket S_2 \rrbracket) \\ \llbracket x^\mu := M; S \rrbracket &= \text{WR}(x, M, \llbracket S \rrbracket) \quad \llbracket \text{skip} \rrbracket = \llbracket 0 \rrbracket = \text{NIL} \\ \llbracket r := x^\mu; S \rrbracket &= \text{RD}(r, x, \llbracket S \rrbracket) \quad \llbracket S_1 \parallel S_2 \rrbracket = \llbracket S_1 \rrbracket \parallel \llbracket S_2 \rrbracket \end{aligned}$$

Ex 15. A simple example of a data dependency is a pomset $P \in \llbracket r := x; y := r \rrbracket$, for which there must be an $v \in \mathcal{V}$ and $P' \in \llbracket y := r \rrbracket$ such as the following, where $v = 1$:

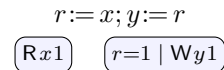


The value chosen for the read may be different from that chosen for the write:

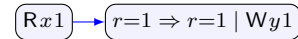


In this case, the pomset’s preconditions depend on a bound register, so cannot contribute to a top-level pomset.

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 $P \in \llbracket r := x; \text{if}(r)\{y := 1\} \rrbracket$, there must be an $v \in \mathcal{V}$ and $P' \in \llbracket \text{if}(r)\{y := 1\} \rrbracket$ such as:

$$\begin{array}{c} \text{if}(r)\{y := 1\} \\ \boxed{r \neq 0 \mid \text{Wy}1} \end{array}$$

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

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

$$\begin{array}{c} y := 1 \\ \boxed{1 = 1 \mid \text{Wy}1} \end{array}$$

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

$$\begin{array}{c} r := x; y := 1 \\ \boxed{\text{Rx}0} \quad \boxed{1 = 1 \mid \text{Wy}1} \end{array}$$

Ex 18. Consider $P \in \llbracket \text{if}(r=1)\{y := r\} \text{ else } \{y := 1\} \rrbracket$, so there must be $P_1 \in \llbracket y := r \rrbracket$, and $P_2 \in \llbracket y := 1 \rrbracket$, such as:

$$\begin{array}{cc} y := r & y := 1 \\ \boxed{r = 1 \mid \text{Wy}1} & \boxed{1 = 1 \mid \text{Wy}1} \end{array}$$

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:

$$\begin{array}{c} \text{if}(r=1)\{y := r\} \text{ else } \{y := 1\} \\ \boxed{(r=1 \Rightarrow r=1) \wedge (r \neq 1 \Rightarrow 1=1) \mid \text{Wy}1} \end{array}$$

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

3.4. 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 $\tau : \Phi \rightarrow \Phi$ such that

- $\tau(\text{ff})$ 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 .

Def 22. If $P \in \text{ABORT}$ then $E = \emptyset$ and

- $\tau^D(\psi)$ implies ff .

If $P \in \text{SKIP}$ then $E = \emptyset$ and

- $\tau^D(\psi)$ implies ψ .

If $P \in \text{LET}(r, M)$ then $E = \emptyset$ and

- $\tau^D(\psi)$ implies $\psi[M/r]$.

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

1-8) as for \parallel in Def 13,

- 9) $\tau^D(\psi)$ implies $\tau_2^D(\psi)$,

10) $\tau^D(s)$ implies $\tau_1^D(s)$, for every quiescence symbol s .

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

1-5) as for \parallel in Def 5 (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 $P \in \text{WRITE}(x, M, \mu)$ then $(\exists v \in \mathcal{V})$

S1) if $d, e \in E$ then $d = e$,

S2) $\lambda(e) = \text{W}xv$,

S3) $\kappa(e)$ implies $M = v$,

S4) $\tau^D(\psi)$ implies $\psi \wedge M = v$,

S5) $\tau^C(\psi)$ implies ψ ,

where $D \cap E \neq \emptyset$ and $C \cap E = \emptyset$.

If $P \in \text{READ}(r, x, \mu)$ then $(\exists v \in \mathcal{V})$

L1) if $d, e \in E$ then $d = e$,

L2) $\lambda(e) = \text{R}xv$,

L3) $\kappa(e)$ implies tt ,

L4) $\tau^D(\psi)$ implies $v = r \Rightarrow \psi$,

L5) $\tau^C(\psi)$ implies ψ ,

where $D \cap E \neq \emptyset$ and $C \cap E = \emptyset$,

If $P \in \text{TOP}(\mathcal{P})$ then $(\exists P_1 \in \mathcal{P})$

1-6) as in Def 13,

- 7) $\tau^{E_1}(s)$ implies s , for every quiescence symbol s .

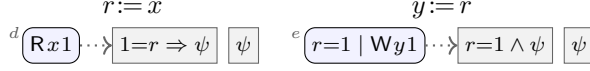
Def 23. The semantics of commands is:

$$\begin{aligned} \llbracket \text{if}(M)\{S_1\} \text{ else } \{S_2\} \rrbracket_\alpha &= \text{IF}(M \neq 0, \llbracket S_1 \rrbracket_\alpha, \llbracket S_2 \rrbracket_\alpha) \\ \llbracket x^\mu := M \rrbracket_\alpha &= \text{WRITE}(x, M, \mu, \sigma)_\alpha & \llbracket \text{skip} \rrbracket_\alpha &= \text{SKIP} \\ \llbracket r := x^\mu \rrbracket_\alpha &= \text{READ}(r, x, \mu, \sigma)_\alpha & \llbracket S_1 \parallel_\gamma S_2 \rrbracket_\alpha &= \llbracket S_1 \rrbracket_\alpha \parallel \llbracket S_2 \rrbracket_\gamma \\ \llbracket r := M \rrbracket_\alpha &= \text{LET}(r, M) & \llbracket S_1; S_2 \rrbracket_\alpha &= \llbracket S_1 \rrbracket_\alpha; \llbracket S_2 \rrbracket_\alpha \\ \llbracket F_\sigma^\mu \rrbracket_\alpha &= \text{FENCE}(\mu, \sigma)_\alpha \end{aligned}$$

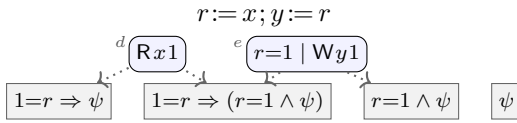
Most of these definitions are straightforward adaptations of §3.3, but the treatment of sequential composition is new. This uses the usual rule for composition of predicate transformers (but preserving the indexing

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 24. For read to write dependency, consider:

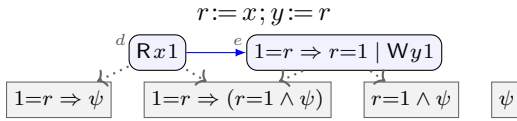


Putting these together without order, we calculate the precondition $\kappa(e)$ as $\tau_1^C(\kappa_2(e))$, where C is $\{c \mid c < e\}$, which is \emptyset . Since $\tau_1^\emptyset(\psi)$ is ψ , this gives that $\kappa(e)$ is $\kappa_2(e)$, which is $r=1$. This gives the pomsaet with predicate transformers:



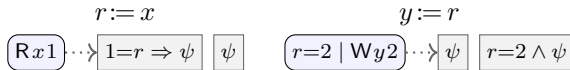
This pomset's preconditions depend on a bound register, so cannot contribute to a top-level pomset.

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

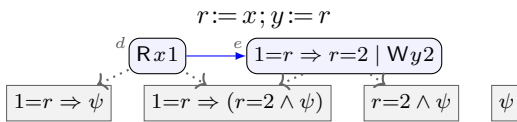


This pomset's preconditions do not depend on a bound register, so can contribute to a top-level pomset.

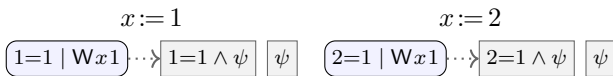
Ex 25. If the read and write choose different values:



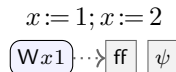
Putting these together with order, we have the following, which cannot be part of a top-level pomset:



Ex 26. S4 includes $M=v$ to ensure that spurious merges do not go undetected. Consider the following.



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



This pomset cannot be part of a top-level pomset, since $\tau^E(s) = \text{ff}$ for every quiescence symbol s . This is what we would hope: that the program $x:=1; x:=2$ should only be top-level if there is a (Wx2) event.

Ex 27. 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 [17]. Attempting to write the predicate transformers in this style we have:

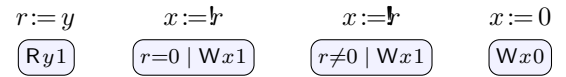
L4) $\tau^D(\psi)$ implies $\psi[v/r]$,

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

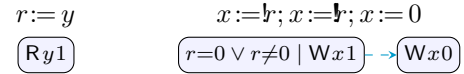
This phrasing of L5 says that ψ must be independent of r in order to appear in a top-level pomset. This choice for L5 is forced by Def 20, which states that the predicate transformer for a small subset of E must imply the transformer for a larger subset.

Sadly, this definition fails associativity.

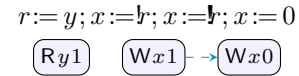
Consider the following, eliding transformers:



Associating to the right and merging:



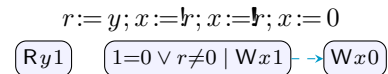
The precondition of (Wx1) is a tautology, thus we have:



If, instead, we associate to the left:



Sequencing and merging:



In this case, the precondition of (Wx1) is not a tautology, forcing a dependency $(\text{Ry1}) \rightarrow (\text{Wx1})$.

Our solution is to Skolemize. We have proven associativity of Def 22 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.)

3.5. 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 §4–6, we explain this definition by looking at its constituent parts, building on Def 22. In §4, we add quiescence, which encodes coherence, release-acquire access, and SC access. In §5, we add peculiarities that are

necessary for efficient implementation on arm8. In §6, 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 22, 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.

4. Quiescence

We introduce quiescence, which captures coherence and synchronized access. Recall from §3.1 that formulae include symbols Q_{sc} , Q_{ro}^x , and Q_{wo}^x . 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 captured in the logic, not in the pomset model itself.

