Quarantining Weakness

Compositional Reasoning Under Relaxed Memory Models Extended version (Draft of 2013/03/10)*

Radha Jagadeesan[1], Gustavo Petri[2], Corin Pitcher[1], and James Riely[1]

[1] DePaul University
[2] Purdue University

1 Introduction

In sequential computing, every method of an object can be described in isolation via preconditions and postconditions. However, reasoning in a concurrent setting requires a characterization of all possible interactions across method invocations. Herlihy and Wing [1990]'s notion of linearizability simplifies such reasoning by intuitively ensuring that each method invocation "takes effect" between its invocation and response events.

This approach had two basic shortcomings. Firstly, in Herlihy and Wing's definition of linearizability, the interfaces are not expressive enough to codify external calls emanating from the component. Thus, objects are closed and passive.

Secondly, the definitions are for a memory model with a global total order on memory operations, thus satisfying *sequential consistency* (SC). SC is not realized by all architectures or runtime systems [Adve and Gharachorloo 1996; Adve and Boehm 2010], motivating models of relaxed memory in hardware, such as TSO [Sewell et al. 2010], PSO [SPARC, Inc. 1994], Power [Sarkar et al. 2011], and runtime systems, such as Java [Manson et al. 2005; Sevcík 2008] and C++ [Boehm and Adve 2008; Batty et al. 2011]. This has motivated recent definitions of linearizability specific to the TSO [Burckhardt et al. 2012; Gotsman et al. 2012] and C11 [Batty et al. 2013] memory models.

We propose new definitions to address both of these limitations. Our methodology aims to keep the interfaces free of the intricacies of particular relaxed memory models. Our approach has the following characteristics.

(1) We model calls to component functions process-algebraically. This allows us to treat callbacks and to give a symmetric definition of composition between clients and libraries. Thus, our definitions encompass active components (that can evolve autonomously even without input from the environment) and open components (that invoke methods on components provided by the environment) and environment assumptions (pre/postconditions and the permitted sequences of method calls to a component).

(2) Our definitions are not specific to a particular memory model. Rather, we identify the criteria that a relaxed memory model needs to satisfy in order to fit into our framework: the examples that satisfy our criteria include SC, TSO, PSO and a variant of the Java Memory Model (JMM).

* Research supported by NSF 0916741.

We establish an abstraction theorem: a component can safely be substituted for its interface in a non-interfering program. Moreover, for special classes of programs, we simplify the reasoning further by quarantining the effects of relaxed memory, allowing programmers to program to sequential interfaces, even when the code has data races. Recall the definition of data race free (DRF) models: Informally, a program is DRF if no SC execution of the program leads to a state in which a write happens concurrently with another operation on the same location. A *DRF model* requires that the programmer's view of relaxed computation coincides with *SC* computations for programs that are DRF. TSO, PSO and the JMM are all DRF models. We establish the following.

(1) If a stateful component is DRF and the underlying memory model satisfies the DRF requirement, our notion of linearizability usually coincides with that of Herlihy and Wing, so classical techniques to verify linearizability can be used directly. Thus, in many cases, our definitions permit the use of standard proof techniques.

(2) If a client is DRF, and the underlying memory model satisfies the DRF requirement, the client can ignore all memory model subtleties when using a library that is linearizable as per our definitions, even if the library itself is *racy*. More precisely, it is sound for the client to reason solely with the sequential interface of the component, as in [Herlihy and Wing 1990].

Rest of the paper. In Section 2, we describe background material on linearizability in order to clarify the difficulties caused by relaxed memory. We discuss related work in Section 3 and develop our semantic framework in Sections 4–6. We define linearizability in Section 7 and provide several examples. In Section 8, we turn to techniques for establishing linearizability under relaxed memory using techniques developed for sequential consistency. In Section 9–10, we establish the basic properties of linearizability. Many definitions and all proofs are elided in this extended abstract.

