A Denotational Semantics for SPARC TSO

Ryan Kavanagh[1] Stephen Brookes[2]

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

The SPARC TSO weak memory model is defined axiomatically, with a non-compositional formulation that makes modular reasoning about programs difficult. Our denotational approach uses pomsets to provide a compositional semantics capturing exactly the behaviours permitted by SPARC TSO. Our approach facilitates the study of SPARC TSO and supports modular analysis of program behaviour.

Keywords: SPARC TSO, denotational semantics, pomsets, concurrency, weak memory models.

1 Introduction

A memory model specifies what values can be read by a thread from a given memory location. Traditional concurrency research has assumed *sequential consistency*, wherein memory actions operate atomically on a global state, and a read is guaranteed to observe the value most recently written to that location *globally*. Consequently, "the result of any execution is the same as if the operations of all the processors were executed in some sequential order" [10]. However, sequential consistency negatively impacts performance, and modern architectures often provide much weaker guarantees. These weaker guarantees mean that classical concurrency algorithms, which often assume sequential consistency, can behave in unexpected ways. Consider, for example, the Dekker algorithm on a system using the SPARC instruction set. The Dekker algorithm seeks to ensure that at most one process enters a critical section at a time. Executing the following instance of the Dekker algorithm on a sequentially consistent system from an initial state where memory locations w, x, y, z are all zero will ensure that we end in a state where not both z and w are set to one:

$$(x := 1; \mathbf{if}\ y = 0\ \mathbf{then}\ z := 1\ \mathbf{else\ skip}) \,\|\, (y := 1; \mathbf{if}\ x = 0\ \mathbf{then}\ w := 1\ \mathbf{else\ skip}).$$

[1] Email: rkavanagh@cs.cmu.edu Funded in part by a Natural Sciences and Engineering Research Council of Canada (NSERC) Postgraduate Scholarship.

[2] Email: brookes@cs.cmu.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

However, the SPARC ISA provides the weaker SPARC TSO (total store ordering) memory model. Under SPARC TSO, it is possible to start from the aforementioned initial state and end in a state where both z and w are set to one, thus violating mutual exclusion.

Weak memory models are often described in standards documents using natural language. This informality makes it difficult to reason about how programs will behave on systems that use these memory models. The SPARC Architecture Manual [12] gives an axiomatic description of TSO using partial orders of actions, and we present this axiomatic description in Section 2. Even with the formality of this axiomatic account, reasoning about the behaviour of programs is difficult, because the axiomatic approach is non-compositional and precludes modular reasoning. We address this problem by presenting a denotational semantics for SPARC TSO in Section 3. Our denotational semantics assigns to each program a collection of pomsets. Pomsets are generalisations of traces and were first used by Pratt [11] to give denotational semantics for concurrency, and later by Brookes [6], with some modifications, to study weak memory models. We illustrate our semantics by validating various *litmus tests* and expected program equivalences. To ensure our denotational semantics accurately captures the behaviour of SPARC TSO, we show in Section 4 that from every denotation of a program we can derive a collection of partial orders satisfying the axiomatic description of Section 2 and, moreover, that we can derive every such partial order from the denotation.

2 An Axiomatic Account

The SPARC manual [12] gives an axiomatic description of TSO in terms of partial orders of actions. Unfortunately, this description is incomplete because it fails to specify the fork and join behaviour of TSO. In this section, we fully axiomatise the SPARC manual's account of TSO. Before doing so, we give an informal description of TSO to help build intuition.

A system providing the TSO weak memory model can be thought of as a collection of processors, each with a write buffer. Whenever a processor performs a write, it places it in its write buffer. The buffer behaves as a queue, and writes migrate out of the buffers one at a time, and shared memory applies them according to a global total order. Whenever a processor tries to read a location, it first checks its buffer for a write to that location. If it finds one, it uses the value of the most recent such write; otherwise, it looks to shared memory for a value. Because of buffering, it is possible for writes and reads to be observed out of order relative to the program order.

2.1 Program Order Pomsets

To make the above intuition precise, we must formalise the notion of a *program order*, i.e., the ordering of read and write actions specified by a program. We do so by means of partially-ordered multisets.

A (strict) **partially-ordered multiset** or **pomset** $(P, <_P, \Phi)$ over a label set L consists of a strict poset $(P, <_P)$ of "action occurrences" and a function $\Phi : P \to L$ mapping each action occurrence to its label or "action". Denote by $\mathrm{Pom}(L)$ the set of pomsets over L.

We do not usually make the poset P explicit, because the structure of the pomset is invariant under relabellings of the elements of P. Consequently, we identify pomsets $(P, <_P, \Phi)$ and $(P', <'_P, \Phi')$ if there exists an order isomorphism $\phi : P \to P'$ such that $\Phi = \Phi' \circ \phi$. We usually denote the elements of the pomset using just their labels, but we sometimes need to specify their exact occurrence, in which case we write l_p, where $l = \Phi(p)$.

It is useful to draw a pomset P as a labelled directed acyclic graph, where multiple vertices can have the same label and we have an edge $a \to b$ if $a <_P b$. For clarity, we omit edges obtained by transitivity of $<_P$. For example, the following graph depicts the pomset where $P = \{0, 1, 2, 3\}$, the order is given by $0 < 1$, $1 < 2$, $0 < 2$, and $0 < 3$, and Φ is given by $\Phi(0) = a$, $\Phi(1) = b$, $\Phi(2) = a$, and $\Phi(3) = c$:

$$c \longleftarrow a \longrightarrow b \longrightarrow a\,.$$

We assume a countably infinite set of locations Loc, ranged over by metavariables $x, y, z, \ldots$, and a set of values V, ranged over by v. In our examples, we will take V to be the set of integers. We call $x := v$ a *global write action*, $x = v$ a *read action*, and δ a *skip action*. Let $\mathcal{A}_w$ and $\mathcal{A}_r$ be the sets of global write actions and read actions, respectively.

A **program order** is a pomset P over the set of action labels $\mathcal{A}_{PO} = \mathcal{A}_w \cup \mathcal{A}_r \cup \{\delta\}$ with the *(locally) finite height property*, that is, such that for all $b \in P$, the set $\{a \in P \mid a <_P b\}$ is finite.

Intuitively, the program order

$$x := 2 \longrightarrow y = 1 \qquad y := 1 \longrightarrow y = 1$$

describes the parallel execution of writing 2 to x before reading 1 from y, and writing 1 to y before reading 1 from y, with no other ordering constraints.

2.2 TSO Axioms

A global state is a finite partial function from locations Loc to values V. We let $\Sigma_{PO} = \text{Loc} \rightharpoonup_{fin} V$ be the set of global states, and use σ to range over Σ_{PO}.

A list L determines a strict total order $<_L$ on its elements, where $\lambda <_L \lambda'$ if and only if λ appears to the left of λ' in L. Given any set S and partial order $<_S$ on it, every element $s \in S$ determines a set $s{\downarrow_S} = \{s' \in S \mid s' <_S s\} \cup \{s\}$ called its *lower closure*. Write $s \#_S s'$ to denote that s and s' are not comparable under $<_S$ and $s \neq s'$, and write $s \perp_S s'$ to denote that they are comparable.

Definition 2.1 Let P be a program order and $<_T$ be a strict partial order on the elements of P. We say $<_T$ is **TSO-consistent** with P from (the initial state) σ if it satisfies the following six axioms:

(O) **Ordering:** $<_T$ is a total order on the write actions $\mathcal{A}_w$ of P.

(V) **Value:** for all reads $(x = v)_r$ in P, either
- (A) there exists a write $(x := v')_w$ maximal under $<_T$ amongst all writes to x in $(x = v)_r{\downarrow_T}$, all writes to x in $(x = v)_r{\downarrow_P}$ are in $(x := v')_w{\downarrow_T}$, and $v = v'$; or
- (B) there exists a write $(x := v')_w$ maximal under $<_P$ amongst all writes to x in $(x = v)_r{\downarrow_P}$, and both $(x = v)_r <_T (x := v')_w$ and $v = v'$; or
- (C) there are no writes to x in $(x = v)_r{\downarrow_T}$ or $(x = v)_r{\downarrow_P}$, and $\sigma(x) = v$.