4.1. 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 $(Q_{ro}^x \wedge Q_{wo}^x)$ to events that write to x , for example:

$$x := 1; x := 2$$

$$1=1 \wedge Q_{ro}^x \wedge Q_{wo}^x \mid Wx1 \rightarrow 2=2 \wedge Q_{ro}^x \wedge Q_{wo}^x \mid Wx2$$

These symbols are left alone in the dependent case, but in the independent case we substitute ff for Q_{wo}^x :

$$1=1 \wedge Q_{ro}^x \wedge Q_{wo}^x \mid Wx1 \quad 2=2 \wedge Q_{ro}^x \wedge \text{ff} \mid Wx2$$

This substitution is part of the predicate transformer:

$$x := 1$$

$$1=1 \wedge Q_{ro}^x \wedge Q_{wo}^x \mid Wx1 \dots \psi \quad \psi[\text{ff}/Q_{wo}^x]$$

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

$$r := x$$

$$Q_{wo}^x \mid Rx1 \dots r=1 \Rightarrow \psi \quad \psi[\text{ff}/Q_{ro}^x]$$

In this model, there is no read-read coherence, but to restore it we would identify Q_{ro}^x with Q_{wo}^x .

When threads are initialized, we substitute every quiescence symbol with tt, so at top level there are no remaining quiescence symbols, for example:

$$x := 1; x := 2 \parallel r := x$$

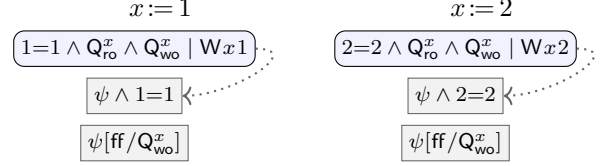
$$1=1 \wedge \text{tt} \wedge \text{tt} \mid Wx1 \dots 2=2 \wedge \text{tt} \wedge \text{tt} \mid Wx2$$

$$\text{tt} \mid Rx1$$

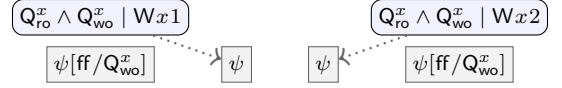
Def 28 (co). Update Def 22:

- S3) $\kappa(e)$ implies $Q_{ro}^x \wedge Q_{wo}^x \wedge M=v$,
- L3) $\kappa(e)$ implies Q_{wo}^x ,
- S5) $\tau^C(\psi)$ implies $\psi[\text{ff}/Q_{wo}^x]$,
- L5) $\tau^C(\psi)$ implies $\psi[\text{ff}/Q_{ro}^x]$.

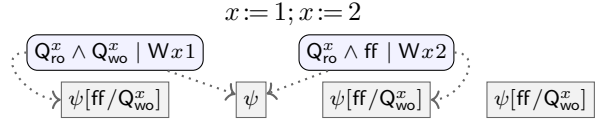
Ex 29. Def 28 enforces coherence. Consider:



Simplifying, we have:



If we attempt to put these together unordered, the precondition of (Wx2) becomes unsatisfiable:

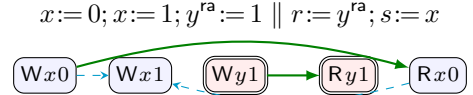


In order to get a satisfiable precondition for (Wx2), we must introduce order:

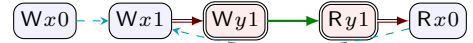


4.2. Synchronized Access (sync)

Ex 30. The publication idiom requires that we disallow the execution below, which is allowed by Def 28.



We disallow this by introducing order $(Wx1) \Rightarrow (Wy1)$ and $(Ry1) \Rightarrow (Rx0)$.



Def 31. Let $Q_{ro}^* = \bigwedge_y Q_{ro}^y$, and similarly for Q_{wo}^* . Let formulae Q_{μ}^{Sx} and Q_{μ}^{Lx} be defined:

$$\begin{aligned} Q_{rx}^{Sx} &= Q_{ro}^x \wedge Q_{wo}^x & Q_{rx}^{Lx} &= Q_{wo}^x \\ Q_{ra}^{Sx} &= Q_{ro}^* \wedge Q_{wo}^* & Q_{ra}^{Lx} &= Q_{wo}^* \\ Q_{sc}^{Sx} &= Q_{ro}^* \wedge Q_{wo}^* \wedge Q_{sc} & Q_{sc}^{Lx} &= Q_{wo}^* \wedge Q_{sc} \end{aligned}$$

Let $[\phi/Q_{ro}^*]$ substitute ϕ for every Q_{ro}^y , and similarly for Q_{wo}^* . Let substitutions $[\phi/Q_{\mu}^{Sx}]$ and $[\phi/Q_{\mu}^{Lx}]$ be defined:

$$\begin{aligned} [\phi/Q_{rx}^{Sx}] &= [\phi/Q_{wo}^x] & [\phi/Q_{rx}^{Lx}] &= [\phi/Q_{ro}^x] \\ [\phi/Q_{ra}^{Sx}] &= [\phi/Q_{wo}^*] & [\phi/Q_{ra}^{Lx}] &= [\phi/Q_{ro}^*, \phi/Q_{wo}^*] \\ [\phi/Q_{sc}^{Sx}] &= [\phi/Q_{wo}^*, \phi/Q_{sc}] & [\phi/Q_{sc}^{Lx}] &= [\phi/Q_{ro}^*, \phi/Q_{wo}^*, \phi/Q_{sc}] \end{aligned}$$

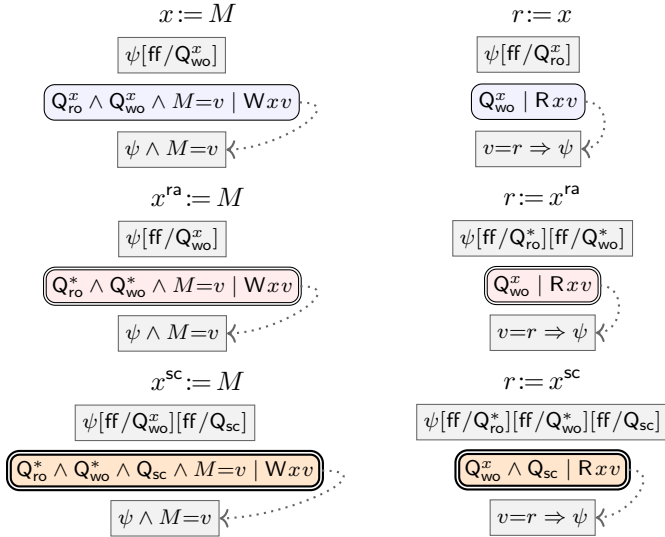
Def 32 (co/sync). Update Def 22 to:

- S3) $\kappa(e)$ implies $Q_{\mu}^{Sx} \wedge M=v$,
L3) $\kappa(e)$ implies Q_{μ}^{Lx} ,
S5) $\tau^C(\psi)$ implies $\psi[\text{ff}/Q_{\mu}^{Sx}]$,
L5) $\tau^C(\psi)$ implies $\psi[\text{ff}/Q_{\mu}^{Lx}]$.

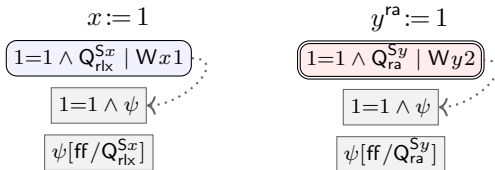
The quiescence formulae indicate what must precede an event. For example, all preceding accesses must be ordered before a releasing write, whereas only writes on x must be ordered before a releasing read on x .

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 13). For example, we substitute **ff** for Q_{ra}^{Sx} 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 Q_{ra}^{Lx} in the independent case for an acquiring write; this ensures that all subsequent accesses follow the acquiring read in top-level pomsets.

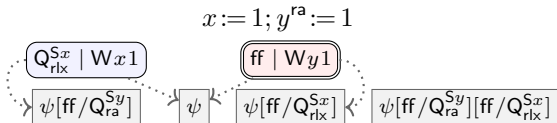
Ex 33. The following pomsets show the effect of quiescence for each access mode.



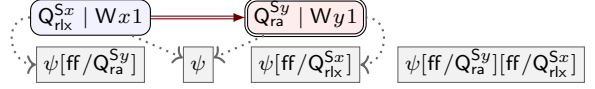
Ex 34. Def 28 enforces publication. Consider:



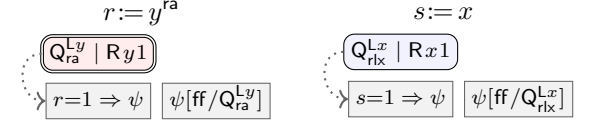
Since $Q_{ra}^{Sy}[\text{ff}/Q_{rx}^{Sx}]$ is **ff**, composing these without order simplifies to:



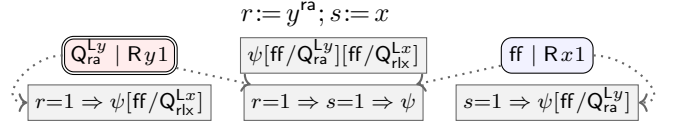
In order to get a satisfiable precondition for (Wy1), we must introduce order:



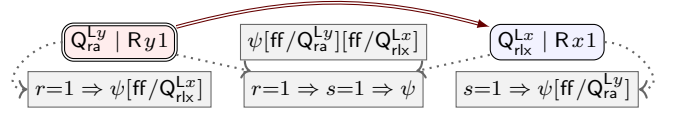
Ex 35. Def 28 enforces subscription. Consider:



Since $Q_{rx}^{Lx}[\text{ff}/Q_{ra}^{Ly}]$ is **ff**, composing these without order simplifies to:



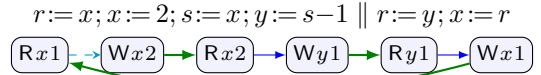
In order to get a satisfiable precondition for (Rx1), we must introduce order:



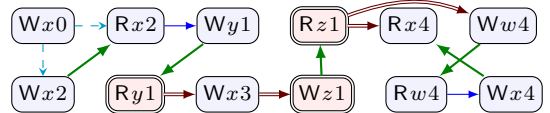
5. Efficient Implementation on ARMv8

We discuss arm8 using external global completion (egc) [1] [3, §B2.3.6] which is very close to our model.

Ex 36. Bad example:



Ex 37. Should not wipe out local state only when there is a release-acquire pair, as in the definition of [17], which allows the following execution. Even though there is a blocker (Wx3), here we are using (Wx0) to justify the lack of dependence (Rx4) → (Ww4).



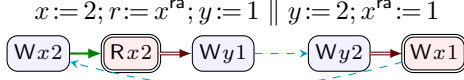
(The read-clobbering semantics from §B.4 does not change this.)

It does seem to me like a non-local blocker for program-order must include a read of x or a release (to get order from the write) and an acquire (to get the order to the read).

