A Unified Memory Model for Heterogenous Systems

ANONYMOUS AUTHOR(S)

ACM Reference Format:
Anonymous Author(s). 2022. A Unified Memory Model for Heterogenous Systems. *Proc. ACM Program. Lang.* 0, POPL, Article 0 (January 2022), 14 pages.

1 MODEL

1.1 Preliminaries

The syntax is built from

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

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

- values and registers are disjoint,
- values include at least the constants 0 and 1,
- expressions include at least registers and values,
- references do not appear in expressions: $M[N/x] = M$,
- thread ids include the *top-level* id $\mathbf{0}$.

We model the following language.

$$\mu, \nu ::= \mathsf{wk} \mid \mathsf{rlx} \mid \mathsf{rel} \mid \mathsf{acq} \mid \mathsf{ra} \mid \mathsf{sc} \qquad\qquad \sigma, \rho ::= \mathsf{cta} \mid \mathsf{gpu} \mid \mathsf{sys}$$

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

Access modes, μ, are weak (wk), relaxed (rlx), release (rel), acquire (acq), release-acquire (ra), and sequentially consistent (sc). Let expressions ($r := M$) only affect thread-local state and thus do not have a mode. Reads ($r := [L]^{\mu}_{\sigma}$) support wk, rlx, acq, sc. Writes ($[L]^{\mu}_{\sigma} := r$) support wk, rlx, rel, sc. Fences ($\mathsf{F}^{\mu}_{\sigma}$) support rel, acq, ra, sc. In the atomic update operations, μ is a read and ν is a write; we require that r does not occur in L.

Scopes, σ, are thread group (cta), processor (gpu) and system (sys).

Commands, aka *statements*, S, include memory accesses at a given mode, as well as the usual structural constructs. Following [Ferreira et al. 1996], $\overline{\|}$ denotes parallel composition. If ($S_1 \overline{\|}_{\gamma} S_2$) is executed with thread ID α, then S_2 runs with ID γ and S_1 continues under ID α. Top level programs run with thread ID $\mathbf{0}$. In examples, we usually drop thread IDs. We use the symmetric $\|$ operator when there is no continuation after the parallel composition.

We use common syntax sugar, such as *extended expressions*, $\mathbb{M}$, which include memory locations. For example, if $\mathbb{M}$ includes a single occurrence of x, then $y := \mathbb{M}; S$ is shorthand for $r := x;$

2475-1421/2022/1-ART0
https://doi.org/

$y := \mathbb{M}[r/x]$; S. Each occurrence of x in an extended expression corresponds to an separate read. We also write `if`$(M)\{S\}$ as shorthand for `if`$(M)\{S\}$`else{skip}`.

The semantics is built from the following.

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

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

- registers include $\mathcal{S}_{\mathcal{E}} = \{s_e \mid e \in \mathcal{E}\}$ which do not appear in commands: $S[N/s_e] = S$,
- formulae include equalities $(M{=}N)$ and $(x{=}M)$,
- formulae are closed under negation, conjunction, disjunction, and substitutions $[M/r]$, $[M/x]$,
- there is a relation $\vDash$ between formulae, capturing entailment,
- $\vDash$ has the expected semantics for $=$, $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and substitution.

We relax the first assumption in examples, assuming that each register is assigned at most once.

Logical formulae include equations over registers, such as $(r{=}s{+}1)$. For LIR, 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 is a *tautology* if $\mathsf{tt} \vDash \phi$. We say ϕ is *unsatisfiable* if $\phi \vDash \mathsf{ff}$.

We also require that there are subsets of actions, distinguishing *read* and *release* actions. We require several binary relations between actions, detailed in the next subsection: *overlaps*, *strongly-overlaps*, *matches*, *strongly-matches*, *strongly-fences*, *blocks*, *sync-delays* and *co-delays*. We require that strongly-overlaps implies overlaps and that strongly-matches implies matches implies blocks implies overlaps.

1.2 Actions

We combine access and fence modes into a single order:
$$\begin{tikzcd} & & \mathsf{rel} \arrow[dr] & & \\ \mathsf{wk} \arrow[r] & \mathsf{rlx} \arrow[ur] \arrow[dr] & & \mathsf{ra} \arrow[r] & \mathsf{sc} \\ & & \mathsf{acq} \arrow[ur] & & \end{tikzcd}$$
We write $\mu \sqsubseteq \nu$ for this order. Let $\mu \sqcup \nu$ denote the least upper bound of μ and ν.

Let actions be reads, writes and fences:

$$a, b \ ::= \ \alpha\mathsf{W}^{\mu}_{\sigma}xv \ \mid \ \alpha\mathsf{R}^{\mu}_{\sigma}xv \ \mid \ \alpha\mathsf{F}^{\mu}_{\sigma}$$

In examples, we systematically drop the default mode rlx and the default scope sys. In definitions, we drop elements of actions that are existentially quantified. We write $(\alpha\mathsf{A}^{\mu}_{\sigma}x)$ to stand for an *access*: either $(\alpha\mathsf{W}^{\mu}_{\sigma}x)$ or $(\alpha\mathsf{R}^{\mu}_{\sigma}x)$. We write $(\mathsf{W}^{\sqsupseteq\mathsf{rel}})$ to stand for either $(\mathsf{W}^{\mathsf{rel}})$ or $(\mathsf{W}^{\mathsf{sc}})$, and similarly for other actions and modes.

We say a *matches* b if $a = (\mathsf{W}xv)$ and $b = (\mathsf{R}xv)$.

We say a *blocks* b if $a = (\mathsf{W}x)$ and $b = (\mathsf{R}x)$, regardless of value.

We say a *overlaps* b if $a = (\mathsf{A}x)$ and $b = (\mathsf{A}x)$, regardless of access type or value.

We say a *co-delays* b if $(a, b) \in \{(\mathsf{W}x, \mathsf{W}x),\ (\mathsf{R}x, \mathsf{W}x),\ (\mathsf{W}x, \mathsf{R}x)\} \cup \{(\mathsf{A}^{\mathsf{sc}}, \mathsf{A}^{\mathsf{sc}})\}$.

We say a *sync-delays* b if $(a, b) \in \{(a, \mathsf{W}^{\sqsupseteq\mathsf{rel}}),\ (a, \mathsf{F}^{\sqsupseteq\mathsf{rel}}),\ (\mathsf{R}, \mathsf{F}^{\sqsupseteq\mathsf{acq}}),\ (\mathsf{R}^{\sqsupseteq\mathsf{acq}}, b),\ (\mathsf{F}^{\sqsupseteq\mathsf{acq}}, b),\ (\mathsf{F}^{\sqsupseteq\mathsf{rel}}, \mathsf{W}),\ (\mathsf{W}^{\sqsupseteq\mathsf{rel}}x, \mathsf{W}x)\}$.[1]

Let $(\mathsf{W}^{\sqsupseteq\mathsf{rel}})$ and $(\mathsf{F}^{\sqsupseteq\mathsf{rel}})$ be *release* actions. Actions (R) are *read* actions.

Definition 1.1. We assume two equivalences: $(=_{\mathsf{gpu}}) \subseteq (\mathcal{T} \times \mathcal{T})$ partitions threads by *processor*, and $(=_{\mathsf{cta}}) \subseteq (=_{\mathsf{gpu}})$ refines the processor partitioning into *thread groups*.

[1] For PTX, one could additionally include $(\mathsf{R}x, \mathsf{R}^{\sqsupseteq\mathsf{acq}}x)$, but this is not sound for Arm or IMM.