2 Background: linearizability

To illustrate the issues that arise when reasoning compositionally, we describe the specification and implementation of a lock and a one-place buffer implemented using the lock.

Specifying the lock. To begin, we give the specification of a lock using an regular expression. We use regular expressions informally; the actual specifications are sets of traces. Let s and t be thread identifiers. Because we are interested in overlapping executions, we separate call and return into separate actions: $\langle \mathsf{s?call\ f\ u} \rangle$ represents a call by s to function f with argument u, and $\langle \mathsf{s!ret\ f\ v} \rangle$ represents the corresponding return with result v. (The $\mathsf{?}$ and $\mathsf{!}$ indicate that these are calls *in* to the lock and returns *out*; we shall see the symmetric case shortly.)

$$(\ (\ \langle \mathsf{s?call\ rl} \rangle \langle \mathsf{s!ret\ rl} \rangle\ )^{+} \langle \mathsf{t?call\ aq} \rangle \langle \mathsf{t!ret\ aq} \rangle\ )^{*}$$

According to the specification, the lock is initially in its "acquired" state. Only after one or more calls to the "release" method rl, can the lock be "reacquired" using aq. This regular expression is not meant to refer to specific concrete thread names s and t. Rather, it is meant to convey the idea that calls and returns have matching thread names.

Let Ψ_{lock} be the prefix-closed set of traces that satisfy this regular expression. This is a “sequential” specification of the lock, in that no two function calls overlap.

We now turn to implementation of the lock. Here we use an *atomic* variable, which we define to be similar to volatile variables in Java, with an additional compare-and-set (cas): $\mathtt{w.cas(u,v)}$ returns false if $\mathsf{w} \neq \mathsf{u}$, otherwise it returns true and sets w to v.

$$\begin{array}{l}\mathsf{atomic\ w{=}1;}\\ \mathsf{fun\ rl()\ \{\ w{=}0;\ \}}\\ \mathsf{fun\ aq()\ \{\ do\ skip\ until\ w.cas(0,1);\ \}}\end{array}\qquad(\mathsf{Lock})$$

Initially, calls to aq will spin, only returning after another thread calls rl. In the vocabulary of [Lamport 1979], a call to rl *happens-before* the return from aq. The happens-before relation allows a partial order to be recovered from the total order prescribed by a trace: actions of a single thread are ordered sequentially, but actions of different threads are unordered. Inter-thread order requires *synchronization*, which is we implement using atomic variables, such as w.

Every write to an atomic variable happens-before every subsequent read of the same atomic. An unsuccessful cas acts like a read, whereas a successful cas acts like both a read and a write. In traces, atomics produce three types of action: writes produce $\langle \mathsf{s}\ \underline{\mathsf{wr}}\ \mathsf{w}\rangle$ actions, reads and unsuccessful cas produce $\langle \mathsf{s}\ \underline{\mathsf{rd}}\ \mathsf{w}\rangle$ actions, and successful cas produce $\langle \mathsf{s}\ \underline{\mathsf{cas}}\ \mathsf{w}\rangle$ actions. The happens-before relation orders every $\langle \underline{\mathsf{wr}}\ \mathsf{w}\rangle$ and $\langle \underline{\mathsf{cas}}\ \mathsf{w}\rangle$ with every subsequent $\langle \underline{\mathsf{rd}}\ \mathsf{w}\rangle$ and $\langle \underline{\mathsf{cas}}\ \mathsf{w}\rangle$. These relations are based on the identity, w, of the atomic.

Let Φ_{lock} be the set of implementation traces generated by the implementation code above. These include traces of the form

$$(\ (\ \langle \mathsf{s?call\ rl}\rangle\langle \mathsf{s}\ \underline{\mathsf{wr}}\ \mathsf{w}\rangle\langle \mathsf{s!ret\ rl}\rangle\ )^{+}\langle \mathsf{t?call\ aq}\rangle\langle \mathsf{t}\ \underline{\mathsf{cas}}\ \mathsf{w}\rangle\langle \mathsf{t!ret\ aq}\rangle\ \ )^{*}.$$

(This regular expression is not exhaustive, since the implementation also generates overlapping function calls; however, it is sufficient for the discussion at hand.)