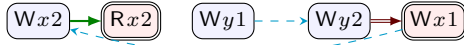
What if an acquire preceded by a release or non-local read on x wiped the local state of x ? The problem with this kind of solution (and P6 in oopsla specifically) is that it destroys roach-motel.

5.1. Downgraded Reads (dgr)

Ex 38. The following example is from Podkopaev [29]. It is allowed by arm8, but disallowed by Def 32. 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 $(Rx2) \Rightarrow (Wy1)$, because $(Rx2)$ 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_μ^x and substitution $[\mu/\downarrow^x]$ be defined:

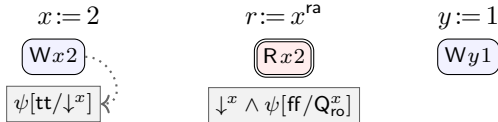
$$\downarrow_\mu^x = \begin{cases} \downarrow^x & \text{if } \mu \neq rlx \\ tt & \text{otherwise} \end{cases} \quad [\mu/\downarrow^x] = \begin{cases} [tt/\downarrow^x] & \text{if } \mu = rlx \\ [tt/\downarrow^x][ff/\downarrow^*] & \text{otherwise} \end{cases}$$

Def 40 (co/sync/dgr). Update Def 32 to:

- S4) $\tau^D(\psi)$ implies $\psi[\mu/\downarrow^x] \wedge M=v$,
- S5) $\tau^C(\psi)$ implies $\psi[\mu/\downarrow^x][ff/Q_\mu^S]$,
- L5) $\tau^C(\psi)$ implies $\downarrow_\mu^x \wedge \psi[ff/Q_\mu^L]$.

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 .

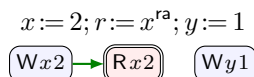
Ex 41. Revisiting Ex 38 and eliding irrelevant transformers:



Associating right:

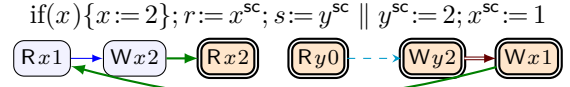


Composing, we have, as desired:



Ex 42. One might worry that our model is too permissive for sc access, but arm8 itself allows some very

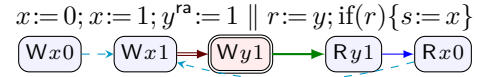
counterintuitive results for sc access. In the following execution we elide the initializing write ($Wy0$).



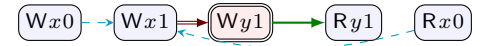
Under egc, this is explained by dropping the order $(Rx2) \Rightarrow (Ry0)$, because $(Rx2)$ is fulfilled by a relaxed write in the same thread.

5.2. Removing Read-Read dependencies (rrd)

Ex 43. The following execution is allowed by arm8, but disallowed by Def 32.



Under egc, this is explained by dropping the order $(Ry1) \Rightarrow (Rx0)$, 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 .

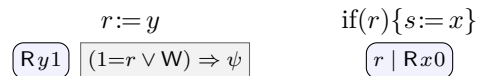
Def 44 (rrd). Update Def 22 to:

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

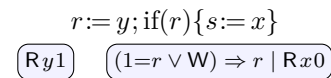
L4 is the dependent case, L5 the independent case, and L6 the empty case. Recall the substitutions performed by TOP (Def 13). For writes, L5 is the same as L6 (since W is tt). For reads, L5 is the same as L4 (since W is ff).

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.

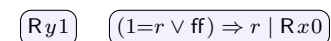
Ex 45. Revisiting Ex 43 and eliding irrelevant transformers:



Composing sequentially:



At top-level, TOP yields:



The precondition of $(Rx0)$ is a tautology, as required.

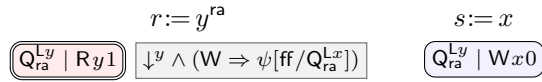
5.3. Full semantics for ARM

Def 46 combines all of the features of §4-5.

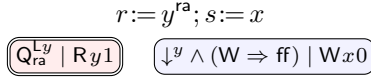
Def 46 (co/sync/dgr/rrd). Let $[\phi/Q]$ be the substitution $[\phi/Q_{ro}^*][\phi/Q_{wo}^*][\phi/Q_{sc}]$. Update Def 22 to:

- S3) $\kappa(e)$ implies $Q_{\mu}^{Sx} \wedge M=v$,
- L3) $\kappa(e)$ implies Q_{μ}^{Lx} .
- S4) $\tau^D(\psi)$ implies $\psi[\mu/\downarrow^x] \wedge M=v$
- S5) $\tau^C(\psi)$ implies $\psi[\mu/\downarrow^x][ff/Q_{\mu}^{Sx}]$
- L4) $\tau^D(\psi)$ implies $v=r \Rightarrow \psi$,
- L5) $\tau^C(\psi)$ implies $\downarrow_{\mu}^x \wedge ((v=r \vee W) \Rightarrow \psi[ff/Q_{\mu}^{Lx}])$,
- L6) $\tau^B(\psi)$ implies $\downarrow_{\mu}^x \wedge \psi$, when $E = \emptyset$.

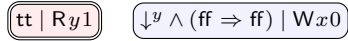
Ex 47. rrd does not adversely affect subscription (Ex 35).



Since $Q_{ra}^{Ly}[ff/Q_{ra}^{Lx}]$ is false, we have:



Applying TOP yields:



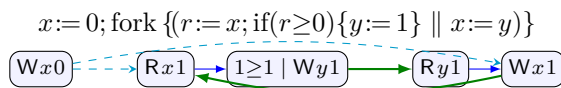
Although $(ff \Rightarrow ff)$ is a tautology, \downarrow^y is not. Thus, the pomset can only contribute to a top-level pomset if the thread is preceded by a relaxed write to y , allowing $(Ry1)$ to be downgraded to a relaxed read.

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.

6. Other Features

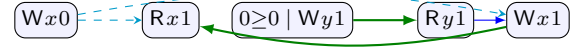
6.1. Local Invariant Reasoning (lir)

Ex 48. JMM causality Test Case 1 [31] states the following execution should be allowed “since interthread compiler analysis could determine that x and y are always non-negative, allowing simplification of $r \geq 0$ to true, and allowing write $y := 1$ to be moved early.”



Under the definitions given thus far, the precondition on $(Wy1)$ 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:

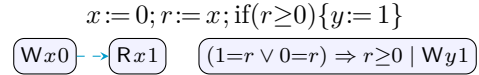
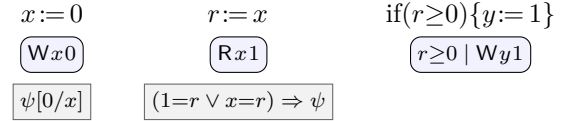


Def 49 (lir). Update Def 22 to:

- S4) $\tau^D(\psi)$ implies $\psi[M/x] \wedge M=v$,
- 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 $E = \emptyset$.

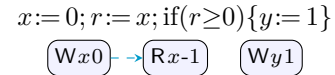
L5 introduces memory references. It states that to be independent of the read, we must establish both $\psi[v/r]$ and $\psi[x/r]$. If a precondition holds in both circumstances, S5 allows a local write to satisfy the precondition without introducing dependence. As in Def 44, we include L6 to provide a predicate transformer for the empty pomset.

Ex 50. Revisiting Ex 48 and eliding irrelevant transformers:



The precondition of $(Wy1)$ is a tautology, as required.

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



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 [31].

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

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

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 combining 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.

6.2. 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.

If $P \in \text{WRITE}(x, M, \mu)$ then $(\exists v : E \rightarrow \mathcal{V}) (\exists \theta : E \rightarrow \Phi)$

S1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,

S2) $\lambda(e) = \text{Wx}v_e$,

S3) $\kappa(e)$ implies $\theta_e \wedge Q_\mu^{Sx} \wedge M=v_e$,

S4) $(\forall e \in E \cap D) \tau^D(\psi)$ implies $\theta_e \Rightarrow (\psi[M/x][\mu/\downarrow^x] \wedge M=v_e)$,

S5) $(\forall e \in E \setminus D) \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/x][\mu/\downarrow^x][\text{ff}/Q_\mu^{Sx}]$.

If $P \in \text{READ}(r, x, \mu)$ then $(\exists v : E \rightarrow \mathcal{V}) (\exists \theta : E \rightarrow \Phi)$

L1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,

L2) $\lambda(e) = \text{Rx}v_e$,

L3) $\kappa(e)$ implies $\theta_e \wedge Q_\mu^{Lx}$,

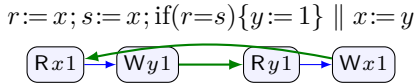
L4) $(\forall e \in E \cap D) \tau^D(\psi)$ implies $\theta_e \Rightarrow (v_e=s_e) \Rightarrow \psi[s_e/r]$,

L5) $(\forall e \in E \setminus D) \tau^D(\psi)$ implies $\theta_e \Rightarrow \downarrow_\mu^x \wedge (\langle v_e=s_e \vee (W \wedge x=s_e) \rangle \Rightarrow \psi[s_e/r][\text{ff}/Q_\mu^{Lx}])$,

L6) $(\forall s) \tau^D(\psi)$ implies $(\exists e \in E \mid \theta_e) \Rightarrow (\downarrow_\mu^x \wedge \psi[s/r][\text{ff}/Q_\mu^{Lx}])$.

Figure 1: Full Semantics of Loads and Stores (See Def 31 for Q_μ^{Sx} , Q_μ^{Lx} and Def 39 for \downarrow_μ^x , $[\mu/\downarrow^x]$)

Ex 51. JMM causality Test Case 2 [31] states the following execution should be allowed “since redundant read elimination could result in simplification of $r=s$ to true, allowing $y:=1$ to be moved early.”



This execution is not allowed under Def 49, since the precondition of (Wy1) 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 §3.1. This allows us to resolve nondeterminism in loads when merging, resulting in:

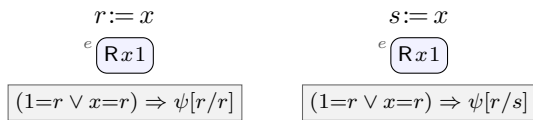


Def 52 (alpha). Update Def 22 to:

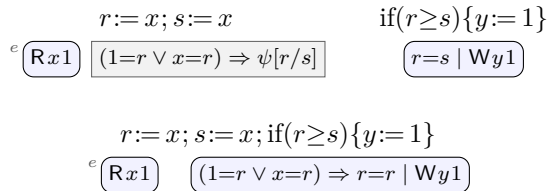
L4) $\tau^D(\psi)$ implies $v=s_e \Rightarrow \psi[s_e/r]$,

L5) $(\forall s) \tau^C(\psi)$ implies $\psi[s/r]$.