(L) **LoadOp:** for all reads $r \in P$ and all actions $a \in P$, $r <_P a$ implies $r <_T a$.

(S) **StoreStore:** for all writes $w, w' \in P$, $w <_P w'$ implies $w <_T w'$.

(F) **Fork:** if $\alpha_1 <_P \alpha_2$, $\alpha_1 <_P \alpha_3$, and $\alpha_2 \#_P \alpha_3$, then $\alpha_1 <_T \alpha_2$ and $\alpha_1 <_T \alpha_3$.

(J) **Join:** if $\alpha_1 <_P \alpha_3$, $\alpha_2 <_P \alpha_3$, and $\alpha_1 \#_P \alpha_2$, then $\alpha_1 <_T \alpha_3$ and $\alpha_2 <_T \alpha_3$.

We simply say $<_T$ is TSO-consistent with P if there exists some initial state σ from which they are TSO-consistent.

Axioms (O), (Va), (Vb), (L), and (S) are directly adapted from the formal specification given in Appendix K.2 of [12]. We introduce axiom (Vc) to simplify our presentation of examples. By requiring that programs first write to any locations from which they read, it can be omitted, and apart from examples, we will assume throughout that our TSO-consistent orders do not require (Vc). Though the formal specification does not provide axioms (F) and (J), they are consistent with the behaviour intended by Appendix J.6 of [12]. Intuitively, axiom (Vb) requires that whenever a processor reads from a location, it must use the most recent write to that location in its write buffer (if it exists), and if there is no such write in its write buffer, but we have observed a global write to that location, then (Va) requires that the most recent such write be the one read. Our presentation differs slightly from the formal specification. In particular, we do not consider instruction fetches or atomic load-store operations, and we do not consider flush actions, because they can be implemented as a derived action in our semantics by forking and immediately joining. To be consistent with our pomset development, we also assume the order to be strict.

If $<_T$ is TSO-consistent for P, then there exists a (not necessarily unique) total order on P that is TSO-consistent with P and contains $<_T$. This is consistent with our physical intuition: given an actual execution, we can imagine taking a temporal linearisation of the operations. As a result, we can view all orders that are TSO-consistent with P as weakenings of total orders that are TSO-consistent with P. One should be careful not to conflate linearisations and TSO-consistent total orders. Consider, for example, the program order

$$x := 2 \longrightarrow x = 2 \qquad x := 3 \longrightarrow x = 3\,.$$

The linearisation $x := 2 < x := 3 < x = 3 < x = 2$ is not TSO-consistent with the program order because it violates (Va); the order $x = 2 < x := 2 < x = 3 < x := 3$ is not a linearisation of the program order but is TSO-consistent with it.

When we have a write followed by a read in the program order, but swapped in the linear order, as in this example, we can imagine the write having gotten stuck in the write buffer, and observing the read before the write.

3 A Denotational Account

So far we have dealt with program orders as an abstract concept. However, there exist program orders having no TSO-consistent orders. Consider, for example, the program order $\{x = 1 < x = 2\}$. We therefore restrict our attention to program orders for well-defined programs in the simple imperative language given below. These program orders are defined in Section 3.2.

Restricting our attention to program orders of well-defined programs raises the question of *compositionality*. The key is to find a way to derive TSO-consistent orders for a sequential composition $c_1; c_2$ or parallel composition $c_1 \parallel c_2$ given TSO-

consistent orders for c_1 and c_2. This is infeasible with the axiomatic approach, which requires reasoning about whole programs and is inherently non-compositional. In contrast, a denotational approach using pomsets is compositional: it allows us to derive the meaning of a program from the meaning of its parts.

Our denotational semantics has two components. The first associates to each program a set of *TSO pomsets*, which serves as the *abstract meaning* or *denotation* of the program. This component is described in Section 3.3. The second associates to each pomset a set of *executions*, which describe its input-output behaviours. This is described in Section 3.4.2.

3.1 A Simple Imperative Language

We express our programs using a simple imperative language. This formalism avoids the complexity of high-level languages, while still capturing the programs we are interested in. In the syntax below, e ranges over integer expressions, b over boolean expressions, c over commands, and p over programs. We distinguish between commands and programs, because although commands can be composed to form new commands, programs are assumed to be syntactically complete and executable.

$$
\begin{aligned}
v &::= \ldots, -2, -1, 0, 1, 2, \ldots \\
e &::= v \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid \cdots \\
b &::= \mathit{true} \mid \mathit{false} \mid \neg b \mid e_1 = e_2 \mid e_1 < e_2 \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid \cdots \\
c &::= \mathbf{skip} \mid x := e \mid c_1; c_2 \mid c_1 \parallel c_2 \mid \mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ b\ \mathbf{do}\ c \\
p &::= c
\end{aligned}
$$

Let VExp denote the set of integer expressions, BExp the set of boolean expressions, and Cmd the set of commands.

3.2 PO Pomsets

Given a command c in our language, we must now compile it down to its set $\mathcal{P}_{PO}(c)$ of program order pomsets. We need operations for the sequential and parallel composition of pomsets over the same set of labels. The **sequential composition** $(P_0, \leq_0, \Phi_0); (P_1, \leq_1, \Phi_1)$ of pomsets is $(P_0, \leq_0, \Phi_0)$ whenever P_0 is infinite, and otherwise is $(P_0 \uplus P_1, \leq, \Phi)$ where $P_0 \uplus P_1 = \{0\} \times P_0 \cup \{1\} \times P_1$ is the disjoint union, $(i, a) \leq (j, b)$ if and only if $i = j$ and $a \leq_i b$, or $i = 0$ and $j = 1$, and $\Phi((i, a)) = \Phi_i(a)$. The **parallel composition** $(P_0, \leq_0, \Phi_0) \parallel (P_1, \leq_1, \Phi_1)$ of pomsets is $(P_0 \uplus P_1, \leq, \Phi)$ where $(i, a) \leq (j, b)$ if and only if $i = j$ and $a \leq_i b$, and $\Phi((i, a)) = \Phi_i(a)$. The empty pomset $\mathbf{0} = (\varnothing, \varnothing, \varnothing \to L)$ is the unit for sequential and parallel composition. Given a pomset P on a set of labels L and a subset $L' \subseteq L$, the **restriction** $P{\restriction_{L'}}$ of P to L' is the pomset on $\Phi^{-1}(L')$ whose ordering is induced by P. The **deletion** of L' from P is $P{\restriction_{L \setminus L'}}$. We lift these operations to sets of pomsets in the obvious manner.

Because δ has no effects, we identify program orders P and P' whenever there exists a non-empty pomset P_δ that can be obtained by deleting a finite number of δ actions from P and also by deleting a finite number of δ actions from P'.

For expository convenience, we identify lists and *linear* pomsets, where we call a pomset linear if its underlying poset is linear. Explicitly, we identify $[\,]$ with $\mathbf{0}$ and $[\,\lambda_1, \ldots, \lambda_n\,]$, with the pomset $\{\lambda_1\}; \cdots; \{\lambda_n\}$. To minimise notation, we leverage this identification and write $L; L'$ to denote the concatenation of the lists L and L'.