Herlihy and Wing [1990] propose linearizability as a way to relate the implementation of a concurrent component to its specification. An implementation is *linearizable* if for every trace of the implementation, there exists a trace in the specification such that (1) each thread makes the same method invocations in the same order, and (2) the order of non-overlapping invocations is preserved. We write $\Phi_{\mathsf{lock}} \vDash \Psi_{\mathsf{lock}}$ to indicate that Φ_{lock} is a valid implementation of Ψ_{lock} in this sense.

Specifying the buffer. We now give the specification and implementation of a one-place buffer using Lock. The buffer's sequential specification can be given as follows.

$$(\ \langle \mathsf{s?call\ put\ v}\rangle\langle \mathsf{s!ret\ put}\rangle\ \langle \mathsf{t?call\ get}\rangle\langle \mathsf{t!ret\ get\ v}\rangle\ )^{*}$$

As before, let Ψ_{buf} be the prefix-closed set of traces that satisfy this regular expression.

The implementation of the one place buffer uses two locks. We use subscripts to distinguish them. One of the locks has interface $\mathsf{aq}_{\mathsf{empty}}/\mathsf{rl}_{\mathsf{empty}}$ (initially “released”, with w==0) and the other has interface $\mathsf{aq}_{\mathsf{full}}/\mathsf{rl}_{\mathsf{full}}$ (initially “acquired” with w==1). Thus, the buffer is initially empty. (Note that two “instances of a class” are represented here as two separate components.)

$$\begin{array}{l}\mathsf{var\ x{=}0;}\\ \mathsf{fun\ put(z)\ \{\ aq_{empty}();\ x{=}z;\ rl_{full}();\ \}}\\ \mathsf{fun\ get()\ \ \{\ aq_{full}();\ let\ z{=}x;\ rl_{empty}();\ return\ z;\ \}}\end{array}\qquad(\mathsf{Buffer})$$

Let $\mathit{\Phi}_{\mathsf{buf}}$ be the set of traces derived from this implementation, including traces such as

$$\begin{array}{l}
(\ \langle \mathsf{s?call\ put\ v}\rangle\\
\quad\langle \mathsf{s!call\ aq_{empty}}\rangle\langle \mathsf{s?ret\ aq_{empty}}\rangle\langle \mathsf{s\ wr\ x\ v}\rangle\langle \mathsf{s!call\ rl_{full}}\rangle\langle \mathsf{s?ret\ rl_{full}}\rangle\\
\langle \mathsf{s!ret\ put}\rangle\langle \mathsf{t?call\ get}\rangle\\
\quad\langle \mathsf{t!call\ aq_{full}}\rangle\langle \mathsf{t?ret\ aq_{full}}\rangle\langle \mathsf{t\ rd\ x\ v}\rangle\langle \mathsf{t!call\ rl_{empty}}\rangle\langle \mathsf{t?ret\ rl_{empty}}\rangle\\
\langle \mathsf{t!ret\ get\ v}\rangle\ )^*.
\end{array}$$

This trace contains actions of the form $\langle \mathsf{s!call\ f\ u}\rangle$ which represent a call out to another component; likewise, $\langle \mathsf{s?ret\ f\ v}\rangle$ represents the corresponding return. In this case, the implementation is using services provided by other components.

We would like to be able to verify the correctness of Buffer using the sequential specification of Lock. That is, conclude $\mathit{\Phi}_{\mathsf{buf}} \otimes \mathit{\Phi}_{\mathsf{lock}} \vDash \mathit{\Psi}_{\mathsf{buf}}$ from $\mathit{\Phi}_{\mathsf{buf}} \otimes \mathit{\Psi}_{\mathsf{lock}} \vDash \mathit{\Psi}_{\mathsf{buf}}$, where $\otimes$ is a suitable notion of composition. Herlihy and Wing validate this approach under SC semantics. Burckhardt, Gotsman, Musuvathi, and Yang [2012] show that Herlihy and Wing's results fail for relaxed memory models and adapt them to TSO. Here we provide a different solution to that problem.