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



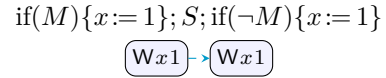
Coalescing and composing:



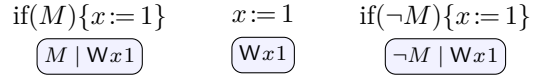
The precondition of (Wy1) is a tautology, as required.

6.3. If-Closure (if)

Ex 54. If $S = (x:=1)$, then Def 22 does not allow:



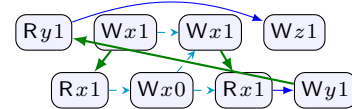
However, if $S = (\text{if}(\neg M)\{x:=1\}; \text{if}(M)\{x:=1\})$, then it does allow the execution. Looking at the initial program:



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

This is not simply a theoretical question; it is observable. For example, Def 22 does not allow the following.

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



Def 55 (alpha/if). Update Def 22 to:

If $P \in \text{WRITE}(x, M, \mu)$ then $(\exists v : E \rightarrow \mathcal{V}) (\exists \theta : E \rightarrow \Phi)$

S1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,

S2) $\lambda(e) = \text{Wx}v_e$,

S3) $\kappa(e)$ implies $\theta_e \wedge M=v$,

S4) $(\forall e \in E \cap D) \tau^D(\psi)$ implies $\theta_e \Rightarrow (\psi \wedge M=v)$,

S5) $\tau^C(\psi)$ implies $(\exists e \in E \cap C \mid \theta_e) \Rightarrow \psi$,

If $P \in \text{READ}(r, x, \mu)$ then $(\exists v : E \rightarrow \mathcal{V}) (\exists \theta : E \rightarrow \Phi)$

L1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,

L2) $\lambda(e) = \text{Rx}v_e$,

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 $(\exists e \in E \mid \theta_e) \Rightarrow \psi[s/r]$.

Ex 56. Revisiting Ex 54, we can split the middle command:

$$\text{if}(M)\{x:=1\} \quad x:=1 \quad \text{if}(\neg M)\{x:=1\}$$

$$\begin{array}{ccccc} \xrightarrow{d} & \xrightarrow{d} & \xrightarrow{e} & \xrightarrow{e} & \\ \boxed{M \mid Wx1} & \boxed{\neg M \mid Wx1} & \boxed{M \mid Wx1} & \boxed{\neg M \mid Wx1} & \end{array}$$

Coalescing events gives the desired result.

These examples show that we must allow inconsistent predicates in a single pomset, unlike [17].

7. Conclusions

We have presented the first model of relaxed memory that treats sequential composition as a first-class citizen. The model builds directly on [17].

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 §4–6. 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] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '11*, pages 55–66, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926394. URL <http://doi.acm.org/10.1145/1926385.1926394>.
- [5] H.-J. Boehm and B. Demsky. Outlawing ghosts: Avoiding out-of-thin-air results. In *Proceedings of the Workshop on Memory Systems Performance and Correctness, MSPC '14*, pages 7:1–7:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2917-0. doi: 10.1145/2618128.2618134. URL <http://doi.acm.org/10.1145/2618128.2618134>.
- [6] 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>.
- [7] 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>.
- [8] R. Cox. Go’s memory model. <http://nil.csail.mit.edu/6.824/2016/notes/gomem.pdf>, Feb. 2016.
- [9] 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>.
- [10] 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>.
- [11] 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>.
- [12] W. Ferreira, M. Hennessy, and A. Jeffrey. A theory of weak bisimulation for core CML. In R. Harper and R. L. Wexelblat, editors, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996*, Philadelphia, Pennsylvania, USA, May 24-26, 1996, pages 201–212. ACM, 1996. doi: 10.1145/232627.232649. URL <https://doi.org/10.1145/232627.232649>.
- [13] J. L. Gischer. The equational theory of pom-sets. *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>.
- [14] 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>.
- [15] R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In A. D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010*, Paphos, Cyprus, March 20-28, 2010. *Proceedings, volume 6012 of Lecture Notes in Computer Science*, pages 307–326. Springer, 2010. ISBN 978-3-642-11956-9. doi: 10.1007/978-3-642-11957-6_17. URL https://doi.org/10.1007/978-3-642-11957-6_17.
- [16] 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.
- [17] 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>.
- [18] A. Jeffrey and J. Riely. On thin air reads towards an event structures model of relaxed memory. In M. Grohe, E. Koskinen, and N. Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16*, New York, NY, USA, July 5-8, 2016, pages 759–767. ACM, 2016. ISBN 978-1-4503-4391-6. doi: 10.1145/2933575.2934536. URL <http://doi.acm.org/10.1145/2933575.2934536>.
- [19] 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 SIG-*

- PLAN 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>.
- [20] R. Kavanagh and S. Brookes. A denotational account of c11-style memory. CoRR, abs/1804.04214, 2018. URL <http://arxiv.org/abs/1804.04214>.
- [21] O. Lahav, V. Vafeiadis, J. Kang, C. Hur, and D. Dreyer. Repairing sequential consistency in C/C++11. In A. Cohen and M. T. Vechev, editors, Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, pages 618–632. ACM, 2017. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062352. URL <https://doi.org/10.1145/3062341.3062352>.
- [22] 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>.
- [23] S. Lee, M. Cho, A. Podkopaev, S. Chakraborty, C. Hur, O. Lahav, and V. Vafeiadis. Promising 2.0: global optimizations in relaxed memory concurrency. In A. F. Donaldson and E. Torlak, editors, Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020, pages 362–376. ACM, 2020. doi: 10.1145/3385412.3386010. URL <https://doi.org/10.1145/3385412.3386010>.
- [24] 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>.
- [25] 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>.
- [26] 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.
- [27] 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>.
- [28] 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>.
- [29] A. Podkopaev. Private correspondence, Nov. 2020.
- [30] A. Podkopaev, O. Lahav, and V. Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang., 3(POPL):69:1–69:31, 2019. doi: 10.1145/3290382. URL <https://doi.org/10.1145/3290382>.
- [31] W. Pugh. Causality test cases, 2004. URL <https://perma.cc/PJT9-XS8Z>.
- [32] C. Watt, A. Rossberg, and J. Pichon-Pharabod. Weakening webassembly. Proc. ACM Program. Lang., 3(OOPSLA):133:1–133:28, 2019. doi: 10.1145/3360559. URL <https://doi.org/10.1145/3360559>.
- [33] C. Watt, C. Pulte, A. Podkopaev, G. Barbier, S. Dolan, S. Flur, J. Pichon-Pharabod, and S. Guo. Repairing and mechanising the javascript relaxed memory model. In A. F. Donaldson and E. Torlak, editors, Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020, pages 346–361. ACM, 2020. doi: 10.1145/3385412.3385973. URL <https://doi.org/10.1145/3385412.3385973>.

Appendix A.

Additional Features

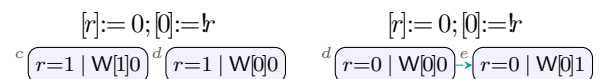
A.1. Address Calculation (addr)

Fig 3 describes the full semantics with address calculation.

Def 58 (addr). Update Def 22 to existentially quantify over ℓ in WRITE and READ:

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

Ex 59. Def 58 is naive with respect to merging events. Consider the following example from [17]:



If $P \in \text{WRITE}(L, M, \mu)$ then $(\exists \ell : E \rightarrow \mathcal{V}) (\exists v : E \rightarrow \mathcal{V}) (\exists \theta : E \rightarrow \Phi)$

S1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,

S2) $\lambda(e) = \mathbf{W}[\ell_e]v_e$,

S3) $\kappa(e)$ implies $\theta_e \wedge \mathbf{Q}_\mu^{S[\ell_e]} \wedge L = \ell_e \wedge M = v_e$,

S4) $(\forall e \in E \cap D) \tau^D(\psi)$ implies $\theta_e \Rightarrow (\psi[M/\ell_e][\mu/\downarrow^{\ell_e}] \wedge L = \ell_e \wedge M = v_e)$,

S5) $(\forall e \in E \setminus D) \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/\ell_e][\mu/\downarrow^{\ell_e}][\text{ff}/\mathbf{Q}_\mu^{S[\ell_e]}]$,

S6) $(\forall k) \tau^D(\psi)$ implies $(\exists e \in E \mid \theta_e) \Rightarrow (L = k) \Rightarrow \psi[M/k][\mu/\downarrow^k][\text{ff}/\mathbf{Q}_\mu^{S[k]}]$.

If $P \in \text{READ}(r, L, \mu)$ then $(\exists \ell : E \rightarrow \mathcal{V}) (\exists v : E \rightarrow \mathcal{V}) (\exists \theta : E \rightarrow \Phi)$

L1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,

L2) $\lambda(e) = \mathbf{R}[\ell_e]v_e$,

L3) $\kappa(e)$ implies $\theta_e \wedge \mathbf{Q}_\mu^{L[\ell_e]} \wedge L = \ell_e$,

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 $\theta_e \Rightarrow \downarrow_\mu^{\ell_e} \wedge (\langle (L = \ell_e \Rightarrow v_e = s_e) \vee (\mathbf{W} \wedge (L = \ell_e \Rightarrow [\ell_e] = s_e)) \rangle \Rightarrow \psi[s_e/r][\text{ff}/\mathbf{Q}_\mu^{L[\ell_e]}])$,

L6) $(\forall k)(\forall s) \tau^D(\psi)$ implies $(\exists e \in E \mid \theta_e) \Rightarrow (L = k) \Rightarrow (\downarrow_\mu^k \wedge \psi[s/r][\text{ff}/\mathbf{Q}_\mu^{L[k]}])$.