We begin with the program order denotation of expressions. To each expression e, we assign a set $\mathcal{P}_{PO}(e)$ of tuples of program orders and corresponding values:

$$\begin{aligned}\mathcal{P}_{PO} &: \mathsf{VExp} \to \wp(\mathrm{Pom}(\mathcal{A}_{PO}) \times V)\\ \mathcal{P}_{PO}(v) &= \{(\{\delta\}, v)\}\\ \mathcal{P}_{PO}(x) &= \{(\{x = v\}, v) \mid v \in V\}\\ \mathcal{P}_{PO}(e_1 \odot e_2) &= \{(P_1 \parallel P_2, v_1 \odot v_2) \mid (P_i, v_i) \in \mathcal{P}_{PO}(e_i)\}\end{aligned}$$

where $\odot$ ranges over binary operations. Read expressions x are associated with arbitrary values in V for reasons of compositionality: we do not know with which writes the read may eventually be composed, and so we need to permit reading arbitrary values. We chose to evaluate binary operations $e_1 \odot e_2$ in parallel; one could just as legitimately have chosen to sequentialise the evaluation and written $P_1; P_2$. We assume $v_1 \odot v_2 \in V$ to be the result of applying the binary operation $\odot$ to v_1 and v_2. We handle program orders for unary expressions analogously, and assume $\neg b'$ is the result of negating the boolean value b'. To simplify the clauses involving conditionals, we give helper functions $\mathcal{P}_{\mathit{true}}$ and $\mathcal{P}_{\mathit{false}}$ to pomsets corresponding to the given boolean values.

$$\begin{aligned}\mathcal{P}_{PO} &: \mathsf{BExp} \to \wp(\mathrm{Pom}(\mathcal{A}_{PO}) \times \mathrm{Bool})\\ \mathcal{P}_{PO}(b) &= \{(\{\delta\}, b)\} \qquad (b \in \{\mathit{true}, \mathit{false}\})\\ \mathcal{P}_{PO}(\neg b) &= \{(P, \neg b') \mid (P, b') \in \mathcal{P}_{PO}(e)\}\\ \mathcal{P}_{PO}(e_1 \odot e_2) &= \{(P_1 \parallel P_2, v_1 \odot v_2) \mid (P_i, v_i) \in \mathcal{P}_{PO}(e_i)\}\\ \mathcal{P}_{\mathit{true}}(b) &= \{P \mid (P, \mathit{true}) \in \mathcal{P}_{PO}(b)\}\\ \mathcal{P}_{\mathit{false}}(b) &= \{P \mid (P, \mathit{false}) \in \mathcal{P}_{PO}(b)\}\end{aligned}$$

Note that in the case of boolean binary operations, the e_i might be integer or boolean expressions, and the corresponding semantic clause for $\mathcal{P}_{PO}(e_i)$ should be used.

We give the program order denotation of commands in a similar manner, this time associating sets of program orders to each command phrase:

$$\begin{aligned}\mathcal{P}_{PO} &: \mathsf{Cmd} \to \wp(\mathrm{Pom}(\mathcal{A}_{PO}))\\ \mathcal{P}_{PO}(\mathbf{skip}) &= \{\{\delta\}\}\\ \mathcal{P}_{PO}(x := e) &= \{P; \{x := v\} \mid (P, v) \in \mathcal{P}_{PO}(e)\}\\ \mathcal{P}_{PO}(c_1; c_2) &= \mathcal{P}_{PO}(c_1); \mathcal{P}_{PO}(c_2)\\ \mathcal{P}_{PO}(c_1 \parallel c_2) &= \mathcal{P}_{PO}(c_1) \parallel \mathcal{P}_{PO}(c_2)\\ \mathcal{P}_{PO}(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) &= \mathcal{P}_{\mathit{true}}(b); \mathcal{P}_{PO}(c_1) \cup \mathcal{P}_{\mathit{false}}(b); \mathcal{P}_{PO}(c_2)\\ \mathcal{P}_{PO}(\mathbf{while}\ b\ \mathbf{do}\ c) &= \bigcup_{n=0}^{\infty} I^n(b, c) \cup I^\omega(b, c),\end{aligned}$$

where

$$\begin{aligned}I^0(b, c) &= \mathcal{P}_{\mathit{false}}(b)\\ I^{n+1}(b, c) &= \mathcal{P}_{\mathit{true}}(b); \mathcal{P}_{PO}(c); I^n(b, c)\end{aligned}$$

The only interesting clause is for **while** b **do** c. Here, we take union of all of the finite unrollings $I^n(b, c)$ of the loop. We must also consider the case of an infinite loop. This is captured by $I^\omega(b, c)$, which describes the infinite pomset obtained by unrolling the loop arbitrarily many times. The **while** b **do** c clause also illustrates why we associate the pomset $\{\delta\}$ instead of $\mathbf{0}$ to values: otherwise, we would have $\mathcal{P}_{PO}(\mathbf{while}\ \mathit{false}\ \mathbf{do}\ c) = \{\mathbf{0}\}$, and this would break both our input-output executions

and our intuition that this program should be denotationally equivalent to **skip**.

To illustrate the above semantic clauses, we return to the Dekker program from the introduction. This program has pomsets of each of the following forms, for each choice of $v \neq 0$ and $v' \neq 0$:

$$\begin{array}{cc} x := 1 & y := 1 \\ \downarrow & \downarrow \\ y = 0 & x = 0 \\ \downarrow & \downarrow \\ z := 1 & w := 1 \end{array}, \qquad \begin{array}{cc} x := 1 & y := 1 \\ \downarrow & \downarrow \\ y = v & x = 0 \\ & \downarrow \\ & w := 1 \end{array}, \qquad \begin{array}{cc} x := 1 & y := 1 \\ \downarrow & \downarrow \\ y = 0 & x = v \\ \downarrow & \\ z := 1 & \end{array}, \qquad \begin{array}{cc} x := 1 & y := 1 \\ \downarrow & \downarrow \\ y = v & x = v' \end{array}.$$

The first program order describes an execution where we read both $y = 0$ and $x = 0$ and where Dekker fails. The next three forms of program orders describe executions in which one or both reads obtain a non-zero value.

3.3 TSO Pomsets

As described above, we wish to assign a set $\mathcal{P}_{TSO}(c)$ of *TSO pomsets* to each command c. Doing so will involve the careful modelling of write buffers. For compactness, we will write $\mathcal{P}$ instead of $\mathcal{P}_{TSO}$ in this section's semantic clauses.

We introduce a set $\mathrm{BLoc} = \{\bar{x} \mid x \in \mathrm{Loc}\}$ of buffer locations and let the set of buffer write actions be $\mathcal{A}_b = \{\bar{x} := v \mid \bar{x} \in \mathrm{BLoc},\ v \in V\}$. An action $\bar{x} := v$ by a thread denotes adding a write $x := v$ to the thread's write buffer. The set of TSO actions $\mathcal{A}_{TSO}$ then consists of $\mathcal{A}_{PO}$ extended with $\mathcal{A}_b$. A *TSO pomset* will then be a pomset in $\mathrm{Pom}(\mathcal{A}_{TSO})$ satisfying the finite height property.

To capture the effects of buffers, we parametrise our semantic clauses with lists of global write actions, which represent the writes currently in our buffer. We let $\mathsf{Ls} = \mathcal{A}_w\ \mathsf{list}$ be the set of all such lists. The intuition is that write buffers behave as queues under TSO, and we can use a list $L \in \mathsf{Ls}$ to model a queue by dequeuing from the head of the list and enqueuing at the end of the list. We use lists to model queues rather than some abstract data structure because of our convenient identification between lists and linear pomsets.

The semantic clauses are given in two strata. The semantic clauses $\mathcal{B}_L$ for "basic TSO pomsets" capture the meaning of the syntactic phrases in a manner very similar to the program order definitions in Section 3.2. $\mathcal{B}_L$ assigns to each command phrase a set of pairs of TSO pomsets and buffer lists. The pomset component captures the meaning of the phrase in the presence of the buffer L, while the buffer component captures the state of the buffer after performing the actions associated with the phrase. In the second stratum, we use $\mathcal{P}_L$ clauses to capture the meaning of the phrase in the presence of buffer flushing. Flushing a write consists of dequeuing a global write $x := v$ from L and inserting it in the pomset. $\mathcal{P}_L$ again takes command phrases to subsets of $\mathrm{Pom}(\mathcal{A}_{TSO}) \times \mathsf{Ls}$.