We say $(\alpha_1 \mathsf{A}^{\mu_1}_{\sigma_1} x)$ *strongly-overlaps* $(\alpha_2 \mathsf{A}^{\mu_2}_{\sigma_2} x)$ when either

(1) $\alpha_1 = \alpha_2$, or
(2a) $\mu_1, \mu_2 \neq \mathsf{wk}$,
(2b) if $\sigma_1 = \mathsf{cta}$ or $\sigma_2 = \mathsf{cta}$ then $\alpha_1 =_{\mathsf{cta}} \alpha_2$,
(2c) if $\sigma_1 = \mathsf{gpu}$ or $\sigma_2 = \mathsf{gpu}$ then $\alpha_1 =_{\mathsf{gpu}} \alpha_2$.

We say $(\alpha_1 \mathsf{F}^{\mu_1}_{\sigma_1})$ *strongly-fences* $(\alpha_2 \mathsf{F}^{\mu_2}_{\sigma_2})$ when $\mu_1 = \mu_2 = \mathsf{sc}$ and either (1) or (2) apply, from the definition of strongly-overlaps.

We say a *strongly-matches* b when a is a release, b is an acquire, and either a strongly-overlaps b or a strongly-fences b.

Note that for a CPUs, all action have scope sys and mode rlx or greater. For this subset of actions, *strongly-overlaps* is the same as *overlaps* and *strongly-fences* applies to any pair of sc fences.

1.3 Pomsets with Predicate Transformers

Definition 1.2. A *predicate transformer* is a function $\tau : \Phi \longrightarrow \Phi$ such that

(x1) $\tau(\mathsf{ff}) \equiv \mathsf{ff}$,
(x2) $\tau(\psi_1 \wedge \psi_2) \equiv \tau(\psi_1) \wedge \tau(\psi_2)$,
(x3) $\tau(\psi_1 \vee \psi_2) \equiv \tau(\psi_1) \vee \tau(\psi_2)$,
(x4) if $\phi \vDash \psi$, then $\tau(\phi) \vDash \tau(\psi)$.

Definition 1.3. 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 $\tau^C(\psi) \vDash \tau^D(\psi)$.

We write $\tau(\psi)$ as an abbreviation of $\tau^E(\psi)$.

Definition 1.4. A *pomset with predicate transformers* is a tuple $(E, \lambda, \kappa, \tau, \checkmark, \trianglelefteq, \leq, \sqsubseteq, \mathsf{rmw})$ where

(M1) $E \subseteq \mathcal{E}$ is a set of *events*,
(M2) $\lambda : E \longrightarrow \mathcal{A}$ defines a *label* for each event,
(M3) $\kappa : E \longrightarrow \Phi$ defines a *precondition* for each event, such that
  (M3a) $\kappa(e)$ is satisfiable,
(M4) $\tau : 2^{\mathcal{E}} \longrightarrow \Phi \longrightarrow \Phi$ is a *family of predicate transformers* over E,
(M5) $\checkmark : \Phi$ is a *termination condition*, such that
  (M5a) $\checkmark \vDash \tau(\mathsf{tt})$,
(M6) $\trianglelefteq : (E \times E)$ is a partial order capturing *dependency*,
(M7) $\leq : (E \times E)$ is a partial order capturing *synchronization*,
(M8) $\sqsubseteq : (E \times E)$ is a partial order capturing *per-location order*, such that
  (M8a) if $\lambda(d)$ overlaps $\lambda(e)$ then $d \leq e$ implies $d \sqsubseteq e$,
(M9) $\mathsf{rmw} : E \longrightarrow E$ is a partial function capturing read-modify-write *atomicity*, such that
  (M9a) if $d \xrightarrow{\mathsf{rmw}} e$ then $\lambda(e)$ blocks $\lambda(d)$,
  (M9b) if $d \xrightarrow{\mathsf{rmw}} e$ then $d \leq e$ and $d \sqsubseteq e$,
  (M9c) if $\lambda(c)$ overlaps $\lambda(d)$ then
    (i) if $d \xrightarrow{\mathsf{rmw}} e$ then $c \trianglelefteq e$ implies $c \trianglelefteq d$, $c \leq e$ implies $c \leq d$, $c \sqsubseteq e$ implies $c \sqsubseteq d$,
    (ii) if $d \xrightarrow{\mathsf{rmw}} e$ then $d \trianglelefteq c$ implies $e \trianglelefteq c$, $d \leq c$ implies $e \leq c$, $d \sqsubseteq c$ implies $e \sqsubseteq c$.

A pomset is a *candidate* if there is an injective relation $\mathsf{rf} : E \times E$, capturing *reads-from*, such that

(C2a) if $d \xrightarrow{\mathsf{rf}} e$ then $\lambda(d)$ matches $\lambda(e)$,
(C6) if $d \xrightarrow{\mathsf{rf}} e$ then $d \trianglelefteq e$,
(C7a) if $d' \leq d \xrightarrow{\mathsf{rf}} e \leq e'$ and $\lambda(d')$ strongly-matches $\lambda(e')$ then $d' \leq e'$,
(C7b) if $\lambda(d)$ strongly-fences $\lambda(e)$ then either $d \leq e$ or $e \leq d$,
(C8a) if $d \xrightarrow{\mathsf{rf}} e$ then $d \sqsubseteq e$,
(C8b) if $d \xrightarrow{\mathsf{rf}} e$ and $\lambda(c)$ blocks $\lambda(e)$ then either $c \underset{\sim}{\sqsubset} d$ or $e \underset{\sim}{\sqsubset} c$,
  where $d' \underset{\sim}{\sqsubset} e'$ when $e' \sqsubseteq d'$ implies $d' = e'$ and $\lambda(d')$ strongly-overlaps $\lambda(e')$ implies $d' \sqsubseteq e'$.

A candidate pomset with rf is *complete* if

(c2b) if $\lambda(e)$ is a read then there is some $d \xrightarrow{\mathsf{rf}} e$,
(c3) $\kappa(e)$ is a tautology (for every $e \in E$),
(c5) $\checkmark$ is a tautology.

Note that for the IMM model, c8b is equivalent to:[2]

$$\text{if } d \xrightarrow{\mathsf{rf}} e \text{ and } \lambda(c) \text{ blocks } \lambda(e) \text{ then either } c \sqsubseteq d \text{ or } e \sqsubseteq c.$$

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

We drop quantifiers when clear from context, such as $(\forall e \in E)(\forall x \in \mathcal{X})$. We write $d < e$ when $d \leq e$ and $d \neq e$, and similarly for $\triangleleft$ and $\sqsubset$. We sometimes use projection functions—for example, if $\lambda(e) = \alpha \mathsf{W}^{\mu}_{\sigma} x v$ then $\lambda_{\mathsf{thrd}}(e) = \alpha$, $\lambda_{\mathsf{mode}}(e) = \mu$, $\lambda_{\mathsf{scope}}(e) = \sigma$, $\lambda_{\mathsf{loc}}(e) = x$, $\lambda_{\mathsf{val}}(e) = v$.

1.4 Semantics

See Figure 1.

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:

- $d \rightarrow e$ arises from control/data/address *dependency* (s3, definition of $\kappa'_2(d)$),
- $d \rightarrow e$ arises from *sync-delays* (s7a),
- $d \dashrightarrow e$ arises from *co-delays* (s8a),
- $d \longrightarrow e$ arises from *matching* (c6), (c7a) and (c8a),
- $d \longrightarrow e$ arises from *strong fencing* (c7b),
- $d \dashrightarrow e$ arises from *blocking* (c8b).