Figure 3: Full Semantics with Address Calculation (See Def 31 for \mathbf{Q}_μ^{Sx} , \mathbf{Q}_μ^{Lx} and Def 39 for \downarrow_μ^x , $[\mu/\downarrow^x]$)

Merging, we have:

$$\begin{array}{c} \text{if}(M)\{[r] := 0; [0] := r\} \text{ else } \{[r] := 0; [0] := r\} \\ \text{c} \quad \boxed{r=1 \mid \mathbf{W}[1]0} \quad \text{d} \quad \boxed{r=0 \vee r=1 \mid \mathbf{W}[0]0} \quad \text{e} \quad \boxed{r=0 \mid \mathbf{W}[0]1} \end{array}$$

The precondition of $\mathbf{W}[0]0$ 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:

$$\begin{array}{cc} [r] := 0 & [0] := r \\ \text{c} \quad \boxed{r=1 \mid \mathbf{W}[1]0} & \text{d} \quad \boxed{r=1 \mid \mathbf{W}[0]0} \\ \text{d} \quad \boxed{r=0 \mid \mathbf{W}[0]0} & \text{e} \quad \boxed{r=0 \mid \mathbf{W}[0]1} \end{array}$$

Sequencing and merging, we have:

$$\begin{array}{c} [r] := 0; [0] := r \\ \text{c} \quad \boxed{r=1 \mid \mathbf{W}[1]0} \quad \text{d} \quad \boxed{r=0 \vee r=1 \mid \mathbf{W}[0]0} \quad \text{e} \quad \boxed{r=0 \mid \mathbf{W}[0]1} \end{array}$$

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

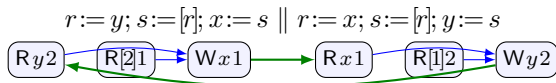
Ex 60. The combination of read-read independency and address calculation (addr/rrd) is somewhat delicate. Combining Def 44 and Def 58 we have:

L4) $\tau^D(\psi)$ implies $(L = \ell \Rightarrow v = r) \Rightarrow \psi$,

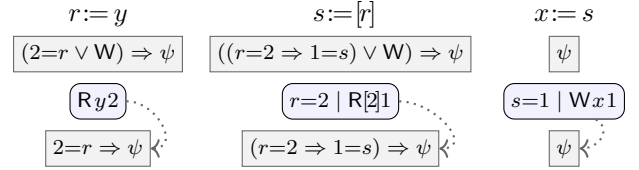
L5) $\tau^C(\psi)$ implies $((L = \ell \Rightarrow v = r) \vee \mathbf{W}) \Rightarrow \psi$.

If we replace the use of $(L = \ell \Rightarrow v = r)$ by $(v = r)$, thin air reads are possible. The subsection of §C on Causal Strengthening discusses this example using the semantics of [17].

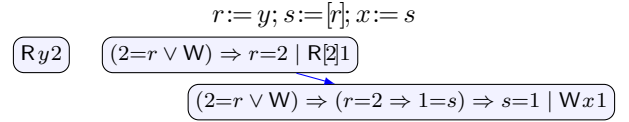
Consider the following program, from [17, §5], where initially $x = 0$, $y = 0$, $[0] = 0$, $[1] = 2$, and $[2] = 1$. It should only be possible to read 0, disallowing the attempted execution below:



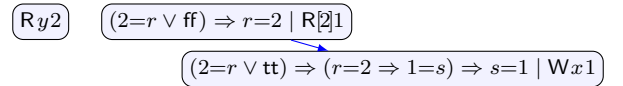
Looking at the left thread:



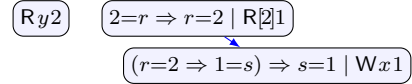
Composing, we have:



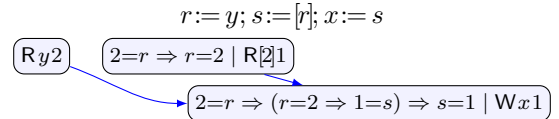
Substituting for W:



Which is:



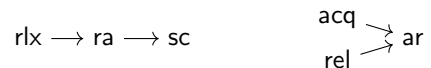
The precondition of $(\mathbf{R}[2]1)$ is a tautology, but the precondition of $(\mathbf{W}x1)$ is not. This forces a dependency:



All the preconditions are now tautologies.

A.2. Merging different labels

We combine access and fence modes into a single order:



Def 61. Define $\prec : \mathcal{A} \times \mathcal{A} \rightarrow 2^{\mathcal{A}}$ as follows. If $a_0 \in a_1 \prec a_2$, then a_1 and a_2 can coalesce, resulting in a_0 . This is useful for replacing $(x:=1; x:=2)$ by $(x:=2)$.

$$\begin{aligned} R x v \prec R x v &= \{R x v\} \\ W x v \prec W x v &= \{W x v\} \\ R x v \prec W x v &= \{W x v\} \\ R x v \prec W x v &= \{W x v\} \\ F^\mu \prec F^\nu &= \{F^{\mu \sqcup \nu}\} \\ a \prec b &= \emptyset, \text{ otherwise} \end{aligned}$$

- 1) if $e \in E_1 \setminus E_2$ then $\lambda(e) = \lambda_1(e)$,
- 2) if $e \in E_2 \setminus E_1$ then $\lambda(e) = \lambda_2(e)$,
- 3) if $e \in E_1 \cap E_2$ then $\lambda(e) \in \lambda_1(e) \prec \lambda_2(e)$, the first has no rf,

A.3. Read-Modify-Write Operations (rmw)

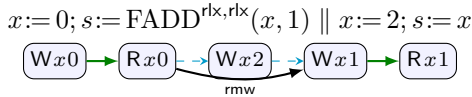
rmw operations are formalized by adding a relation $\xrightarrow{\text{rmw}} \subseteq E \times E$ that relates the read of a successful rmw to the succeeding write. The definition of a pomset requires the following, where two actions overlap if they access the same location:

- 1) $\text{rmw} \subseteq \leq$ is a relation capturing read-modify-write atomicity, such that for any $c, d, e \in E$, where $\lambda(c)$ and $\lambda(d)$ access the same location:
 - if $d \xrightarrow{\text{rmw}} e$ and $c \leq e$ then $c \leq d$,
 - if $d \xrightarrow{\text{rmw}} e$ and $d \leq c$ then $e \leq c$.

Extend the definition of par, if, seq to include:

- $\text{rmw} \supseteq (\text{rmw}_1 \cup \text{rmw}_2)$,

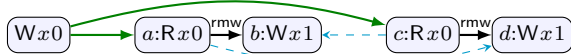
Ex 62. This definition ensures atomicity, disallowing executions such as [30, Ex. 3.2]:



By ??, since $(Wx2) \dashrightarrow (Wx1)$, it must be that $(Wx2) \dashrightarrow (Rx0)$, creating a cycle.

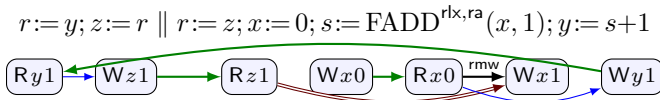
Ex 63. Two successful rmws cannot see the same write:

$$x := 0; (\text{FADD}^{\text{rlx}, \text{rlx}}(x, 1) \parallel \text{FADD}^{\text{rlx}, \text{rlx}}(x, 1))$$



The order from read-to-write is required by fulfillment. Apply ?? to $a \dashrightarrow d$, we have that $a \dashrightarrow c$. Subsequently applying ??, we have $b \dashrightarrow c$, creating a cycle.

Ex 64. By using two actions rather than one, the definition allows examples such as the following, which is allowed by arm8 [30, Ex. 3.10]:

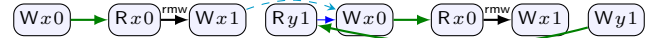


Ex 65. For rmw operations, the independent case for a read should be the same as the empty case. To see why, consider the semantics of local invariant reasoning (lir) from Def 49:

- S4) $\tau^D(\psi)$ implies $\psi[M/x] \wedge M=v$,
- 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 $E = \emptyset$.

Consider the relaxed variant of the cdrf example from [23], using a semantics for FADD that simply composes the rules for load and store above.

$$\begin{aligned} x := 0; (r := \text{FADD}^{\text{rlx}, \text{rlx}}(x, 1); \text{if}(\mathfrak{b})\{x := 0\}) \parallel \\ r := \text{FADD}^{\text{rlx}, \text{rlx}}(x, 1); \text{if}(\mathfrak{b})\{y := 1\} \end{aligned}$$



Looking at the independent transformers of the second thread and initializer, we have:

$$\begin{array}{ccc} x := 0 & \text{FADD}^{\text{rlx}, \text{rlx}}(x, 1) & \text{if}(\mathfrak{b})\{y := 1\} \\ \boxed{Wx0} & \boxed{Rx0 \xrightarrow{\text{rmw}} Wx1} & \boxed{r=0 \mid Wy1} \\ \boxed{\psi[0/x]} & \boxed{(0=r \vee x=r) \Rightarrow \psi[1/x]} & \boxed{\psi[1/y]} \end{array}$$

After sequencing, the precondition of $(Wy1)$ is a tautology: $(0=r \vee 0=r) \Rightarrow r=0$.

Here, local invariant reasoning is using the initializing write to x to justify the independence of the write to y . But this write is made unavailable by the first thread's successful rmw.

As a result, we disallow the use of L5 when treating the read event in an rmw.

[Todo: write out the rules.]

A.4. Fence Operations (fence)

Syntactic fences F^ν have corresponding actions: (F^ν) . The syntactic fence mode ($\nu ::= \text{rel} \mid \text{acq} \mid \text{ar}$) is either release, acquire, or sequentially-consistent.

Formalizing this, Q_{rel}^{Sx} substitutes for Q_{wo}^* in addition to Q_{wo}^x , as in Q_{ra}^{Sx} . Q_{acq}^{Lx} requires Q_{ro}^* in addition to Q_{wo}^x , as in Q_{ra}^{Lx} .