Traditional linearizability fails here, because it is impossible to establish the premise $\mathit{\Phi}_{\mathsf{buf}} \otimes \mathit{\Psi}_{\mathsf{lock}} \vDash \mathit{\Psi}_{\mathsf{buf}}$. To see why, observe that any reasonable definition $\mathit{\Phi}_{\mathsf{buf}} \otimes \mathit{\Psi}_{\mathsf{lock}}$ admits the following trace under relaxed memory. (For brevity, the calls to the locks are shown as elipses.)

$$\begin{array}{l}
\langle \mathsf{s?call\ put\ 1}\rangle\cdots\langle \mathsf{s\ wr\ x\ 1}\rangle\cdots\langle \mathsf{s!ret\ put}\rangle\langle \mathsf{t?call\ get}\rangle\cdots\langle \mathsf{t\ rd\ x\ 1}\rangle\cdots\langle \mathsf{t!ret\ get\ 1}\rangle\\
\quad\langle \mathsf{r?call\ put\ 2}\rangle\cdots\langle \mathsf{r\ wr\ x\ 2}\rangle\cdots\langle \mathsf{r!ret\ put}\rangle\langle \mathsf{t?call\ get}\rangle\cdots\langle \mathsf{t\ rd\ x\ 1}\rangle\cdots\langle \mathsf{t!ret\ get\ 1}\rangle
\end{array} \tag{†}$$

The final call to get returns a stale value. The race on variable x is not resolved, and thus the earlier write on x remains visible.

Of course, if one looks at the specification of Lock, the problem is immediately apparent: it's too weak! In relaxed models, data structures have memory effects which are not captured by their functional interface. Indeed, the documentation in APIs such as java.util.concurrent [Sun Microsystems 2004] pays significant attention to exactly this fact. These APIs detail the happens-before behavior of the methods using happens-before edges that go from the beginning of one method activation to the end of another (or a set of others); that is, from call to return.

We allow happens-before to be captured in specifications by introducing names, a, on actions. Each $\langle ?\mathsf{call}\rangle$ gets a unique name, and each $\langle !\mathsf{ret}\rangle$ gets a set of names. The interpretation is that $\langle s?\mathsf{call}\ f\ \vec{u}\ a\rangle$ happens-before $\langle t!\mathsf{ret}\ f\ \vec{v}\ A\rangle$ if $a \in A$.

With this addition, Lock can be specified as follows

$$(\ (\ \langle \mathsf{r?call\ rl}\rangle\langle \mathsf{r!ret\ rl}\rangle\ )^*\ \langle \mathsf{s?call\ rl\ a}\rangle\langle \mathsf{s!ret\ rl}\rangle\langle \mathsf{t?call\ aq}\rangle\langle \mathsf{t!ret\ aq\ \{a\}}\rangle\ )^*$$

This specification is now strong enough to deduce happens-before edges from each put to get that it enables, and vice versa. Thus, in trace (†) above, the write to x in the first put is no longer visible to the second get. More generally, we are able to establish $\mathit{\Phi}_{\mathsf{buf}} \otimes \mathit{\Psi}_{\mathsf{lock}} \vDash \mathit{\Psi}_{\mathsf{buf}}$.

3 Related work

We discuss the most closely related papers here, referring to others in context. Herlihy and Wing [1990] defined linearizability. From a client perspective, the set of linearizations of a linearizable object is an operational refinement of the object [Filipovic, O'Hearn, Rinetzky, and Yang 2010], i.e. the client is unable to distinguish the implementation from the specification. Thus, a client of a linearizable object can take an atomic view of method invocations. The verification method for object linearizability relies on finding linearization points for methods. For each function call, the linearization point is the moment at which the function appears to execute atomically. Composition of non-interfering objects preserves linearizability. Gotsman and Yang [2012] mitigate the stricture of interference-freedom in this framework using ownership ideas.