1.5 Address Calculation

Definition 1.5. If $P \in \mathit{WRITE}(L, M, \mu, \sigma)_\alpha$ then $(\exists \ell \in \mathcal{V})$ $(\exists v \in \mathcal{V})$

(w1) if $d, e \in E$ then $d = e$,
(w2) $\lambda(e) = \alpha \mathsf{W}^{\mu}_{\sigma}[\ell]v$,
(w3) $\kappa(e) \equiv L{=}\ell \wedge M{=}v$,
(w4a) if $E \neq \emptyset$ then $\tau^D(\psi) \equiv (L{=}\ell) \Rightarrow \psi[M/[\ell]]$,
(w4b) if $E = \emptyset$ then $\tau^D(\psi) \equiv (\forall k)(L{=}k) \Rightarrow \psi[M/[k]]$
(w5a) if $E \neq \emptyset$ then $\checkmark \equiv L{=}\ell \wedge M{=}v$,
(w5b) if $E = \emptyset$ then $\checkmark \equiv \mathsf{ff}$.

If $P \in \mathit{READ}(r, L, \mu, \sigma)_\alpha$ then $(\exists \ell \in \mathcal{V})$ $(\exists v \in \mathcal{V})$

(R1) if $d, e \in E$ then $d = e$,
(R2) $\lambda(e) = \alpha \mathsf{R}^{\mu}_{\sigma}[\ell]v$
(R3) $\kappa(e) \equiv L{=}\ell$,
(R4a) $(\forall e \in E \cap D)$ $\tau^D(\psi) \equiv (L{=}\ell \Rightarrow v{=}s_e) \Rightarrow \psi[s_e/r]$,
(R4b) $(\forall e \in E \setminus D)$ $\tau^D(\psi) \equiv ((L{=}\ell \Rightarrow v{=}s_e) \vee (L{=}\ell \Rightarrow [\ell]{=}s_e)) \Rightarrow \psi[s_e/r]$,
(R4c) $(\forall s)$ if $E = \emptyset$ then $\tau^D(\psi) \equiv \psi[s/r]$,
(R5a) if $E \neq \emptyset$ or $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{tt}$.
(R5b) if $E = \emptyset$ and $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \mathsf{ff}$.

1.6 If-closure

Definition 1.6. Let $E \subseteq \mathcal{E}$ and $\theta : E \rightarrow \Phi$ and $\Omega \in \Phi$. We say that θ *partitions* Ω if

[2]If all accesses are morally strong with each other, weak fulfillment degenerates to

$$\forall \lambda(c) = (\mathsf{W}x) \text{ either } c \sqsubseteq d \text{ or } e \sqsubseteq c$$

If no accesses are morally strong with each other, weak fulfillment degenerates to

$$\nexists \lambda(c) = (\mathsf{W}x) \text{ both } d \sqsubset c \text{ and } c \sqsubset e$$

Note that the difference between strong and weak fulfillment is limited to $\sqsubseteq$. We sometimes write $\sqsubseteq_{\mathsf{s}}$ for strong fulfillment and $\sqsubseteq_{\mathsf{w}}$ for weak fulfillment.

If $P \in \mathit{SKIP}$ then $E = \emptyset$ and $\tau^D(\psi) \equiv \psi$.

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

- (P1) $E = (E_1 \uplus E_2)$,
- (P2) $\lambda = (\lambda_1 \cup \lambda_2)$,
- (P3a) if $e \in E_1$ then $\kappa(e) \equiv \kappa_1(e)$,
- (P3b) if $e \in E_2$ then $\kappa(e) \equiv \kappa_2(e)$,
- (P4) $\tau^D(\psi) \equiv \tau_1^D(\psi)$,
- (P5) $\checkmark \equiv \checkmark_1 \wedge \checkmark_2$,
- (P6) $\trianglelefteq \supseteq (\trianglelefteq_1 \cup \trianglelefteq_2)$,
- (P7) $\leq \supseteq (\leq_1 \cup \leq_2)$,
- (P8) $\sqsubseteq \supseteq (\sqsubseteq_1 \cup \sqsubseteq_2)$,
- (P9) $\mathsf{rmw} = (\mathsf{rmw}_1 \cup \mathsf{rmw}_2)$.

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

- (S1) $E = (E_1 \cup E_2)$,
- (S2) (S6) (S7) (S8) (S9) as for *PAR*,
- (S3a) if $e \in E_1 \setminus E_2$ then $\kappa(e) \equiv \kappa_1(e)$,
- (S3b) if $e \in E_2 \setminus E_1$ then $\kappa(e) \equiv \kappa'_2(e) \wedge \checkmark_1(e)$,
- (S3c) if $e \in E_1 \cap E_2$ then $\kappa(e) \equiv (\kappa_1(e) \vee \kappa'_2(e)) \wedge \checkmark_1(e)$,
- (S4) $\tau^D(\psi) \equiv \tau_1^D(\tau_2^D(\psi))$,
- (S5) $\checkmark \equiv \checkmark_1 \wedge \tau_1(\checkmark_2)$,
- (S7a) if $\lambda_1(d)$ sync-delays $\lambda_2(e)$ and $\kappa_1(d) \wedge \kappa_2(e)$ is satisfiable then $d \leq e$,
- (S8a) if $\lambda_1(d)$ co-delays $\lambda_2(e)$ and $\kappa_1(d) \wedge \kappa_2(e)$ is satisfiable then $d \sqsubseteq e$,

where $\kappa'_2(e) = \tau_1(\kappa_2(e))$ if $\lambda(e)$ is a read—otherwise $\kappa'_2(e) = \tau_1^{\downarrow e}(\kappa_2(e))$, where $\downarrow e = \{c \mid c \triangleleft e\}$;
where $\checkmark_1(e) = \checkmark_1$ if $\lambda(e)$ is a release—otherwise $\checkmark_1(e) = \mathrm{tt}$.

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

- (I1) $E = (E_1 \cup E_2)$,
- (I2) (I6) (I7) (I8) (I9) as for *PAR*,
- (I3a) if $e \in E_1 \setminus E_2$ then $\kappa(e) \equiv \phi \wedge \kappa_1(e)$,
- (I3b) if $e \in E_2 \setminus E_1$ then $\kappa(e) \equiv \neg\phi \wedge \kappa_2(e)$,
- (I3c) if $e \in E_1 \cap E_2$ then $\kappa(e) \equiv (\phi \wedge \kappa_1(e)) \vee (\neg\phi \wedge \kappa_2(e))$,
- (I4) $\tau^D(\psi) \equiv (\phi \wedge \tau_1^D(\psi)) \vee (\neg\phi \wedge \tau_2^D(\psi))$,
- (I5) $\checkmark \equiv (\phi \wedge \checkmark_1) \vee (\neg\phi \wedge \checkmark_2)$.

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

If $P \in \mathit{READ}(r, x, \mu, \sigma)_\alpha$ then $(\exists v \in \mathcal{V})$

- (R1) if $d, e \in E$ then $d = e$,
- (R2) $\lambda(e) = \alpha\mathsf{R}^{\mu}_{\sigma}xv$,
- (R3) $\kappa(e) \equiv \mathrm{tt}$,
- (R4a) if $E = \{e\}$ and $(E \cap D) \neq \emptyset$ then $\tau^D(\psi) \equiv v{=}s_e \Rightarrow \psi[s_e/r]$,
- (R4b) if $E = \{e\}$ and $(E \cap D) = \emptyset$ then $\tau^D(\psi) \equiv (v{=}s_e \vee x{=}s_e) \Rightarrow \psi[s_e/r]$,
- (R4c) if $E = \emptyset$ then $(\forall s)\tau^D(\psi) \equiv \psi[s/r]$,
- (R5a) if $E \neq \emptyset$ or $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathrm{tt}$.
- (R5b) if $E = \emptyset$ and $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \mathrm{ff}$.