Def 66. Extend Def 31 and Def 39.

$$\begin{aligned} Q_{\text{rel}}^{Fx} &= Q_{\text{ro}}^* \wedge Q_{\text{wo}}^* & Q_{\text{acq}}^{Fx} &= Q_{\text{wo}}^x \wedge Q_{\text{ro}}^* \\ [\phi/Q_{\text{rel}}^{Fx}] &= [\phi/Q_{\text{wo}}^*] & [\phi/Q_{\text{acq}}^{Fx}] &= [\phi/Q_{\text{ro}}^*, \phi/Q_{\text{wo}}^*] \end{aligned}$$

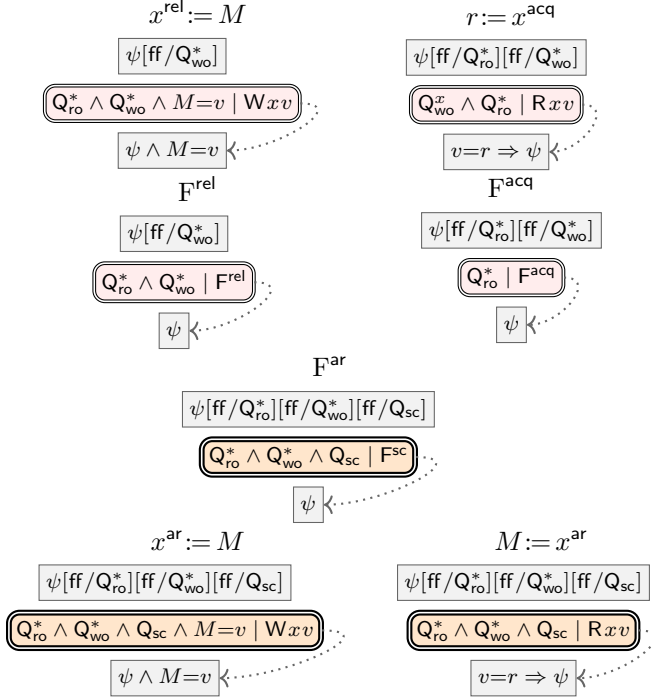
$$\begin{aligned} Q_{\text{ar}}^{Fx} &= Q_{\text{ro}}^* \wedge Q_{\text{wo}}^* \wedge Q_{\text{sc}} \\ [\phi/Q_{\text{ar}}^{Fx}] &= [\phi/Q_{\text{ro}}^*, \phi/Q_{\text{wo}}^*, \phi/Q_{\text{sc}}] \end{aligned}$$

If $P \in \text{FENCE}(\mu, \sigma)$ then $(\exists X \subseteq \mathcal{X})$

- 2) if $d, e \in E$ then $d = e$,
- 3) $\lambda(e) = F_\sigma^\mu X$,
- 4) $\kappa(e)$ implies $\bigwedge_{x \in (\mathcal{X} \setminus X)} Q_\mu^{Fx}$,
- 5) $\tau^D(\psi)$ implies ψ , where $D \cap E \neq \emptyset$,

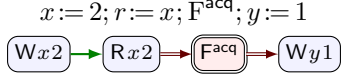
6) $\tau^C(\psi)$ implies $\bigwedge_{x \in X} \downarrow^x \wedge \psi[\text{ff}/Q^{Fx}]$, where $C \cap E = \emptyset$.

Ex 67. Extend Ex 33.



A.5. Fence Actions with Downgrading Reads (fence/dgr)

Ex 68. Revisiting Ex 41 using fences:



What we want is this:



Let acquiring fence actions include a set: $(F^\nu X)$. The set is nondeterministically chosen by the semantics. Idea is to downgrade the reads in X and fence everything else.

A.6. Access Elimination

For reads, get rid of ff/Q in L6.

For writes, change the label rules of sequential composition to:

- 1) if $e \in E_1 \setminus E_2$ then $\lambda(e) = \lambda_1(e)$,
- 2) if $e \in E_2 \setminus E_1$ then $\lambda(e) = \lambda_2(e)$,
- 3) if $e \in E_1 \cap E_2$ then $\lambda(e) \in \lambda_1(e) \prec \lambda_2(e)$.

Def 69.

$$\begin{aligned} R x v \prec R x v &= \{R x v\} \\ W x v \prec W x v &= \{W x v\} \\ F^\mu \prec F^\nu &= \{F^{\mu \sqcup \nu}\} \\ a \prec b &= \emptyset, \text{ otherwise} \end{aligned}$$

A.7. Extended Access Modes

We can enrich read and write actions to use fence modes. The resulting order is:

$$\text{rlx} \rightarrow \text{ra} \begin{matrix} \nearrow \text{acq} \\ \nearrow \text{sc} \\ \nearrow \text{rel} \end{matrix} \rightarrow \text{ar}$$

We write $\mu \sqsubseteq \nu$ for this order. Let $\mu \sqcup \nu$ denote the least upper bound of μ and ν .

Reads allow all annotations but rel . Writes allow all annotations but acq . Fences allow only the three annotations rel , acq and ar .

Def 70.

$$\begin{aligned} R^\mu x v \prec R^\nu x v &= \{R^{\mu \sqcup \nu} x v\} \\ W^\mu x v \prec W^\nu x v &= \{W^{\mu \sqcup \nu} x v\} \\ F^\mu \prec F^\nu &= \{F^{\mu \sqcup \nu}\} \\ F^\mu \prec R^\nu x v &= R^\mu x v \prec F^\nu = \{R^{\mu \sqcup \nu} x v\} \\ F^\mu \prec W^\nu x v &= W^\mu x v \prec F^\nu = \{W^{\mu \sqcup \nu} x v\} \\ a \prec b &= \emptyset, \text{ otherwise} \end{aligned}$$

Appendix B.

Discussion

B.1. Closure properties

We would like the semantics to 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 71. 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 72. If $P_1 \in \llbracket S \rrbracket$ and P_2 augments P_1 then $P_2 \in \llbracket S \rrbracket$.

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

Def 73. P_2 is a 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) $(\forall d \in E_1) (\forall e \in E_2) \text{ if } d \leq_1 e \text{ then } d \in E_2$.

Downset closure fails due to rrd and lir . The key property is that the empty set transformer should behave the same as the independent transformer.

Ex 74. For `rrd`, Def 44 states:

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

This semantics is not downset closed due to the lack of read-read dependencies. In both cases, for subsequent writes, L5 is the same as L6. For subsequent reads, L5 is the same as L4. Consider

$$\begin{array}{c} r := x; \text{if}(r) \{ s := y \} \\ \boxed{\text{Rx0}} \quad \boxed{\text{Ry0}} \end{array}$$

The semantics of this program includes the singleton pomset (Rx0), but not the singleton pomset (Ry0). To get (Rx0), we combine:

$$\begin{array}{c} r := x \quad \text{if}(r) \{ s := y \} \\ \boxed{\text{Rx0}} \quad \emptyset \end{array}$$

Attempting to get (Ry0), we instead get:

$$\begin{array}{c} r := x \quad \text{if}(r) \{ s := y \} \\ \emptyset \quad \boxed{r=0 \mid \text{Ry0}} \end{array}$$

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

Ex 75. For `lir`, Def 49 states:

- 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 $E = \emptyset$.

This semantics is not downset closed: The independency reasoning of L5 is only applicable for pomsets where the ignored read is present! Revisiting Ex 50

$$\begin{array}{ccc} x := 0 & r := x & \text{if}(r \geq 0) \{ y := 1 \} \\ \boxed{\text{Wx0}} & \boxed{\text{Rx1}} & \boxed{r \geq 0 \mid \text{Wy1}} \\ \boxed{\psi[0/x]} & \boxed{(1=r \vee x=r) \Rightarrow \psi} & \end{array}$$

$$\begin{array}{c} x := 0; r := x; \text{if}(r \geq 0) \{ y := 1 \} \\ \boxed{\text{Wx0}} \rightarrow \boxed{\text{Rx1}} \quad \boxed{(1=r \vee 0=r) \Rightarrow r \geq 0 \mid \text{Wy1}} \end{array}$$

The precondition of (Wy1) is a tautology.

Taking the empty set for the read, however, we have:

$$\begin{array}{c} x := 0; r := x; \text{if}(r \geq 0) \{ y := 1 \} \\ \boxed{\text{Wx0}} \quad \boxed{r \geq 0 \mid \text{Wy1}} \end{array}$$

The precondition of (Wy1) is not a tautology.

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 §3–5.

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 $\phi \Rightarrow \text{wp}_S(\psi)$.

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:

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

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:

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

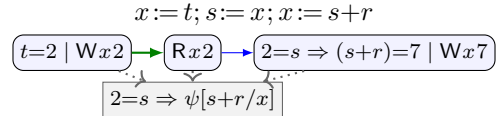
Only the read rule differs from the traditional one.

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:

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

If $P \in \text{WRITE}(x, M, \mu)$ then $(\exists v : E \rightarrow \mathcal{V}) (\exists \theta : E \rightarrow \Phi)$

S1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
S2) $\lambda(e) = W^\mu x v_e$,
S3) $\kappa(e)$ implies $\theta_e \wedge M = v_e$,
S4) $(\forall e \in E \cap D) \tau^D(\psi)$ implies $\theta_e \Rightarrow (\psi[M/x][\mu/\downarrow^x] \wedge M = v_e)$,
S5) $(\forall e \in E \setminus D) \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/x][\mu/\downarrow^x][\text{ff}/Q]$.
S6) $\tau^D(\psi)$ implies $(\exists e \in E \mid \theta_e) \Rightarrow \psi[M/x][\mu/\downarrow^x][\text{ff}/Q]$.

If $P \in \text{READ}(r, x, \mu)$ then $(\exists v : E \rightarrow \mathcal{V}) (\exists \theta : E \rightarrow \Phi) (\exists \nu \in \{\mu, \text{rlx}\})$

L1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
L2) $\lambda(e) = R^\nu x v_e$,
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 $\theta_e \Rightarrow \downarrow_{\mu, \nu}^x \wedge (\langle v_e = s_e \vee (W \wedge x = s_e) \rangle \Rightarrow \psi[s_e/r][\text{ff}/Q])$,
L6) $(\forall s) \tau^D(\psi)$ implies $(\exists e \in E \mid \theta_e) \Rightarrow (\downarrow_{\mu, \nu}^x \wedge \psi[s/r][\text{ff}/Q])$.

Figure 4: Simplified Quiescence Semantics w/o Address Calculation (See Def 79 for $\downarrow_{\mu, \nu}^x$, $[\mu/\downarrow^x]$)

B.3. Coherence/Synchronization via Reordering

In §4.2, we encoded coherence and synchronized access using quiescence symbols. Building on the language with fork-join, it is possible to model these using reorderability (§3.2), rather than encoding them in the logic. With synchronization, the relationship becomes asymmetric.

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

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

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

It seems that fences generally do not commute, except for read-motel.