To generate TSO pomsets, we modify the semantic clauses generating program orders in four key places to get our basic pomsets. The first is for write commands $x := e$. Starting from a buffer $L \in \mathsf{Ls}$, we get the pomset P and associated value v for e using $\mathcal{P}_L(e)$. The buffer L may have changed to a buffer L' while we were evaluating e, and $\mathcal{P}_L(e)$ also gives us this L'. Instead of immediately making a global write to x as we would have in the program order clause, we enqueue the global write on the buffer L':

$$\mathcal{B}_L(x := e) = \{(P; \{\bar{x} := v\}, L'; \{x := v\}) \mid (P, v, L') \in \mathcal{P}_L(e)\}.$$

We must also change the semantic clauses for read expressions. By axiom (VB), whenever we read from a location x, we must use the most recent value available for it in the write buffer, if available. We use the following helper function to convert a buffer $L \in \mathsf{Ls}$ to a partial function $\beta_L : \mathrm{Loc} \rightharpoonup_{fin} V$ giving us the value of the most recent write in L to a given location:

$$\beta_{[\,]}(x) = \mathit{undefined}, \qquad \beta_{L;\{x:=v\}}(y) = \begin{cases} v & \text{if } x = y \\ \beta_L(y) & \text{otherwise.} \end{cases}$$

Then, the semantic clause giving us the basic pomsets for reads is

$$\mathcal{B}_L(x) = \{(\{x = v\}, v, L) \mid \beta_L(x) = v\} \cup \{(\{x = v\}, v, L) \mid x \notin \mathrm{dom}(\beta_L), v \in V\}.$$

The first part tells us to use the value associated with x in the buffer L, if available. The second part uses arbitrary values if unavailable, as with program orders.

The third major change involves parallel composition. We explain parallel composition of expressions; parallel composition of commands is analogous. By axioms (F) and (J), we must flush our buffers before every fork and join. We therefore begin by flushing our entire buffer, i.e., by taking L and placing it at the beginning of our pomset. Having flushed the buffer, we then evaluate the e_i with empty buffers and get back pomsets P_i and v_i. Because we can only join threads if their buffers are empty, we require that these P_i and v_i be associated with empty buffers in $\mathcal{P}_{[\,]}(e_i)$. We then proceed as for the program order, and add the parallel composition of the P_i to our pomset, and compute the value $v_1 \odot v_2$. Because we just joined two empty buffers, our resulting buffer is empty:

$$\mathcal{B}_L(e_1 \odot e_2) = \{(L; (P_1 \parallel P_2), v_1 \odot v_2, [\,]) \mid (P_i, v_i, [\,]) \in \mathcal{P}_{[\,]}(e_i)\}.$$

Finally, when we sequentially compose two commands c_1 and c_2 (assuming no forking or joining), c_2 continues executing from the buffer c_1 finished with. We express this using the polymorphic helper function $\curvearrowright$:

$$\curvearrowright : \forall A.\forall B.\wp(\mathrm{Pom} \times A) \to (A \to \wp(\mathrm{Pom} \times B)) \to \wp(\mathrm{Pom} \times B)$$
$$S \curvearrowright f = \{(P; P', b) \mid (P, a) \in S \wedge (P', b) \in f(a)\}$$

Taking $A = B = \mathsf{Ls}$, sequential composition can be expressed using as

$$\mathcal{B}_L(c_1; c_2) = \mathcal{P}_L(c_1) \curvearrowright \mathcal{P}_{\_}(c_2),$$

where we take $\mathcal{P}_{\_}(c)$ to be a function of type $\mathsf{Ls} \to \wp(\mathrm{Pom}(\mathcal{A}_{TSO}) \times \mathsf{Ls})$. Explicitly, this means $\mathcal{B}_L(c_1; c_2) = \{(P_1; P_2, L_2) \mid (P_1, L_1) \in \mathcal{P}_L(c_1), (P_2, L_2) \in \mathcal{P}_{L_1}(c_2)\}$. This idiom of chaining pairs of pomsets and buffers together using $\curvearrowright$ will be useful throughout. We make $\curvearrowright$ polymorphic so that we can handle, e.g., the case of $A = \mathsf{Ls}$ and $B = V \times \mathsf{Ls}$ below.

The remainder of the basic clauses are analogous to those for program order pomsets, subject to the modifications described above:

$$\mathcal{B} : \mathsf{VExp} \to \mathsf{Ls} \to \wp(\mathrm{Pom}(\mathcal{A}_{TSO}) \times V \times \mathsf{Ls})$$
$$\mathcal{B}_L(v) = \{(\{\delta\}, v, L)\}$$
$$\mathcal{B} : \mathsf{BExp} \to \mathsf{Ls} \to \wp(\mathrm{Pom}(\mathcal{A}_{TSO}) \times \mathrm{Bool} \times \mathsf{Ls})$$
$$\mathcal{B}_L(\neg e) = \{(P, \neg b, L') \mid (P, b, L') \in \mathcal{B}_L(e)\}$$
$$\mathcal{P}_{L,\mathit{true}}(b) = \{(P, L') \mid (P, \mathit{true}, L') \in \mathcal{P}_L(b)\}$$
$$\mathcal{P}_{L,\mathit{false}}(b) = \{(P, L') \mid (P, \mathit{false}, L') \in \mathcal{P}_L(b)\}$$

$$\begin{aligned}
\mathcal{B} &: \mathsf{Cmd} \to \mathsf{Ls} \to \wp(\mathrm{Pom}(\mathcal{A}_{TSO}) \times \mathsf{Ls}) \\
\mathcal{B}_L(\mathbf{skip}) &= \{(\{\delta\}, L)\} \\
\mathcal{B}_L(c_1 \parallel c_2) &= \{(L; (P_1 \parallel P_2), [\,]) \mid (P_i, [\,]) \in \mathcal{P}_{[\,]}(c_i)\} \\
\mathcal{B}_L(\mathbf{if}\ b\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2) &= \mathcal{P}_{L,true}(b) \frown \mathcal{P}_-(c_1) \cup \mathcal{P}_{L,false}(b) \frown \mathcal{P}_-(c_2) \\
\mathcal{B}_L(\mathbf{while}\ b\ \mathbf{do}\ c) &= \bigcup_{n=0}^{\infty} I_L^n(b,c) \cup I_L^\omega(b,c)
\end{aligned}$$

where

$$\begin{aligned}
I_L^0(b,c) &= \mathcal{P}_{L,false}(b) \\
I_L^{n+1}(b,c) &= \mathcal{P}_{L,true}(b) \frown \mathcal{P}_-(c) \frown I_-^n(b,c)
\end{aligned}$$

We now turn our attention to flushing. The intent is that a thread can flush arbitrarily many of its writes at any point in its execution. Thus, the pomsets associated with flushes for a buffer L are the prefixes L' of L, and the resulting buffers are the remainders of L. We use $\mathrm{split}(L)$ to denote these prefix-suffix pairs:

$$\begin{aligned}
&\mathrm{split} : \mathsf{Ls} \to \wp(\mathrm{Pom}(\mathcal{A}_{TSO}) \times \mathsf{Ls}) \\
&\mathrm{split}(L) = \{(L', L'') \mid L = L'; L''\}
\end{aligned}$$

We introduce a variant of $\frown$ to cope with triples of pomsets, values, and buffers, and will rely on types to disambiguate the version needed in any given situation:

$$\begin{aligned}
&\frown : \forall A.\forall B.\wp(\mathrm{Pom} \times A \times B) \to (B \to \wp(\mathrm{Pom} \times B)) \to \wp(\mathrm{Pom} \times A \times B) \\
&S \frown f = \{(P; P', A, B') \mid (P, A, B) \in S \wedge (P', B') \in f(B)\}
\end{aligned}$$

We define the TSO pomsets $\mathcal{P}$ in terms of $\mathcal{B}$ and split. $\mathcal{P}$ composes split and $\mathcal{B}$ in a manner that we can flush some writes from the buffer, then evaluate e or perform c, and then flush some writes at the end:

$$\begin{aligned}
\mathcal{P}_L(e) &= \mathrm{split}(L) \frown \mathcal{B}_-(e) \frown \mathrm{split} \\
\mathcal{P}_L(c) &= \mathrm{split}(L) \frown \mathcal{B}_-(c) \frown \mathrm{split}
\end{aligned}$$