If $P \in \mathit{WRITE}(x, M, \mu, \sigma)_\alpha$ then $(\exists v \in \mathcal{V})$

- (W1) if $d, e \in E$ then $d = e$,
- (W2) $\lambda(e) = \alpha\mathsf{W}^{\mu}_{\sigma}xv$,
- (W3) $\kappa(e) \equiv M{=}v$,
- (W4) $\tau^D(\psi) \equiv \psi[M/x]$,
- (W5a) if $E \neq \emptyset$ then $\checkmark \equiv M{=}v$.
- (W5b) if $E = \emptyset$ then $\checkmark \equiv \mathrm{ff}$,

If $P \in \mathit{FENCE}(\mu, \sigma)_\alpha$ then

- (F1) if $d, e \in E$ then $d = e$,
- (F2) $\lambda(e) = \alpha\mathsf{F}^{\mu}_{\sigma}$,
- (F3) $\kappa(e) \equiv \mathrm{tt}$,
- (F4) $\tau^D(\psi) \equiv \psi$,
- (F5a) if $E \neq \emptyset$ then $\checkmark \equiv \mathrm{tt}$,
- (F5b) if $E = \emptyset$ then $\checkmark \equiv \mathrm{ff}$.

$$\begin{array}{rlrl}
[\![r := M]\!]_\alpha &= \mathit{LET}(r, M) & [\![\mathtt{skip}]\!]_\alpha &= \mathit{SKIP} \\
[\![r := x^{\mu}]\!]_\alpha &= \mathit{READ}(r, x, \mu, \sigma)_\alpha & [\![S_1 \;]\!]_\gamma\; S_2]\!]_\alpha &= \mathit{PAR}([\![S_1]\!]_\alpha, [\![S_2]\!]_\gamma) \\
[\![x^{\mu} := M]\!]_\alpha &= \mathit{WRITE}(x, M, \mu, \sigma)_\alpha & [\![S_1 ; S_2]\!]_\alpha &= \mathit{SEQ}([\![S_1]\!]_\alpha, [\![S_2]\!]_\alpha) \\
[\![\mathsf{F}^{\mu}_{\sigma}]\!]_\alpha &= \mathit{FENCE}(\mu, \sigma)_\alpha & [\![\mathtt{if}(M)\{S_1\}\mathtt{else}\{S_2\}]\!]_\alpha &= \mathit{IF}(M{\neq}0, [\![S_1]\!]_\alpha, [\![S_2]\!]_\alpha)
\end{array}$$

Fig. 1. Semantics of programs

- if $\theta_e \wedge \theta_d$ is satisfiable then $e = d$,
- $\Omega \equiv \bigvee_{e \in E} \theta_e$.

Definition 1.7. If $P \in \mathit{WRITE}(x,\ M,\ \mu,\ \sigma)_\alpha$ then $(\exists v : E \to \mathcal{V})\ (\exists \theta : E \to \Phi)\ (\exists \Omega \in \Phi)$

(w1) θ partitions Ω,
(w2) $\lambda(e) = \alpha\mathsf{W}^{\mu}_{\sigma} x v_e$,
(w3) $\kappa(e) \equiv \theta_e \wedge M{=}v_e$,
(w4) $\tau^D(\psi) \equiv \psi[M/x]$,
(w5) $\checkmark \equiv \Omega \wedge \bigwedge_{e \in E} \theta_e \Rightarrow M{=}v_e$.

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

(R1) θ partitions Ω,
(R2) $\lambda(e) = \alpha\mathsf{R}^{\mu}_{\sigma} x v_e$
(R3) $\kappa(e) \equiv \theta_e$,
(R4) $\tau^D(\psi) \equiv \bigwedge_{e \in E \cap D} \theta_e \Rightarrow (v_e{=}s_e) \Rightarrow \psi[s_e/r]$
$\quad \wedge \bigwedge_{e \in E \setminus D} \theta_e \Rightarrow (v_e{=}s_e \vee x{=}s_e) \Rightarrow \psi[s_e/r]$
$\quad \wedge \neg\Omega \Rightarrow (\forall s)\psi[s/r]$,
(R5a) if $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{tt}$.
(R5b) if $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \Omega$

1.7 Address Calculation and If-closure

Definition 1.8. If $P \in \mathit{WRITE}(L,\ M,\ \mu,\ \sigma)_\alpha$ then $(\exists \ell : E{\to}\mathcal{V})(\exists v : E{\to}\mathcal{V})(\exists \theta : E{\to}\Phi)(\exists \Omega \in \Phi)$

(w1) θ partitions Ω,
(w2) $\lambda(e) = \alpha\mathsf{W}^{\mu}_{\sigma}[\ell]v_e$,
(w3) $\kappa(e) \equiv \theta_e \wedge L{=}\ell_e \wedge M{=}v_e$,
(w4) $\tau^D(\psi) \equiv \bigwedge_{e \in E} \theta_e \Rightarrow (L{=}\ell_e) \Rightarrow \psi[M/[\ell_e]]$
$\quad \wedge \neg\Omega \Rightarrow (\forall k)(L{=}k) \Rightarrow \psi[M/[k]]$,
(w5) $\checkmark \equiv \Omega \wedge \bigwedge_{e \in E} \theta_e \Rightarrow (L{=}\ell_e \wedge M{=}v_e)$.

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

(R1) θ partitions Ω,
(R2) $\lambda(e) = \alpha\mathsf{R}^{\mu}_{\sigma}[\ell]v_e$
(R3) $\kappa(e) \equiv \theta_e \wedge L{=}\ell_e$,
(R4) $\tau^D(\psi) \equiv \bigwedge_{e \in E \cap D} \theta_e \Rightarrow (L{=}\ell_e \Rightarrow v_e{=}s_e) \Rightarrow \psi[s_e/r]$
$\quad \wedge \bigwedge_{e \in E \setminus D} \theta_e \Rightarrow ((L{=}\ell_e \Rightarrow v_e{=}s_e) \vee (L{=}\ell_e \Rightarrow [\ell]{=}s_e)) \Rightarrow \psi[s_e/r]$
$\quad \wedge \neg\Omega \Rightarrow (\forall s)\psi[s/r]$,
(R5a) if $\mu \sqsubseteq \mathsf{rlx}$ then $\checkmark \equiv \mathsf{tt}$.
(R5b) if $\mu \sqsupseteq \mathsf{acq}$ then $\checkmark \equiv \Omega$

Definition 1.9. Let READ' be defined as for READ, adding the constraint:

(R4d) if $(E \cap D) = \emptyset$ then $\tau^D(\psi) \equiv \psi$.

If $P \in \mathit{FADD}(r, L, M, \mu, \nu)$ then $(\exists P_1 \in \mathit{SEQ}(\mathit{READ}'(r,\ L,\ \mu),\ \mathit{WRITE}(L,\ r{+}M,\ \nu)))$
(U1) if $\lambda_1(e)$ is a write then there is a read $\lambda_1(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.
If $P \in \mathit{EXCHG}(r, L, M, \mu, \nu)$ then $(\exists P_1 \in \mathit{SEQ}(\mathit{READ}'(r,\ L,\ \mu),\ \mathit{WRITE}(L,\ M,\ \nu)))$
(U1) if $\lambda_1(e)$ is a write then there is a read $\lambda_1(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.
If $P \in \mathit{CAS}(r, L, M, N, \mu, \nu)$ then $(\exists P_1 \in \mathit{SEQ}(\mathit{READ}'(r,\ L,\ \mu),\ \mathit{IF}(r{=}M,\ \mathit{WRITE}(L,\ N,\ \nu),\ \mathit{SKIP})))$
(U1) if $\lambda_1(e)$ is a write then there is a read $\lambda_1(d)$ such that $\kappa(e) \vDash \kappa(d)$ and $d \xrightarrow{\mathsf{rmw}} e$.