	2 nd								
1 st	R ^{rlx}	R ^{ra}	R ^{sc}	W ^{rlx}	W ^{ra}	W ^{sc}	F ^{rel}	F ^{acq}	F ^{ar}
R ^{rlx}	✓	✓	✓	✓	✗	✗	✗	✗	✗
R ^{ra}	✗	✗	✗	✗	✗	✗	✗	✗	✗
R ^{sc}	✗	✗	✗	✗	✗	✗	✗	✗	✗
W ^{rlx}	✓	✓	✓	✓	✗	✗	✗	✓	✗
W ^{ra}	✓	✓	✓	✓	✗	✗	✗	✓	✗
W ^{sc}	✓	✓	✗	✓	✗	✗	✗	✓	✗
F ^{rel}	✓	✓	✓	✗	✗	✗	✗	✓	✗
F ^{acq}	✗	✗	✗	✗	✗	✗	✗	✗	✗
F ^{ar}	✗	✗	✗	✗	✗	✗	✗	✗	✗

Def 76. Including coherence, reorderability is defined:

$$\begin{aligned}
\bowtie_{\text{loc}} &= \{(Wx, Ry) \mid x \neq y\} \cup \{(Wx, Wy) \mid x \neq y\} \\
&\quad \cup \{(Rx, Wy) \mid x \neq y\} \cup \{(Rx, Ry)\} \\
\bowtie_{\text{sync}} &= \{(W^\mu, R^\nu) \mid \mu \neq \text{sc} \vee \nu \neq \text{sc}\} \cup \{(W^\mu, W^{\text{rlx}}) \\
&\quad \cup \{(R^\mu, W^\nu) \mid \mu = \text{rlx} \wedge \nu = \text{rlx}\} \cup \{(R^{\text{rlx}}, R^\nu)\} \\
&\quad \cup \{(F^{\text{rel}}, F^{\text{acq}})\} \cup \{(F^{\text{rel}}, R^\nu)\} \cup \{(W^\mu, F^{\text{acq}})\} \\
\bowtie &= \bowtie_{\text{sync}} \cap \bowtie_{\text{loc}}
\end{aligned}$$

Here is the version with generalized modes for read and write:

$$\begin{aligned}
\bowtie_{\text{sync}} &= \{(W^\mu, R^\nu) \mid \mu \not\supseteq \text{sc} \vee \nu \not\supseteq \text{sc}\} \cup \{(W^\mu, W^{\text{rlx}} \mid \mu \not\supseteq \text{rel}\} \\
&\quad \cup \{(R^\mu, W^\nu) \mid \mu = \text{rlx} \wedge \nu = \text{rlx}\} \cup \{(R^{\text{rlx}}, R^\nu) \mid \nu \not\supseteq \text{acq}\} \\
&\quad \cup \{(F^{\text{rel}}, F^{\text{acq}})\} \cup \{(F^{\text{rel}}, R^\nu)\} \cup \{(W^\mu, F^{\text{acq}})\}
\end{aligned}$$

	2 nd									
1 st	R ^{rlx}	R ^{ra}	R ^{acq}	R ^{sc}	R ^{ar}	W ^{rlx}	W ^{ra}	W ^{rel}	W ^{sc}	W ^{ar}
R ^{rlx}	✓	✓	✗	✓	✗	✓	✗	✗	✗	✗
R ^{ra}	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
R ^{acq}	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
R ^{sc}	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
R ^{ar}	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
W ^{rlx}	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗
W ^{ra}	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗
W ^{rel}	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗
W ^{sc}	✓	✓	✓	✗	✗	✓	✗	✗	✗	✗
W ^{ar}	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Def 77. 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 22,

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

Update the read and write rules of Def 22 to:

- S2) $\lambda(e) = W^\mu x v$,
- S3) $\kappa(e)$ implies $M = v$,
- S4) $\tau^D(\psi)$ implies $\psi \wedge M = v$,
- S5) $\tau^C(\psi)$ implies $\psi[\text{ff}/Q]$,
- L2) $\lambda(e) = R^\mu x v$,
- L3) $\kappa(e)$ implies tt .
- L4) $\tau^D(\psi)$ implies $v = r \Rightarrow \psi$,
- L5) $\tau^C(\psi)$ implies $\psi[\text{ff}/Q]$.

Ex 78. The logic of quiescence is greatly simplified. Compare the following to Ex 33.

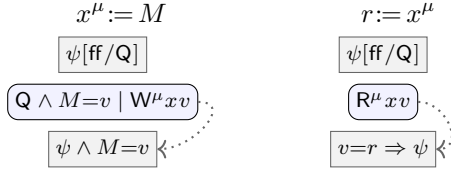


Fig 4 shows the resulting full semantics of read and write, without address calculation. Fig 5 shows the resulting full semantics with address calculation.

Def 79. Let substitution $[\mu/\downarrow^x]$ be defined:

$$[\mu/\downarrow^x] = \begin{cases} [\text{tt}/\downarrow^x] & \text{if } \mu = \text{rlx} \\ [\text{ff}/\downarrow^*] & \text{otherwise} \end{cases}$$

Let formula $\downarrow_{\mu,\nu}^x$ be defined:

$$\downarrow_{\mu,\nu}^x = \begin{cases} \text{tt} & \text{if } \mu = \text{rlx} \text{ or } \mu = \nu \\ \downarrow^x & \text{otherwise} \end{cases}$$

B.4. Substitutions

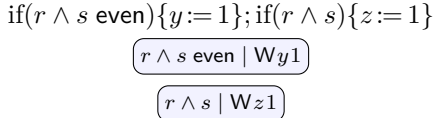
Recall the load rules from §6.1:

- 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 $E = \emptyset$.

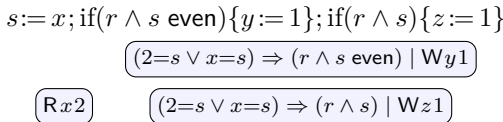
It is also possible to collapse x and r when doing a load:

- 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 $E = \emptyset$.

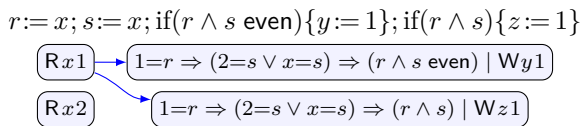
Perhaps surprisingly, these two semantics are incomparable. Consider the following:



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:

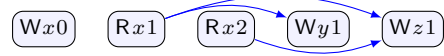


Prepending $(r:=x)$, we now get different results since the preconditions mention x . Without substitution:

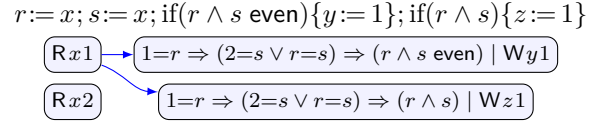


Prepending $(x:=0)$, which substitutes $[0/x]$, the precondition of $(Wy1)$ becomes $(1=r \Rightarrow (2=s \vee 0=s) \Rightarrow$

$(r \wedge s \text{ even}))$, which is a tautology, whereas the precondition of $Wz1$ becomes $(1=r \Rightarrow (2=s \vee 0=s) \Rightarrow (r \wedge s))$, which is not. In order to be top-level, $Wz1$ must depend on $Rx2$; 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 $[r/x]$:



Prepending $(x:=0)$:



The dependency has changed from $(Rx2) \rightarrow (Wz1)$ to $(Rx2) \rightarrow (Wy1)$. 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. [17] uses substitution rather than Skolemizing. Indeed our use of Skolemization is motivated by disjunction closure for predicate transformers, which do not appear in [17]; see §3.4.

In §6.1, 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 [17], 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 §6.1. Without Skolemizing it is necessary to substitute $[x/r]$, since the reverse substitution $[r/x]$ is useless when r is bound—compare with §B.4.

Including this substitution affects the interaction of local invariant reasoning and downset closure—see the following discussion of downset closure.

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

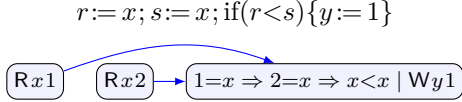
If $P \in \text{WRITE}(L, M, \mu)$ then $(\exists \ell : E \rightarrow \mathcal{V}) (\exists v : E \rightarrow \mathcal{V}) (\exists \theta : E \rightarrow \Phi)$
S1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
S2) $\lambda(e) = W^\mu[\ell_e]v_e$,
S3) $\kappa(e)$ implies $\theta_e \wedge L = \ell_e \wedge M = v_e$,
S4) $(\forall e \in E \cap D) \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/\ell_e][\mu/\downarrow^{\ell_e}] \wedge M = v_e \wedge L = \ell_e$,
S5) $(\forall e \in E \setminus D) \tau^D(\psi)$ implies $\theta_e \Rightarrow \psi[M/\ell_e][\mu/\downarrow^{\ell_e}][\text{ff}/Q]$,
S6) $(\forall k) \tau^D(\psi)$ implies $(\exists e \in E \mid \theta_e) \Rightarrow (L = k) \Rightarrow \psi[M/k][\mu/\downarrow^k][\text{ff}/Q]$.
If $P \in \text{READ}(r, L, \mu)$ then $(\exists \ell : E \rightarrow \mathcal{V}) (\exists v : E \rightarrow \mathcal{V}) (\exists \theta : E \rightarrow \Phi) (\exists \nu \in \{\mu, \text{rx}\})$
L1) if $\theta_d \wedge \theta_e$ is satisfiable then $d = e$,
L2) $\lambda(e) = R^\nu[\ell_e]v_e$,
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 $\theta_e \Rightarrow \downarrow_{\mu, \nu}^{\ell_e} \wedge (\langle (L = \ell_e \Rightarrow v_e = s_e) \vee (W \wedge (L = \ell_e \Rightarrow [\ell_e] = s_e)) \rangle \Rightarrow \psi[s_e/r][\text{ff}/Q])$,
L6) $(\forall k)(\forall s) \tau^D(\psi)$ implies $(\exists e \in E \mid \theta_e) \Rightarrow (L = k) \Rightarrow (\downarrow_{\mu, \nu}^k \wedge \psi[s/r][\text{ff}/Q])$.

Figure 5: Simplified Quiescence Semantics with Address Calculation (See Def 79 for $\downarrow_{\mu, \nu}^x, [\mu/\downarrow^x]$)

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:

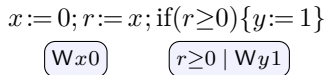
L4) $\tau^D(\psi)$ implies $v = x \Rightarrow \psi[x/r]$.

Using this definition, consider the following:



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

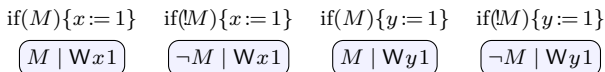
Downset closure. [17] enforces downset closure in the prefixing rule. Even without this, downset closure would be different for the two semantics, due to the use of substitution in [17]. Consider the final pomset of Ex 75, under the semantics of this paper, which elides the middle read event:



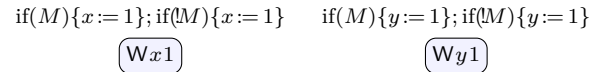
In [17], the substitution $[x/r]$ is performed by the middle read regardless of whether it is included in the pomset, with the subsequent substitution of $[0/x]$ by the preceding write, we have $[x/r][0/x]$, which is $[0/r][0/x]$, resulting in:



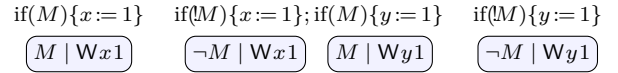
Consistency. [17] imposes consistency, which requires that for every pomset P , $\bigwedge_e \kappa(e)$ is satisfiable. Associativity requires that we allow pomsets with inconsistent preconditions. Consider a variant of Ex 54 from §6.3.