We can validate various expected equivalences by unfolding these definitions. For example, sequential composition of commands is associative, because $\mathcal{P}_L(c_1; (c_2; c_3)) = \mathrm{split}(L) \frown \mathcal{B}_-(c_1) \frown \mathrm{split} \frown \mathcal{B}_-(c_2) \frown \mathrm{split} \frown \mathcal{B}_-(c_3) \frown \mathrm{split} = \mathcal{P}_L((c_1; c_2); c_3)$. Using the identity $\mathbf{0}; P = P$ and the fact that parallel composition of pomsets is associative, one can show that parallel composition of commands is associative, i.e., that $\mathcal{P}_L(c_1 \parallel (c_2 \parallel c_3)) = \mathcal{P}_L((c_1 \parallel c_2) \parallel c_3)$. The parallel composition of pomsets commutes, so the parallel composition of commands commutes, i.e., $\mathcal{P}_L(c_1 \parallel c_2) = \mathcal{P}_L(c_2 \parallel c_1)$.

Because we expect programs to be run from an empty buffer and to only stop after emptying all buffers, we let the set of **TSO pomsets** associated with a program p be given by $\mathcal{P}_{TSO}(p) = \{P \mid (P, [\,]) \in \mathcal{P}_{[\,]}(p)\}$.

We illustrate the constructions by giving four example families of TSO pomsets for the Dekker program from the introduction, again assuming $v \neq 0$ and $v' \neq 0$:

$$\begin{array}{cc} \bar{x} := 1 & \bar{y} := 1\ , \\ \downarrow & \downarrow \\ x := 1 & y := 1 \\ \downarrow & \downarrow \\ y = 0 & x = 0 \\ \downarrow & \downarrow \\ \bar{z} := 1 & \bar{w} := 1 \\ \downarrow & \downarrow \\ z := 1 & w := 1 \end{array} \qquad \begin{array}{cc} \bar{x} := 1 & \bar{y} := 1\ , \\ \downarrow & \downarrow \\ y = 0 & x = 0 \\ \downarrow & \downarrow \\ x := 1 & y := 1 \\ \downarrow & \downarrow \\ \bar{z} := 1 & \bar{w} := 1 \\ \downarrow & \downarrow \\ z := 1 & w := 1 \end{array} \qquad \begin{array}{cc} \bar{x} := 1 & \bar{y} := 1\ , \\ \downarrow & \downarrow \\ x := 1 & y := 1 \\ \downarrow & \downarrow \\ y = v & x = 0 \\ & \downarrow \\ & \bar{w} := 1 \\ & \downarrow \\ & w := 1 \end{array} \qquad \begin{array}{cc} \bar{x} := 1 & \bar{y} := 1\ . \\ \downarrow & \downarrow \\ y = v & y := 1 \\ \downarrow & \downarrow \\ x := 1 & x = v' \end{array}$$

In the first family of pomsets, we flush the writes immediately after inserting them in the buffers, while in the second, we flush the writes to x and y after reading y and x. In the third family, we flush x right after placing its write in the buffer, but fall into the *false* case of the conditional after reading some value $v \neq 0$, thus taking the **skip** branch. In the fourth pomset, we read y after placing the write $x := 1$ in the buffer, but before it gets flushed, and both threads fall into the **skip** branch.

3.4 Executions

Our TSO pomset semantics gives an abstract account capturing families of possible executions. However, compositionality comes with its price: we associate to programs some pomsets that cannot in any real sense be "executed". Consider for example, the pomset $x := 2 \to \bar{x} := 2 \to x = 1 \to \cdots$ for the program $c = (x := 2;\ \textbf{if}\ x = 1\ \textbf{then}\ c_1\ \textbf{else}\ c_2)$. In no circumstance do we expect to execute c_1 when this program is run alone, and so the above TSO pomset has, in a sense made precise later, no executional meaning. However, compositionality requires this pomset be associated with the command c, because one *could* execute c_1 if our program were instead $c \parallel x := 1$. Our notion of *execution* filters out these pomsets with no executional meaning and yields an input/output behaviour for programs built from their pomset semantics.

3.4.1 Buffered States

Our notion of execution requires the concept of a buffered global state, i.e., a global state with a write buffer per thread. We execute threads individually. Each thread's execution starts from a state with a buffer, which in combination reflect that thread's view of shared memory. Let $\mathrm{Locs} = \mathrm{BLoc} \cup \mathrm{Loc}$ be the set of all locations. To model the combination of a global state and a buffer, we use elements of $\Sigma = (\mathrm{BLoc} \rightharpoonup_{fin} (V \times \mathbb{N})/\!\approx) \times (\mathrm{Loc} \rightharpoonup_{fin} V)$, where $\approx$ is the least equivalence relation generated by $(v, 0) \approx (v', 0)$ for all $v, v' \in V$. Because Loc and BLoc are disjoint, we can and will identify Σ with the its obvious inclusion in $\mathrm{Locs} \rightharpoonup_{fin} ((V \times \mathbb{N})/\!\approx \cup V)$. The intuition is that if $\sigma(\bar{x}) = (v, n)$, then there are n writes to x in σ's write buffer, and the most recent write to σ was the value v. We need to keep track of the number n of writes to x still in the buffer to know whether we should continue reading x from the buffer after a flush. We identify $(v, 0)$ and $(v', 0)$ for all v and v' because one should not be able to observe a value for a write that is no longer in the buffer, and this identification allows us to "forget" the value by setting n to 0. For $x_i \in \mathrm{Locs}$ and v_i in the corresponding subset of $((V \times \mathbb{N})/\!\approx) \cup V$, we denote by $[x_1 : v_1, \ldots, x_n : v_n]$ the buffered state with graph $\{(x_1, v_1), \ldots, (x_n, v_n)\}$. For compactness of notation, we write v_n for the equivalence class of (v, n) in $(V \times \mathbb{N})/\!\approx$.

3.4.2 Footprints

Footprints are the first step towards filtering out pomsets with no executional meaning. Informally, a footstep $(\sigma, \tau) \in \Sigma \times \Sigma$ of an action λ is a minimal piece of state σ required to be able to perform λ, and a description τ of the effects of performing λ. For example, to perform a global write $x := v$, we need to have x in the domain of the initial state and present in the buffer, so $\sigma = [x : v', \bar{x} : v''_{n+1}]$ for some v' and v'', and the result is setting the global value of x to v while removing one occurrence of x from the buffer, so $\tau = [x : v, \bar{x} : v''_n]$. Though v and v'' are unrelated,

this gives the correct behaviour in the context of command pomsets because global writes to x occur in the same order as buffer writes to x. To perform a read action $x = v$, we must either have no entries for x in the buffer and have $x : v$ in the global state, or we must have x in the buffer with value v, i.e., $\bar{x} : v_n$ for some $n > 0$. We call the set of footsteps associated with an action its *footprint*. Similarly, pomsets have footsteps and footprints.