2 PROPERTIES

Lemma 2.1. *(a)* $\mathcal{P} = (\mathcal{P} \parallel \mathit{SKIP}) = (\mathcal{P} ; \mathit{SKIP}) = (\mathit{SKIP} ; \mathcal{P})$.
(b) $(\mathcal{P}_1 \parallel \mathcal{P}_2) \parallel \mathcal{P}_3 = \mathcal{P}_1 \parallel (\mathcal{P}_2 \parallel \mathcal{P}_3)$.
(c) $(\mathcal{P}_1 ; \mathcal{P}_2) ; \mathcal{P}_3 = \mathcal{P}_1 ; (\mathcal{P}_2 ; \mathcal{P}_3)$.
(d) $\mathtt{if}(\phi)\{\mathcal{P}_1\}\mathtt{else}\{\mathcal{P}_2\} = \mathtt{if}(\phi)\{\mathcal{P}_1\}; \mathtt{if}(\neg\phi)\{\mathcal{P}_2\} = \mathtt{if}(\neg\phi)\{\mathcal{P}_2\}; \mathtt{if}(\phi)\{\mathcal{P}_1\}$.

(e) $\texttt{if}(\phi)\{\mathcal{P}_1\}\,\texttt{else}\,\{\mathcal{P}_2\} = \mathcal{P}_1$ *if* ϕ *is a tautology.*
(f) $\texttt{if}(\phi)\{\texttt{if}(\psi)\{\mathcal{P}\}\} = \texttt{if}(\phi \land \psi)\{\mathcal{P}\}$.
(g) $\texttt{if}(\phi)\{\mathcal{P}_1\,;\,\mathcal{P}_3\}\,\texttt{else}\,\{\mathcal{P}_2\,;\,\mathcal{P}_3\} \supseteq \texttt{if}(\phi)\{\mathcal{P}_1\}\,\texttt{else}\,\{\mathcal{P}_2\}\,;\,\mathcal{P}_3$.
(h) $\texttt{if}(\phi)\{\mathcal{P}_1\,;\,\mathcal{P}_2\}\,\texttt{else}\,\{\mathcal{P}_1\,;\,\mathcal{P}_3\} \supseteq \mathcal{P}_1\,;\,\texttt{if}(\phi)\{\mathcal{P}_2\}\,\texttt{else}\,\{\mathcal{P}_3\}$.
(i) $\texttt{if}(\phi)\{\mathcal{P}\}\,\texttt{else}\,\{\mathcal{P}\} \supseteq \mathcal{P}$.

Proof. Straightforward calculation. (a) requires M5a for the termination condition in ($\mathcal{P}$; *SKIP*).
(c) requires both conjunction closure (x2, for the termination condition) and disjunction closure (x3, for the predicate transformers themselves).
(d) requires s7a and s8a not impose order when $\kappa_1(d) \land \kappa_2(e)$ is unsatisfiable, which in turn requires that κ calculates *weakest* preconditions, rather than simple preconditions (see [Jeffrey and Riely 2021]).
(e) requires M3a.
In §1.6, we refine the semantics to validate the reverse inclusions for (g), (h), and (i). □

Definition 2.2. P_2 is an *augment* of P_1 if

(1) $E_2 = E_1$,
(2) $\lambda_2(e) = \lambda_1(e)$,
(3) $\kappa_2(e) \equiv \kappa_1(e)$,
(4) $\tau^D_2(\psi) \equiv \tau^D_1(\psi)$,
(5) $\checkmark_2 \equiv \checkmark_1$,
(6) $\mathsf{rf}_2 \supseteq \mathsf{rf}_1$,
(7) $\le_2 \supseteq \le_1$.

Lemma 2.3. *If* $P_1 \in [\![S]\!]$ *and* P_2 *augments* P_1 *then* $P_2 \in [\![S]\!]$.

Proof. Induction on the definition of $[\![\cdot]\!]$. □

3 DEAD STORE ELIMINATION, STORE FORWARDING, AND MONOTONICITY

We validate "monotonicity" by updating the rules for read, write and fence to include $(\exists \nu \sqsupseteq \mu)$:

(R2) $\lambda(e) = \alpha\mathsf{R}^{\nu}_{\sigma}xv$, (W2) $\lambda(e) = \alpha\mathsf{W}^{\nu}_{\sigma}xv$, (F2) $\lambda(e) = \alpha\mathsf{F}^{\nu}_{\sigma}$.

One could do the same for scopes.
The semantics already validates:

- $[\![x := M\,;\,x := M]\!] \supseteq [\![x := M]\!]$
- $[\![s := x\,;\,r := x]\!] \supseteq [\![s := x\,;\,r := s]\!]$
- $[\![r := x]\!] \supseteq [\![\texttt{skip}]\!]$

It does not validate:

- $[\![x := M\,;\,x := N]\!] \supseteq [\![x := N]\!]$
- $[\![x := M\,;\,r := x]\!] \supseteq [\![x := M\,;\,r := M]\!]$

As noted in §??, the semantics of Figure 1 validates elimination of irrelevant relaxed reads. In §??, we discussed redundant read elimination. Figure 1 also validates elimination of writes of the same value. However, Figure 1 does not validate general write elimination, where, for example, ($x := 1$; $x := 2$) is refined to $x := 2$. Nor does it validate store forwarding, where, for example, ($x := 1$; $r := x$) is refined to ($x := 1$; $r := 1$).

Elimination can be justified in pomset by *merging* actions with different labels. A list of safe merges can be found in [Chakraborty and Vafeiadis 2017, §E] and [Kang 2019, §7.1]. For examples of unsafe merges and reorderings, see [Chakraborty and Vafeiadis 2017, §D]. See also [Chakraborty and Vafeiadis 2019, §6.2]

Read-read and fence-fence merges can be handled by "monotonicity": allowing actions to put down stronger modes in the model. Then they can merge on the nose.

Sad: read elimination can't be done the nice way using $\tau^D(\psi) \equiv x{=}r \Rightarrow \psi$ for R4c because there may be a release-acquire pair between the read and the matching write.

Let $\text{merge} : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ be a partial function defined as follows.

$$\text{merge}(a,\ b) = \begin{cases} a & \text{if } a = b \text{ or } a = (\alpha \mathsf{W}^{\mu}_{\sigma} x v) \text{ and } b = (\alpha \mathsf{R}^{\nu}_{\sigma} x v) \\ b & \text{if } a = b \text{ or } a = (\alpha \mathsf{W}^{\mu}_{\sigma} x v) \text{ and } b = (\alpha \mathsf{W}^{\nu}_{\sigma} x w) \\ \text{undefined} & \text{otherwise} \end{cases}$$

(If we have "monotonicity" then we can require $\mu = \nu$.)

If $a_0 = \text{merge}(a_1,\ a_2)$, then a_1 and a_2 can coalesce, resulting in a_0. This allows optimizations such as $(x := 1;\ x := 2)$ to $(x := 2)$ and $(x := 1;\ r := x)$ to $(x := 1;\ r := 1)$. For associativity of sequential composition, it is important that merge always take an upper bound on the modes of the two actions. For example, it would invalidate associativity to allow $(\mathsf{W} x v) = \text{merge}(\mathsf{W} x v,\ \mathsf{R}^{\mathsf{acq}} x v)$, although this is considered safe.