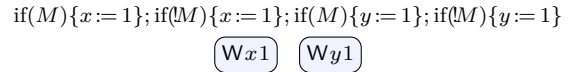
Associating left and right, we have:



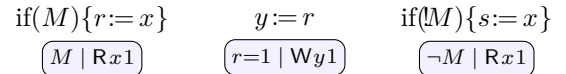
Associating into the middle, instead, we require:



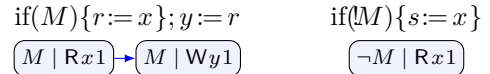
Joining left and right, we have:



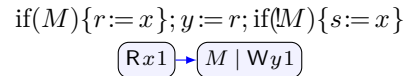
Causal Strengthening. Causal Strengthening [17] imposes causal strengthening, which requires for every pomset P , if $d \leq e$ then $\kappa(e)$ implies $\kappa(d)$. Associativity requires that we allow pomsets without causal strengthening. Consider the following.



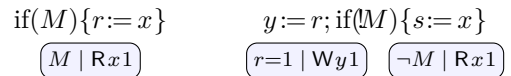
Associating left, with causal strengthening:



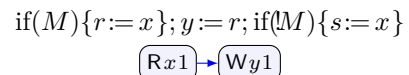
Finally, merging:



Instead, associating right:

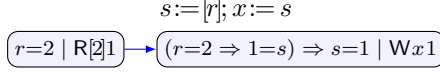


Merging:

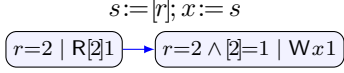


With causal strengthening, the precondition of $Wy1$ depends upon how we associate. This is not an issue in [17], which always associates to the right.

One use of causal strengthening is to ensure that address dependencies do not introduce thin air reads. Associating to the right, the intermediate state of Ex 60 is:



In [17], we have, instead:



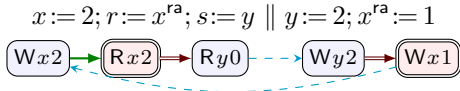
Without causal strengthening, the precondition of $(Wx1)$ would be simply $[2]=1$. The treatment in this paper, using implication rather than conjunction, is more precise.

Parallel Composition. In [17, §2.4], parallel composition is defined allowing coalescing of events. Here we have forbidden coalescing. This difference appears to be arbitrary. In [17], however, there is a mistake in the handling of termination actions. The predicates should be joined using \wedge , not \vee .

Read-Modify-Write Actions. In [17], the atomicity axioms $??/??$ erroneously applies only to overlapping writes, not overlapping reads. The difficulty can be seen in Ex 63.

[17] does not specify the calculation of dependency for rmws, as discussed in Ex 65.

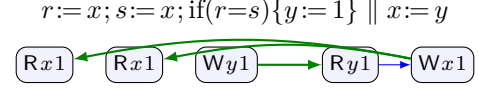
Downgrading Internal Acquiring Reads. Shortly after publication, Podkopaev [29] noticed a shortcoming of the implementation on arm8 in [17, §7]. The proof given there assumes that all internal reads can be dropped. However, this is not the case for acquiring reads. For example, [17] disallows the following execution, which is allowed by arm8 and tso.



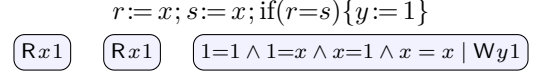
The solution we have adopted is to allow an acquiring read to be downgraded to a relaxed read when it is preceded (sequentially) by a relaxed write that could fulfill it. This solution allows executions that are not allowed under arm8 since we do not insist that the local relaxed write is actually read 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 [17]. We discussed redundant read elimination in §6.2. Consider JMM Causality Test Case 2, which we discussed there.



Under the semantics of [17], we have

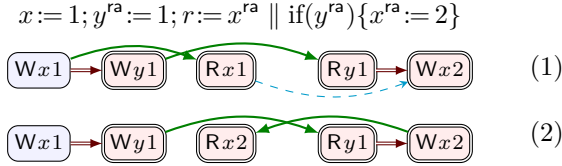


The precondition of $(Wy1)$ is not a tautology, and therefore redundant read elimination fails. (It is a tautology in $r := x; s := r; \text{if}(r=s)\{y := 1\}$.) In [17, §3.1], we incorrectly stated that the precondition of $(Wy1)$ was $1=1 \wedge x=x$.

Appendix D. More Stuff

D.1. 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 [11].



The program is data-race free. The two executions shown are the only top-level executions that include (Wx2).

Theorem 4.1 of [11] is stated by extending execution sequences. In the terminology of [11], a read is *L*-weak if it is sequentially stale. Let $\rho = (\text{Wx1})(\text{Wy1})(\text{Ry1})(\text{Wx2})$ be a sequence and $\alpha = (\text{Rx1})$. 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 [11], 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 (Rx1) is stale, but the program is data race free.

In proving the SC-LDRF result in [17, §8], we noted that our proof technique is more robust than that of [11], because it limits the prefixes that must be considered. In (1), the induction hypothesis requires that we add (Rx1) before (Wx2) since $(\text{Rx1}) \dashrightarrow (\text{Wx2})$. In particular,



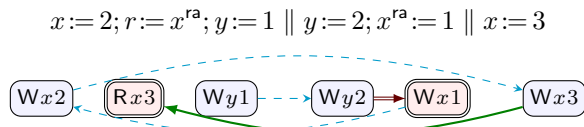
is not a downset of (1), because $(\text{Rx1}) \dashrightarrow (\text{Wx2})$. As we noted in [17, §8], this affects the inductive order in which we move across pomsets, but does not affect the set of pomsets that are considered. In particular,



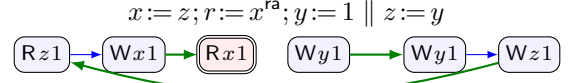
is a downset of (1).

D.2. Downgraded Reads

We allow downgrades in executions that are not be allowed by arm8.



arm8 disallows this because the acquiring read is fulfilled by an external write.



arm8 disallows this because data and control dependencies change acquiring read is fulfilled by an external write.

D.3. If Closure and Address Dependencies

An optimization (p/q are registers):

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

vs

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

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

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

D.4. About ARM

Hypothesis: gcb cannot contradict (poloc minus RxR).

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 Q_{wo}^x and Q_{ro}^x . Previously, we had given the semantics of *ra* access using Q_{wo}^* and Q_{ro}^* , which were encoded using Q_{wo}^x and Q_{ro}^x . With these gone, we introduce the quiescence symbol Q^* and Q_{acq} . Thus, the only quiescence symbols required are Q^* , Q_{acq} and Q_{sc} . Fig 6 shows the difference with the semantics of §4.2.

Def 80. Let formulae Q_μ^S and Q_μ^L be defined:

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

Let substitutions $[\phi/Q_\mu^S]$ and $[\phi/Q_\mu^L]$ be defined:

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

Def 81. Update Def 22 to:

- S3) $\kappa(e)$ implies $Q_\mu^S \wedge M=v$,
- L3) $\kappa(e)$ implies Q_μ^L ,
- S4) $\tau^D(\psi)$ implies $\psi \wedge M=v$,
- S5) $\tau^C(\psi)$ implies $\psi[\text{ff}/Q_\mu^S]$,
- L4) $\tau^D(\psi)$ implies $v=r \Rightarrow \psi$,

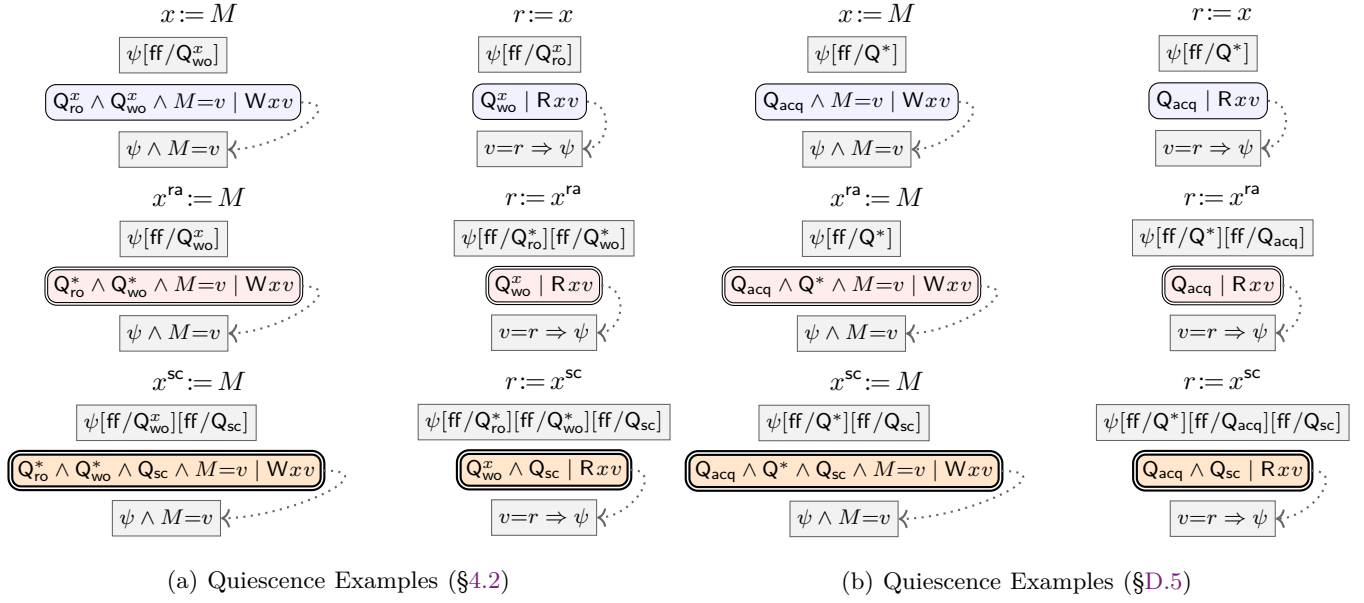
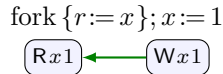


Figure 6: Quiescence Examples for Coherence

L5) $\tau^C(\psi)$ implies $\psi[\text{ff}/Q_\mu^L]$.

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

Item 10 of Def 77 ensures coherence. This definition is incompatible with asynchronous fork parallelism of Def ??, where we expect executions such as:



Item 10 would require $(Rx1) \dashrightarrow (Wx1)$, forbidding this.