TSO footprints for actions are given as follows:

$$\begin{aligned}
[\![x = v]\!] &= \{([x : v, \bar{x} : v'_0], [\,]), ([\bar{x} : v_{n+1}], [\,]) \mid v' \in V \wedge n \in \mathbb{N}\} \\
[\![\bar{x} := v]\!] &= \{([\bar{x} : v'_n], [\bar{x} : v_{n+1}]) \mid v' \in V \wedge n \in \mathbb{N}\} \\
[\![x := v]\!] &= \{([x : v', \bar{x} : v''_{n+1}], [x : v, \bar{x} : v''_n]) \mid v', v'' \in V \wedge n \in \mathbb{N}\} \\
[\![\delta]\!] &= \{([\,], [\,])\}
\end{aligned}$$

To give footprints to pomsets, we need to know when it makes sense to combine two footprints sequentially or in parallel. We say two states σ_1 and σ_2 are **consistent**, $\sigma_1 \Uparrow \sigma_2$, if for all $x \in \mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2)$, $\sigma_1(x) = \sigma_2(x)$. In this case, $\sigma_1 \cup \sigma_2$ is also a state. Let $\sigma \setminus \mathrm{dom}(\tau) = \sigma\restriction_{\mathrm{dom}(\sigma)\setminus\mathrm{dom}(\tau)}$. Then the result of updating σ by τ is $[\sigma \mid \tau] = (\sigma \setminus \mathrm{dom}(\tau)) \cup \tau$. For subsets S_1 and S_2 of $\Sigma \times \Sigma$, we define the associative operation $\triangleleft$ to be:

$$S_1 \triangleleft S_2 = \{(\sigma_1 \cup (\sigma_2 \setminus \mathrm{dom}(\tau_1)), [\tau_1 \mid \tau_2]) \mid (\sigma_i, \tau_i) \in S_i \wedge [\sigma_1 \mid \tau_1] \Uparrow \sigma_2\}.$$

To account for global writes occurring elsewhere during the program, we parametrise the clauses assigning footprints to actions and pomsets by a list Λ containing a linearisation of the pomset as a subsequence, combined with any number of other global writes that represent flushes from foreign buffers. Formally, given a pomset P, we let $\mathrm{Lin}(P)$ be the set of its linearisations. Then the clauses are parametrised by $\Lambda \in \Gamma(P) = \{\mathrm{Lin}(P \parallel L) \mid L \in \mathsf{Ls}\}$, where given some global-write environment $\Lambda \in \Gamma(P)$, we identify P with its image in Λ. Given some $L \in \mathsf{Ls}$, let $[\![L]\!]^*$ be inductively defined on L as follows:

$$\begin{aligned}
[\![\,[\,]\,]\!]^* &= \{([\,], [\,])\} \\
[\![x := v :: L]\!]^* &= \{([x : v'], [x : v]) \mid v' \in V\} \triangleleft [\![L]\!]^*
\end{aligned}$$

The intuition here is that a foreign buffer flush should only affect the global part of the state and have no effect on our buffer.

We will need to know if buffered states have empty buffers. We let $\zeta(\sigma)$ hold if and only if for all $x \in \mathrm{dom}(\sigma\restriction_{\mathrm{BLoc}})$, $\sigma(x) = v_0$ for some v. Thus, for example, both $\zeta([\,])$ and $\zeta([x : 1, \bar{y} : 2_0])$, but neither $\zeta([x : 1, \bar{x} : 2_5])$ nor $\zeta([\bar{y} : 1_1])$.

The footprint $[\![P]\!]_\Lambda$ of a pomset P under a global-write environment $\Lambda \in \Gamma(P)$ is the set inductively defined according to the three following rules:

(Act) If $P = \{\lambda\}$ for some action λ and $\Lambda = \Lambda_1; P; \Lambda_2$ for some $\Lambda_1, \Lambda_2 \in \mathsf{Ls}$, then $[\![P]\!]_\Lambda = [\![\Lambda_1]\!]^* \triangleleft [\![\lambda]\!] \triangleleft [\![\Lambda_2]\!]^*$.

(Seq) If $P = P_1; P_2$ and $\Lambda = \Lambda_1; \Lambda_2$, then $[\![P_1]\!]_{\Lambda_1} \triangleleft [\![P_2]\!]_{\Lambda_2} \subseteq [\![P]\!]_\Lambda$.

(Par) If $P = P_1 \parallel P_2$, Λ_1 is the result of deleting the read and buffer write actions of P_2 from Λ, Λ_2 is the symmetric restriction, $(\sigma_i, \tau_i) \in [\![P_i]\!]_{\Lambda_i}$, $\zeta(\sigma_i)$, $\zeta(\tau_i)$ $(i = 1, 2)$, and $\sigma_1 \Uparrow \sigma_2$, then $(\sigma_1 \cup \sigma_2, \tau_1 \cup \tau_2) \in [\![P]\!]_\Lambda$.

This definition is inspired by Lamport's "happened before" relation [9]. In the case of (Act), for a given $P = \{\lambda\}$ and $\Lambda \in \Gamma(P)$, there exists a unique splitting of Λ

into Λ_1 and Λ_2, and the intuition is that Λ specifies that the global writes in Λ_1 appeared before λ, and that λ was followed by some global writes in Λ_2. In the case of (Seq), for $[\![P_i]\!]_{\Lambda_i}$ to be well-defined, we are implicitly assuming that $\Lambda_i \in \Gamma(P_i)$. (Seq) tells us that the result of sequentially executing a program in the presence of global writes should be the same as executing the pieces sequentially in the presence of the appropriate subset of global writes. Finally, in (Par), the restrictions of Λ are such that both parallel components observe *all* writes in the same order, and this is how we simulate the effects of writes to a global state.

Because global writes are observed in the same order, we can show by induction on the derivation of a footstep that the second component of the footsteps of a pomset is determined by the global-write environment whenever its buffer is empty. This validates the intuition that the final state should be determined by the total order imposed on the writes:

Proposition 3.1 *For all P, $\Lambda \in \Gamma(P)$, and $(\sigma, \tau) \in [\![P]\!]_\Lambda$ such that $\zeta(\tau)$, we have $\tau\restriction_{\mathrm{Loc}} = \beta_{\Lambda\restriction_{\mathcal{A}_w}}$.*

3.4.3 Executions

We say that a finite pomset P is **TSO executable** if there exists a $\Lambda \in \mathrm{Lin}(P)$ such that $[\![P]\!]_\Lambda$ is non-empty. The set of **TSO executions** of a finite pomset P is given by the set $\mathcal{E}(P) = \{(\sigma, [\sigma \mid \tau]) \mid \Lambda \in \mathrm{Lin}(P), (\sigma', \tau) \in [\![P]\!]_\Lambda, \sigma' \subseteq \sigma, \zeta(\sigma), \zeta(\tau)\}$; we take $\mathcal{E}(P) = \varnothing$ if P is infinite. These executions take all of the states σ containing a minimal fragment σ' required to execute P to a state updated with the effects τ of P. The set of TSO executions for a program p is then $\mathcal{E}(p) = \bigcup_{P \in \mathcal{P}_{TSO}(p)} \mathcal{E}(P)$.

We illustrate TSO pomset executions by validating the IRIW litmus test, i.e., by showing that all writes appear in the same order to all threads. For example, starting from a state initialised to zero, executing the program

$$x := 1 \parallel y := 1 \parallel (w_1 := x; w_2 := y) \parallel (z_1 := y; z_2 := x)$$

under TSO should never give a state consistent with $[w_1 : 1, w_2 : 0, z_1 : 1, z_2 : 0]$. To show this, it is sufficient to show the following pomset P is not executable:

$$\begin{array}{cccc} \bar{x} := 1 & \bar{y} := 1 & x = 1 & y = 1 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ x := 1 & y := 1 & y = 0 & x = 0. \end{array}$$

Consider some $\Lambda \in \mathrm{Lin}(P)$. Without loss of generality, assume $x := 1 <_\Lambda y := 1$. To get an execution, we must apply (Par), and eventually we will need to compute $[\![P_4]\!]_{\Lambda_4}$ where P_4 is $y = 1 \to x = 0$ and Λ_4 is such that $x := 1 <_{\Lambda_4} y := 1$. To be able to execute $y = 1$ and still get a footstep with an empty initial buffer, we need Λ_4 to satisfy $y := 1 < y = 1$. But then $[\![P_4]\!]_{\Lambda_4} = [\![[\, x := 1, y := 1\,]]\!]^* \triangleleft [\![\{y = 1\}]\!]_{[\,y=1\,]} \triangleleft [\![\{x = 0\}]\!]_{[\,x=0\,]}$, and there are no footsteps in $[\![[\, x := 1, y := 1\,]]\!]^* \triangleleft [\![\{y = 1\}]\!]_{[\,y=1\,]}$ that can be combined with those in $[\![\{x = 0\}]\!]_{[\,x=0\,]}$ to get states with empty buffers. This means we cannot combine footsteps from P_4 to get footsteps for P using the (Par) rule, and so $[\![P]\!]_\Lambda$ will be empty.