Then we can replace s2-s3 in Figure 1 by:

(s2a) if $e \in E_1 \setminus E_2$ then $\lambda(e) = \lambda_1(e)$,
(s2b) if $e \in E_2 \setminus E_1$ then $\lambda(e) = \lambda_2(e)$,
(s2c) if $e \in E_1 \cap E_2$ then $\lambda(e) = \text{merge}(\lambda_1(e),\ \lambda_2(e))$,
(s3a) if $e \in E_1 \setminus E_2$ then $\kappa(e) \equiv \kappa_1(e)$,
(s3b) if $e \in E_2 \setminus E_1$ then $\kappa(e) \equiv \kappa'_2(e)$,
(s3c) if $e \in E_1 \cap E_2$ then either
- $\lambda_1(e) = \lambda(e) = \lambda_2(e)$ and $\kappa(e) \equiv \kappa_1(e) \vee \kappa'_2(e)$,
- $\lambda_1(e) = \lambda(e) \neq \lambda_2(e)$ and $\kappa'_2(e) \equiv \kappa(e) \equiv \kappa_1(e)$ (write-read),
- $\lambda_1(e) \neq \lambda(e) = \lambda_2(e)$ and $\kappa_1(e) \equiv \kappa(e) \equiv \kappa'_2(e)$ (write-write).

Full merge: `if`(M)`{`$x := 1$`}`; $x := 2$ can become $x := 2$.
Partial merge: $x := 1$; `if`(M)`{`$x := 2$`}` can become `if`(M)`{`$x := 2$`}` `else` `{`$x := 1$`}`.
To get associativity, you need the ability to merge with multiple events.

$$x := 1;\ \mathtt{if}(M)\{x := 2\} \quad \boxed{\neg M \mid \mathsf{W}x1}\ \boxed{M \mid \mathsf{W}x2} \qquad\qquad \mathtt{if}(!M)\{x := 2\} \quad \boxed{\neg M \mid \mathsf{W}x2}$$

This is asymmetric. We don't expect to merge all three events in the following:

$$\mathtt{if}(!M)\{x := 2\} \quad \boxed{\neg M \mid \mathsf{W}x2} \qquad\qquad x := 1;\ \mathtt{if}(M)\{x := 2\} \quad \boxed{\neg M \mid \mathsf{W}x1}\ \boxed{M \mid \mathsf{W}x2}$$

We could have a lot merging:

$$\mathtt{if}(N)\{x := 1;\ \mathtt{if}(M)\{x := 3\}\};\ \mathtt{if}(\neg N)\{x := 2;\ \mathtt{if}(M)\{x := 3\}\} \quad \boxed{\neg M \wedge N \mid \mathsf{W}x1}\ \boxed{M \wedge N \mid \mathsf{W}x3}\ \boxed{\neg M \wedge \neg N \mid \mathsf{W}x2}\ \boxed{M \wedge \neg N \mid \mathsf{W}x3} \qquad \mathtt{if}(!M)\{x := 3\} \quad \boxed{\neg M \mid \mathsf{W}x3}$$

Full merge: $x := 1$; `if`(M)`{`$r := x$`}` can become $x := 1$; `if`(M)`{`$r := 1$`}`.
Partial merge: `if`(M)`{`$x := 1$`}`; $r := x$ can become `if`(M)`{`$x := 1;\ r := 1$`}` `else` `{`$r := x$`}`.

I don't think we need multi-merge for write-read. Reads only affect the world via the predicate transformer. Any conditional surrounding a read is baked into the predicate transformer, and so does not to persist in the preconditions of the actions themselves after the merge. Consider $r := 1$; $x := 2$; `if`(M)`{`$r := x$`}`. This can safely transform to $r := 1$; $x := 2$; `if`(M)`{`$r := 2$`}`.

In the example below, the reads should *not* merge. Although the second read can merge with the write.

$$\mathtt{if}(!M)\{x := 1\};\ \mathtt{if}(M)\{r := x\} \quad \boxed{\neg M \mid \mathsf{W}x1}\ \boxed{M \mid \mathsf{R}x1} \qquad\qquad \mathtt{if}(!M)\{s := x\} \quad \boxed{\neg M \mid \mathsf{R}x1}$$

Another example:

$$\begin{array}{cc} x := 1;\ \mathtt{if}(M)\{r := x\} & \mathtt{if}(!M)\{s := x\} \\ \boxed{\mathsf{W}x1} & \boxed{\neg M \mid \mathsf{R}x1} \end{array}$$

Another example:

$$\begin{array}{cc} x := 1 & \mathtt{if}(M)\{r := x\};\ \mathtt{if}(!M)\{s := x\} \\ \boxed{\mathsf{W}x1} & \boxed{\mathsf{R}x1} \end{array}$$

Idea for multi-merge. Use $E_1' \subseteq E_1$, with a surjective function $\pi : E_1 \rightarrow E_1'$ that shows how writes merge.

- Require that $(\forall d \in E_1')\ \pi(d) = d$.
- Require that if $c \in (E_1 \setminus E_1')$ then $\pi(c) \in E_2$—and therefore $\pi(c) \in (E_1' \cap E_2)$.
- Take $E = E_1' \cup E_2$.

Require that the writes that coalesce have disjoint preconditions.

- if $\pi(c) = \pi(c')$ then $\kappa_1(c) \wedge \kappa_1(c')$ is unsatisfiable

Then each of them has to merge into the same write $e \in E_2$ using the merge function and combining the predicates as specified above.

(s2a) if $e \in E_1' \setminus E_2$ then $\lambda(e) = \lambda_1(e)$,
(s2b) if $e \in E_2 \setminus E_1'$ then $\lambda(e) = \lambda_2(e)$,
(s2c) if $e \in (E_1' \cap E_2)$ and $c \in E_1$ and $\pi(c) = e$ then $\lambda(e) = \mathsf{merge}(\lambda_1(c),\ \lambda_2(e))$,
(s3a) if $e \in E_1' \setminus E_2$ then $\kappa(e) \equiv \kappa_1(e)$,
(s3b) if $e \in E_2 \setminus E_1'$ then $\kappa(e) \equiv \kappa_2'(e)$,
(s3c) if $e \in (E_1' \cap E_2)$ then

- $\kappa(e) \equiv \kappa_2'(e) \vee \bigvee_{c \in C} \kappa_1(c)$, where $C = \{c \in E_1 \mid \pi(c) = e \text{ and } \lambda_1(c) = \lambda_2(e)\}$,
- if $\pi(c) = e$ and $\lambda_1(c) = \lambda(e) \neq \lambda_2(e)$ then $\kappa_2'(c) \equiv \kappa(e)$ (write-read),
- if $\pi(c) = e$ and $\lambda_1(c) \neq \lambda(e) = \lambda_2(e)$ then $\kappa_1(c) \equiv \kappa(e)$ (write-write).

4 DRF

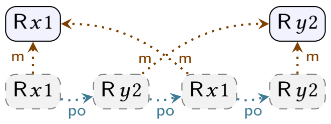
Restrict the syntax to top-level parallel composition.

Definition 4.1. A *pomset with program order* is a tuple $(P, \mathsf{m}, \mathsf{po})$, where P is a pomset with predicate transformers $(E, \lambda, \kappa, \tau, \checkmark, \trianglelefteq, \leq, \sqsubseteq, \mathsf{rmw})$ and

(M10) $\mathsf{m} : (F \times G)$ is a relation capturing *merging*, where
  (M10a) $F \subseteq E$ represents *phantom unmerged* events, let $\overline{F}$ be $E \setminus F$,
  (M10b) $G \subseteq \overline{F}$ represents *merged* events, let $\overline{G}$ be $E \setminus G$,
  (M10c) $\trianglelefteq, \leq, \sqsubseteq, \mathsf{rmw} : (\overline{F} \times \overline{F})$,
(M11) $\mathsf{po} : (\overline{G} \times \overline{G})$ is a partial order capturing *program order*, such that
  (M11a) po is total on events (in $\overline{G}$) of the same thread:
  - if $\lambda_{\mathsf{thrd}}(d) = \lambda_{\mathsf{thrd}}(e)$ then either $d \xrightarrow{\mathsf{po}} e$ or $e \xrightarrow{\mathsf{po}} d$,
  - if $d \xrightarrow{\mathsf{po}} e$ then $\lambda_{\mathsf{thrd}}(d) = \lambda_{\mathsf{thrd}}(e)$.