As discussed in the introduction, the Dekker mutual exclusion algorithm fails under TSO. Indeed, the second pomset for Dekker on page 9 can be executed from an initial state having both x and y set to zero. To do so, we take a Λ such that $y = 0 <_\Lambda y := 1$ and $x = 0 <_\Lambda x := 1$, and apply (Par) followed by (Seq).

In contrast, the Peterson algorithm successfully enforces mutual exclusion under TSO. Consider the following instance of the Peterson algorithm:

$$(x := 1; \textbf{if } x = 2 \textbf{ then } l := 1 \textbf{ else skip}) \parallel (x := 2; \textbf{if } x = 1 \textbf{ then } r := 1 \textbf{ else skip}).$$

Starting from the initial state $[x : 0, l : 0, r : 0]$, one cannot execute the above under TSO and reach a state where both l and r are 1. In showing this, we can safely ignore all pomsets where a read from x appears before the global write to x, because whenever we have $\bar{x} := v \to x = v' \to x := v$ in a command's TSO pomset, we must have $v = v'$. This implies that if a thread reads x before it does the global write to x, it will take the **skip** branch of the conditional. It is then sufficient to show that the following pomset is not executable:

$$\bar{x} := 1 \longrightarrow x := 1 \longrightarrow x = 2$$

$$\bar{x} := 2 \longrightarrow x := 2 \longrightarrow x = 1.$$

Consider some $\Lambda \in \mathrm{Lin}(P)$. Without loss of generality, assume $x := 1 <_\Lambda x := 2$. To get an execution, we must apply (Par) and derive a footstep for the bottom row P_2 under some Λ_2 where $x := 1 <_{\Lambda_2} x := 2$. To compute this footstep, we must repeatedly apply (Seq), and will eventually reach the stage where $[\![P_2]\!]_{\Lambda_2} = \{([x : 0], [x : 2])\} \triangleleft [\![x = 1]\!]_{[x=1]}$. But this footprint must be empty, because $[x : 0 \mid x : 2]$ is not consistent with $[x : 1]$. We thus cannot apply (Par) and we conclude that the pomset is not executable. It follows that the Peterson algorithm enforces mutual exclusion under TSO.

3.5 Fences

We can extend the above semantics to deal with fences. This extension will not be referenced in subsequent sections, and we include it here merely to emphasise the flexibility of our general development.

A *fence* constrains the reordering of memory actions. To capture fences, we first introduce a command **fence**. Under TSO, fences cause all actions before the fence to be observed before any actions after the fence. It is sufficient to flush the thread's buffer to ensure this, giving rise to the semantic clause $\mathcal{P}_L(\textbf{fence}) = \{(L; \{\delta\}, [\,])\}$.

4 Soundness and Completeness

We show that our denotational account of TSO in Section 3 is sound and complete relative to the axiomatic account of Section 2. Soundness implies we capture only behaviours permitted by the axiomatic account; completeness implies we capture all behaviours permitted by the axiomatic account. Because all TSO-consistent orders are contained in TSO-consistent total orders and can be obtained by weakening these, it is sufficient to show that we capture all TSO-consistent total orders. We identify total orders and lists.

4.1 Soundness

We call a function $f : \mathrm{Pom}(\mathcal{A}_{PO}) \to \wp(\mathcal{A}_{PO}\ \mathsf{list})$ **sound** when for every program p and finite pomset $P \in \mathcal{P}_{PO}(p)$, if $L \in f(P)$, then L is TSO-consistent with P.

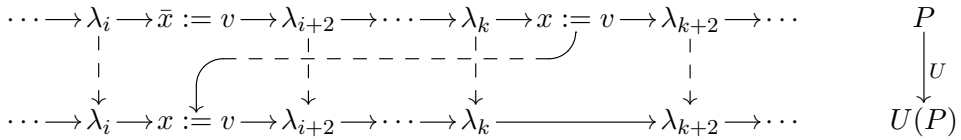
We will construct such an f and show that it is sound in this subsection, and we will show that it is complete in the next subsection.

Lemma 4.1 *For every program p and pomset $P \in \mathcal{P}_{TSO}(p)$, there exists an order*

isomorphism $\omega : P{\restriction_{\mathcal{A}_b}} \to P{\restriction_{\mathcal{A}_w}}$ *such that if* $\omega(\bar{x} := v) = (y := w)$*, then* $y = x$ *and* $w = v$*, and such that for all* $b \in P{\restriction_{\mathcal{A}_b}}$*, we have* $b <_P \omega(b)$*.*

For programs, this means that global writes appear after the corresponding write to the buffer, that global writes occur in the same order as the writes to the buffer, and that all writes to the buffer give rise to global writes. We call a TSO pomset for which such an ω exists *well-balanced*. Given a program p, all TSO pomsets in $\mathcal{P}_{TSO}(p)$ are well-balanced, and finite $P \in \mathcal{P}_{TSO}(p)$ have a unique such ω. We assume in the rest of this section that our TSO pomsets are well-balanced.

Consider the function $U : \mathrm{Pom}(\mathcal{A}_{TSO}) \to \mathrm{Pom}(\mathcal{A}_{PO})$ that takes each TSO pomset to its underlying program order. It does so by first deleting all global write actions, and then relabelling all buffer write actions $\bar{x} := v$ by corresponding global write actions $x := v$. We identify all reads in P with the corresponding reads in $U(P)$ and $(x := v) = \omega(\bar{x} := v)$ with the write $x := v$ below $\bar{x} := v$. We can imagine U and the identifications as being given in the following diagram, where dashed arrows indicate identifications, solid arrows indicate the pomset orders, and the λ_j are arbitrary read actions:



By observing that the PO pomset clauses are essentially special cases of the TSO pomset clauses, we have that for all programs p, $U(\mathcal{P}_{TSO}(p)) \subseteq \mathcal{P}_{PO}(p)$. This inclusion is actually an equality, because given any program order P for p, we can construct a TSO pomset P' such that $U(P') = P$ by immediately flushing the buffer with split after every write, i.e., by replacing all occurrences of $x := v$ in P with $\bar{x} := v \to x := v$ to get P'.

Let the set $\mathcal{T}(P)$ of TSO-consistent total orders of $P \in \mathrm{Pom}(\mathcal{A}_{TSO})$ be given by

$$\mathcal{T}(P) = \bigcup_{P' \in U^{-1}(P)} \{\Lambda{\restriction_{\mathcal{A}_{PO}}} \mid \Lambda \in \mathrm{Lin}(P') \wedge [\![P']\!]_\Lambda \neq \varnothing\}.$$

Informally, $\mathcal{T}(P)$ captures the linearisations of TSO pomsets in $U^{-1}(P)$ that give rise to TSO executions of pomsets.

Theorem 4.2 *The function* $\mathcal{T}$ *is sound.*

Proof. Let p be an arbitrary program, and let $P' \in \mathcal{P}_{PO}(p)$, $P \in U^{-1}(P')$, and $\Lambda \in \mathrm{Lin}(P)$ be arbitrary such that there exists a $(\sigma, \tau) \in [\![P]\!]_\Lambda$. Because P is well-balanced, we can assume without loss of generality that $\zeta(\sigma)$. Let $L = \Lambda{\restriction_{\mathcal{A}_{PO}}}$. It is straightforward but tedious to check the six axioms to show that L is TSO-consistent for $U(P) = P'$ from σ. We show the proof for (S) as an example.