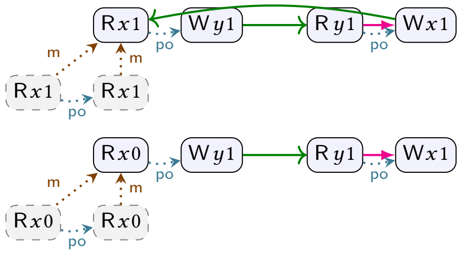
po can have cycles when interpreted on merged events. For example:

$r_1 := x\,;\, r_2 := y\,;\, s_1 := x\,;\, s_2 := y$



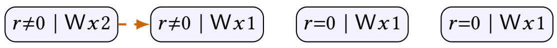
For TC2, we have:

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



Idea: merging reads can only make a difference is if there is a race.

$\texttt{if}(r)\{x := 2\,;\, x := 1\}\,;\, \texttt{if}(!r)\{x := 2\,;\, x := 1\}$



4.1 Local Data Race Freedom and Sequential Consistency

We adapt Dolan et al.'s [2018] notion of *Local Data Race Freedom (LDRF)* to our setting.

The result requires that locations are properly initialized. We assume a sufficient condition: that programs have the form "$x_1 := v_1\,;\, \cdots x_n := v_n\,;\, S$" where every location mentioned in S is some x_i.

We make two further restrictions to simplify the exposition. To simplify the definition of *happens-before*, we ban fences and RMWs. To simplify the proof, we assume there are no local declarations of the form $(\texttt{var}\, x\,;\, S)$.

To state the theorem, we require several technical definitions. The reader unfamiliar with [Dolan et al. 2018] may prefer to skip to the examples in the proof sketch, referring back as needed.

Data Race. Data races are defined using *program* order (po), not *pomset* order ($\le$). In ??, for example, $(\mathsf{R}x0)$ has an x-race with $(\mathsf{W}x1)$, but not $(\mathsf{W}x0)$, which is po-before it.

It is obvious how to enhance the semantics of prefixing and most other operators to define po. When combining pomsets using the conditional, the obvious definition may result in cycles, since po-ordered reads may coalesce—see the discussion of ?? in §??. In this case we include a separate pomset for each way of breaking these cycles.

Because we ignore the features of §??, we can adopt the simplest definition of *synchronizes-with* (sw): Let $d \xrightarrow{\mathsf{sw}} e$ exactly when d fulfills e, d is a release, e is an acquire, and $\neg(d \xrightarrow{\mathsf{po}} e)$.

Let $\mathsf{hb} = (\mathsf{po} \cup \mathsf{sw})^+$ be the *happens-before* relation. In ??, for example, $(\mathsf{W}x1)$ happens-before $(\mathsf{R}x0)$, but this fails if either ra access is relaxed.

Let $L \subseteq \mathcal{X}$ be a set of locations. We say that d *has an L-race with* e (notation $d \overset{L}{\rightsquigarrow} e$) when at least one is relaxed, they *conflict* (Def. ??) at some location in L, and they are unordered by hb: neither $d \xrightarrow{\mathsf{hb}} e$ nor $e \xrightarrow{\mathsf{hb}} d$.

Generators. We say that P' *generates* P if either P augments P' or P implies P'. For example, the unordered pomset $(\mathsf{R}x1)$ $(\mathsf{W}y1)$ generates the ordered pomset $(\mathsf{R}x1) \rightarrow (r = 1 \mid \mathsf{W}y1)$.

We say that P is a *generation-minimal* in $\mathcal{P}$ if $P \in \mathcal{P}$ and there is no $P \neq P' \in \mathcal{P}$ that generates P.

Let $\text{gen}[\![S]\!] = \{P \in [\![S]\!] \mid P$ is *top-level* (Def. ??) and generation-minimal in $[\![S]\!]\}$.

Extensions. We say that P' *S-extends* P if $P \neq P' \in \text{gen}[\![S]\!]$ and P is a downset of P'.

Similarity. We say that P' *is e-similar to* P if they differ at most in (1) pomset order adjacent to e and (2) the value associated with event e, if it is a read. Formally: $E' = E$, $\kappa' = \kappa$, $\leq'|_{E\setminus\{e\}} = \leq|_{E\setminus\{e\}}$, if e is not a read then $\lambda' = \lambda$, and if e is a read then $\lambda'|_{E\setminus\{e\}} = \lambda|_{E\setminus\{e\}}$ and $\lambda'(e) = \lambda(e)[v'/v]$, for some v', v.

Stability. We say that P is *L-stable in S* if (1) $P \in \text{gen}[\![S]\!]$, (2) P is po-convex (nothing missing in program order), (3) there is no S-extension of P with a *crossing* L-race: that is, there is no $d \in E$, no P' S-extending P, and no $e \in E' \setminus E$ such that $d \overset{L}{\leadsto} e$. The empty pomset is L-stable.

Sequentiality. Let $\lessdot_L = <_L \cup \text{po}$, where $<_L$ is the restriction of $<$ to events that access locations in L. We say that P' is *L-sequential after P* if P' is po-convex and $\lessdot_L$ is acyclic in $E' \setminus E$.

Theorem 4.2. *Let P be L-stable in S. Let P' be a S-extension of P that is L-sequential after P. Let P'' be a S-extension of P' that is* po*-convex, such that no subset of E'' satisfies these criteria. Then either (1) P'' is L-sequential after P or (2) there is some S-extension P''' of P' and some $e \in (E'' \setminus E')$ such that (a) P''' is e-similar to P'', (b) P''' is L-sequential after P, and (c) $d \overset{L}{\leadsto} e$, for some $d \in (E'' \setminus E)$.*

The theorem provides an inductive characterization of *Sequential Consistency for Local Data-Race Freedom (SC-LDRF)*: Any extension of a L-stable pomset is either L-sequential, or is e-similar to a L-sequential extension that includes a race involving e.

Proof Sketch. In order to develop a technique to find P''' from P'', we analyze pomset order in generation-minimal top-level pomsets. First, we note that $\leq_*$ (the transitive reduction $\leq$) can be decomposed into three disjoint relations. Let $\text{ppo} = (\leq_* \cap \text{po})$ denote *preserved* program order, as required by prefixing (Def. ??). The other two relations are cross-thread subsets of $(\leq_* \setminus \text{po})$, as required by fulfillment (Def. ??): rfe orders writes before reads, satisfying fulfillment requirement ??; xw orders read and write accesses before writes, satisfying requirement ??. (Within a thread, ?? and ?? follow from prefixing requirement ??, which is included in ppo.)

Using this decomposition, we can show the following.

Lemma 4.3. *Suppose $P'' \in \text{gen}[\![S]\!]$ has a read e that is maximal in* (ppo $\cup$ rfe) *and such that every* po*-following read is also $\leq$-following ($e \xrightarrow{\text{po}} d$ implies $e \leq d$, for every read d). Further, suppose there is an e-similar P''' that satisfies the requirements of fulfillment. Then $P''' \in \text{gen}[\![S]\!]$.*

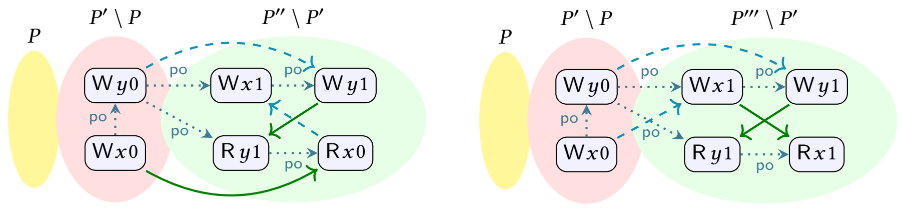
The proof of the lemma follows an inductive construction of $\text{gen}[\![S]\!]$, starting from a large set with little order, and pruning the set as order is added: We begin with all pomsets generated by the semantics without imposing the requirements of fulfillment (including only ppo). We then prune reads which cannot be fulfilled, starting with those that are minimally ordered. This proof is simplified by precluding local declarations.

We can prove a similar result for (po $\cup$ rfe)-maximal read and write accesses.

Turning to the proof of the theorem, if P'' is L-sequential after P, then the result follows from (1). Otherwise, there must be a $\lessdot_L$ cycle in P'' involving all of the actions in $(E'' \setminus E')$: If there were no such cycle, then P'' would be L-sequential; if there were elements outside the cycle, then there would be a subset of E'' that satisfies these criteria.

If there is a (po ∪ rfe)-maximal access, we select one of these as e. If e is a write, we reverse the outgoing order in xw; the ability to reverse this order witnesses the race. If e is a read, we switch its fulfilling write to a "newer" one, updating xw; the ability to switch witnesses the race. For example, for P'' on the left below, we choose the P''' on the right; e is the read of x, which races with $(\mathsf{W}x1)$.

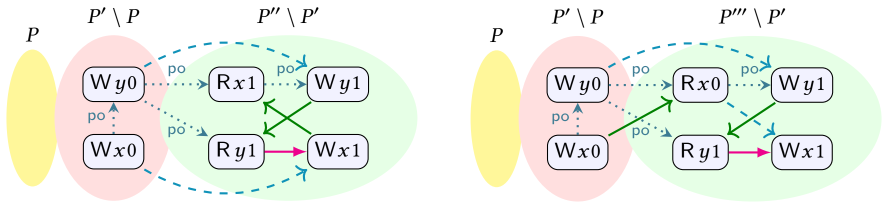
$$x := 0;\ y := 0;\ (x := 1;\ y := 1 \parallel \mathtt{if}(y)\{r := x\})$$



It is important that e be (po ∪ rfe)-maximal, not just (ppo ∪ rfe)-maximal. The latter criterion would allow us to choose e to be the read of y, but then there would be no e-similar pomset: if an execution reads 0 for y then there is no read of x, due to the conditional.

If there is no (po ∪ rfe)-maximal access, then all cross-thread order must be from rfe. In this case, we select a (ppo ∪ rfe)-maximal read, switching its fulfilling write to an "older" one. As an example, consider the following; once again, e is the read of x, which races with $(\mathsf{W}x1)$.

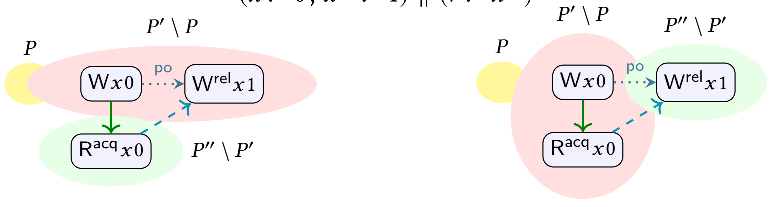
$$x := 0;\ y := 0;\ (r := x;\ y := 1 \parallel s := y;\ x := s)$$



This example requires $(\mathsf{W}x0)$. Proper initialization ensures the existence of such "older" writes. □

The premises of the theorem allow us to avoid the complications caused by "mixed races" in [Dongol et al. 2019]. In the left pomset below, P'' is not an extension of P', since P' is not a downset of P''. When considering this pomset, we must perform the decomposition on the right.

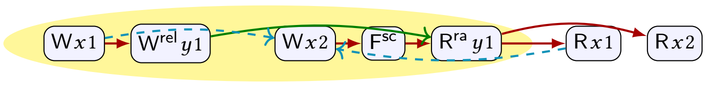
$$(x := 0;\ x^{\mathsf{ra}} := 1) \parallel (r := x^{\mathsf{ra}})$$



This affects the inductive order in which we move across pomsets, but does not affect the set of pomsets that are considered. This simplification is enabled by denotational reasoning.

In our language, past races are always resolved at a stable point, as in ??. As another example, consider the following, which is disallowed here, but allowed by Java [Dolan et al. 2018, Ex. 2]. We include an SC fence here to mimic the behavior of volatiles in the JMM.

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

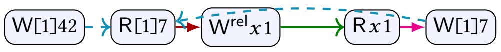


(PAST)

The highlighted events are L-stable. The order from $(\mathsf{R}x1)$ to $(\mathsf{W}x2)$ is required by fulfillment, causing the cycle. If the fence is removed, there would be no order from $(\mathsf{W}x2)$ to $(\mathsf{R}^{\mathsf{acq}}y1)$, the highlighted events would no longer be L-stable, and the execution would be allowed. This more relaxed notion of "past" is not expressible using Dolan et al.'s synchronization primitives.

The notion of "future" is also richer here. Consider [Dolan et al. 2018, Ex. 3]:

$$(r := 1;\ [r] := 42;\ s := [r];\ x^{\mathsf{ra}} := r) \parallel (r := x;\ [r] := 7)$$



(FUTURE)

There is no interesting stable point here. The execution is disallowed because of a read from the causal future. If we changed x^{ra} to x^{rlx}, then there would be no order from $(\mathsf{R}[1]7)$ to $(\mathsf{W}^{\mathsf{rlx}}x1)$, and the execution would be allowed. The distinction between "causal future" and "temporal future" is not expressible in Dolan et al.'s operational semantics.

Our definition of L-sequentiality does not quite correspond to SC executions, since actions may be elided by read/write elimination (§??). However, for any properly initialized L-sequential pomset that uses elimination, there is larger L-sequential pomset that does not use elimination. This can be shown inductively—in the inductive step, writes that are introduced can be ignored by existing reads, and reads that are introduced can be fulfilled, for some value, by some preceding write.

REFERENCES

Soham Chakraborty and Viktor Vafeiadis. 2017. Formalizing the concurrency semantics of an LLVM fragment. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Vijay Janapa Reddi, Aaron Smith, and Lingjia Tang (Eds.). ACM, 100–110. http://dl.acm.org/citation.cfm?id=3049844

Soham Chakraborty and Viktor Vafeiadis. 2019. Grounding thin-air reads with event structures. *PACMPL* 3, POPL (2019), 70:1–70:28. https://doi.org/10.1145/3290383

Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding Data Races in Space and Time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 242–255. https://doi.org/10.1145/3192366.3192421

Brijesh Dongol, Radha Jagadeesan, and James Riely. 2019. Modular transactions: bounding mixed races in space and time. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, Jeffrey K. Hollingsworth and Idit Keidar (Eds.). ACM, 82–93. https://doi.org/10.1145/3293883.3295708

William Ferreira, Matthew Hennessy, and Alan Jeffrey. 1996. A Theory of Weak Bisimulation for Core CML. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming, ICFP 1996, Philadelphia, Pennsylvania, USA, May 24-26, 1996*, Robert Harper and Richard L. Wexelblat (Eds.). ACM, 201–212. https://doi.org/10.1145/232627.232649

Alan Jeffrey and James Riely. 2021. Sequential Composition for Relaxed Memory: Pomsets with Predicate Transformers. https://github.com/chicago-relaxed-memory/seqcomp.

Jeehoon Kang. 2019. *Reconciling Low-Level Features of C with Compiler Optimizations*. Ph.D. Dissertation. Seoul National University, Seoul, South Korea. https://sf.snu.ac.kr/jeehoon.kang/thesis/