Axiom (S). Lemma 4.1 implies there exists an order isomorphism between $P{\restriction_{\mathcal{A}_w}}$ and $U(P){\restriction_{\mathcal{A}_w}}$, so $w <_P w'$ implies $w <_{U(P)} w'$. Linearisation preserves order, so $w <_{U(P)} w'$ implies $w <_\Lambda w'$ as desired. □

4.2 Completeness

We call a function $f : \mathrm{Pom}(\mathcal{A}_{PO}) \to \wp(\mathcal{A}_{PO}\ \mathsf{list})$ **complete** when for every program p and finite pomset $P \in \mathcal{P}_{PO}(p)$, if L is TSO-consistent with P, then $L \in f(P)$.

Our goal is to show that $\mathcal{T}$ is complete. To do so, we first construct a pomset $s(P, L) \in U^{-1}(P)$ for any $P \in \mathrm{Pom}(\mathcal{A}_{PO})$ and total order L that is TSO-consistent with P. Let the underlying set $s(P, L)$ be given by $\{0\} \times P \cup \{1\} \times P{\restriction_{\mathcal{A}_w}}$. Let $\Phi_{s(P,L)}$ be given by $\Phi_{s(P,L)}((i, p)) = (\bar{x} := v)$ if both $i = 0$ and $\Phi_P(p) = (x := v)$, and $\Phi_{s(P,L)}((i, p)) = \Phi_P(p)$ otherwise. Intuitively, $<_{s(P,L)}$ merges the global writes into the program order in the places specified by L. Let $<_{s(P,L)}$ be the least strict partial order generated by the following collection of inequalities: (i) $(0, p) <_{s(P,L)} (0, p')$ if $p <_P p'$; (ii) $(1, w) <_{s(P,L)} (1, w')$ if $w \perp_P w'$ and $w <_L w'$; (iii) $(0, w) <_{s(P,L)} (1, w)$ if $w \in P{\restriction_{\mathcal{A}_w}}$; (iv) $(0, p) <_{s(P,L)} (1, w)$ if $p \perp_P w$ and $p <_L w$; and (v) $(1, w) <_{s(P,L)} (0, p)$ if $w \perp_P p$ and $w <_L p$.

The proof of completeness can be broken down into three lemmas as follows:

Lemma 4.3 *If p is a program, $P \in \mathcal{P}_{PO}(p)$, and L is TSO-consistent with P, then $s(P, L) \in \mathcal{P}_{TSO}(p)$ and $U(s(P, L)) = P$.*

Lemma 4.4 *If $P \in \mathrm{Pom}(\mathcal{A}_{PO})$ and L is TSO-consistent with P, there exists a $\Lambda \in \mathrm{Lin}(s(P, L))$ such that $\Lambda{\restriction_{\mathcal{A}_{PO}}} = L$.*

Lemma 4.5 *If p is a program, $P \in \mathcal{P}_{TSO}(p)$, $L_0 \in \mathsf{Ls}$, L is TSO-consistent with $U(P) \parallel L_0$, $\Lambda \in \mathrm{Lin}(P \parallel L_0)$, and $\Lambda{\restriction_{\mathcal{A}_{PO}}} = L$, then $[\![P]\!]_\Lambda \neq \varnothing$.*

The presence of L_0 in the statement of Lemma 4.5 lets us use an induction hypothesis in the case where P is a parallel composition of pomsets, because when we want to apply the induction hypothesis to one of the pomsets, we need to be able to reference the global writes in the other pomset.

Theorem 4.6 *The function $\mathcal{T}$ is complete.*

Proof. Given a program p, a $P \in \mathcal{P}_{PO}(P)$ and a L that is TSO-consistent with P, we have by Lemma 4.3 that $s(P, L) \in U^{-1}(P)$. By Lemma 4.4, we have a $\Lambda \in \mathrm{Lin}(s(P, L))$ such that $\Lambda{\restriction_{\mathcal{A}_{PO}}} = L$. By Lemma 4.5 with $L_0 = [\,]$, $[\![s(P, L)]\!]_\Lambda \neq \varnothing$. So $L \in \mathcal{T}(P)$, and we conclude completeness. □

5 Related Work

Other approaches to semantics for weak memory models mostly use execution graphs and operational semantics. Execution graphs [1,2] serve to describe the executional behaviour of an entire program, an inherently non-modular approach. We see our denotational framework as offering an alternative basis for program analysis, compositional and modular by design. Boudol and Petri [3] gave an operational semantics for TSO. Jagadeesan et al. [8] adapted a fully abstract, trace-based semantics by Brookes [4] to give a fully abstract denotational semantics for TSO.

Pratt [11] was the first to generalise from traces to pomsets in the study of concurrency. He introduced the parallel composition operations we presented in Section 3 and he used pomsets in the study of concurrent processes. Building on Pratt's work, Brookes [5,6,7] introduced a pomset framework to study weak memory. This framework used Pratt's parallel composition operator, and its sequential composition is a variant of Pratt's concatenation operation. The TSO semantics given above builds on Brookes's work. The key technical differences involve adapting pomset semantics to incorporate state equipped with abstract buffers, with careful

accounting to deal properly with order relaxations allowed by TSO. Our formal axiomatisation of SPARC TSO, including full treatment of forks and joins, is a crucial part of the set-up, allowing us to be precise about the relationship between our abstract denotational semantics and the more concrete and informal characterisation of TSO that appears in the manual.

6 Conclusion

Our denotational semantics accurately captures the behaviours of SPARC TSO, and its compositionality enables us reason modularly about programs. The main strength of the pomset approach is its conceptual simplicity. Unlike trace semantics, which include irrelevant orderings of actions, our pomset semantics specifies only relevant orderings of actions. The simplicity of the pomset approach also makes it readily adaptable to other memory models. For example, to capture SPARC PSO, which relaxes TSO to provide only per-location global orders on writes, we conjecture that it is sufficient to replace the single buffer parameter L in the semantic clauses $\mathcal{B}_L$ of Section 3.3 to instead use families $\{L_x\}_{x \in \mathrm{Loc}}$ of buffers, and to then modify the buffer flushing clauses in the obvious manner and to handle store barrier "`stbar`" commands. To capture other memory models, it should be sufficient to specify the correct set of actions, and to modify the semantic clauses generating pomsets and footsteps accordingly. As a result, we believe pomset semantics provide fertile ground for future research in semantics for weak memory models.

References

[1] Batty, M., S. Owens, S. Sarkar, P. Sewell and T. Weber, *Mathematizing C++ Concurrency*, SIGPLAN Not. **46** (2011), pp. 55–66.

[2] Boehm, H.-J. and S. V. Adve, *Foundations of the c++ concurrency memory model*, in: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08 (2008), pp. 68–78.

[3] Boudol, G. and G. Petri, *Relaxed memory models: An operational approach*, in: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09 (2009), pp. 392–403.

[4] Brookes, S., *Full abstraction for a shared variable parallel language*, Information and Computation **127** (1996), pp. 145–163.

[5] Brookes, S., *Partial order semantics and weak memory* (2015), Invited talk, Domains XII, Boole Symposium, University of Cork.

[6] Brookes, S., *A denotational semantics for weak memory concurrency* (2016), Invited talk, Mathematical Foundations of Computer Science.

[7] Brookes, S., *A denotational semantics for weak memory concurrency* (2016), Midlands Graduate School in the Foundations of Computing Science.
URL http://www.cs.bham.ac.uk/~pbl/mgs2016/brookesslides.pdf

[8] Jagadeesan, R., G. Petri and J. Riely, *Brookes is relaxed, almost!*, in: *Proceedings of the 15th International Conference on Foundations of Software Science and Computational Structures*, FOSSACS'12 (2012), pp. 180–194.

[9] Lamport, L., *Time, clocks, and the ordering of events in a distributed system*, Commun. ACM **21** (1978), pp. 558–565.

[10] Lamport, L., *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Trans. Comput. **28** (1979), pp. 690–691.

[11] Pratt, V., *Modeling concurrency with partial orders*, Int. J. Parallel Program. **15** (1986), pp. 33–71.

[12] SPARC Insternational Inc., "The SPARC Architecture Manual," Menlo Park, CA, USA (1992), version 8, revision SAV080SI9308.