The papers cited above make a sharp distinction between clients and libraries; clients are permitted to make method invocations and libraries accept method invocations. Thus, they are unable to describe the interface of open components such as a thread pool that relies on an external bounded buffer library. In contrast, our enhanced notion of interfaces is able to describe such components. In terms of implementations, our library can both make and receive method invocations in external interactions, in addition to also being able to invoke internal library methods. Indeed, we stop short of adding full objects, as suggested by Filipovic, O'Hearn, Rinetzky, and Yang [2010], only to avoid cluttering the presentation with heavy syntactic machinery.

The definition of linearizability relies on an SC view of shared memory. Batty, Dodds, and Gotsman [2013] address linearizability in the context of the C/C++ memory models. When specialized to SC, their definition of linearizability is stricter than that of Herlihy and Wing. In contrast, when specialized to SC, our definitions are *not* stricter.

In TSO, an update to a variable might be buffered and may not be seen by a reader in a different thread until the update is committed to the main memory. Burckhardt et al. [2012] address linearizability for the TSO memory model. In contrast both to Herlihy and Wing and to our definitions, their paper incorporates two extra actions for each method invocation in the sequential specification of an object: one to record when buffer updates made by the client are seen by the library, and the other to record when the updates made by library are committed to main memory. In our work we maintain the atomicity of methods of Herlihy and Wing by only associating call and return actions with each method invocation.

More generally, our methodology keeps the interface of a component free of the intricacies of the particular relaxed memory model under consideration. In this paper, we are thus able to address SC, TSO, PSO and a JMM variant. In particular, our analysis of TSO is subtle enough to address all the examples of Burckhardt et al. [2012], even though, from a purely formal TSO perspective, there is clearly greater expressiveness in their definition. Consequently, any data race free client can work precisely against a SC interface in our setting, whereas Gotsman, Musuvathi, and Yang [2012] explore the conditions on compilation necessary to validate the use of SC interfaces under TSO.

4 Traces

The semantics of a component is given by a set of *traces*, defined below. We build the syntax from the following disjoint sets. Let $u, v \in \mathbb{Z}$ range over values, $a, b \in \mathit{Act}$ over action names, $A, B \subseteq \mathit{Act}$ over sets of action names, $f, g \in \mathit{Fun}$ over function names, $F \subseteq \mathit{Fun}$ over sets of function names, $s, t \in \mathit{Thrd}$ over thread names (including the reserved thread names “tinit” and “tcom”) and $S, T \subseteq \mathit{Thrd}$ over sets thread names. Let $\eta \in \mathit{Fun} \uplus \mathit{Thrd}$ range over names, which include both function and thread names, and H, G over sets of names.

Traces are strings of *actions*. These are divided into *communication actions*, described below, and *memory actions*, described in Section 5. For now, let *Mem* be the set of all memory actions.

$$\begin{aligned}\alpha, \gamma ::= {} & \langle s\,\mathtt{!}\mathsf{call}\, f\, \vec{u}\, a\, A\rangle \mid \langle s\mathtt{?}\mathsf{call}\, f\, \vec{u}\, a\, A\rangle \mid \langle s\,\mathtt{.}\,\mathsf{call}\, f\, \vec{u}\, a\, A\rangle \\ & \mid \langle s\mathtt{?}\mathsf{ret}\ \ f\, \vec{u}\, a\, A\rangle \mid \langle s\,\mathtt{!}\mathsf{ret}\ \ f\, \vec{u}\, a\, A\rangle \mid \langle s\,\mathtt{.}\,\mathsf{ret}\ \ f\, \vec{u}\, a\, A\rangle \mid \cdots\end{aligned}$$

Communication actions include seven components, discussed below: thread identifier s, polarity in $\{\mathtt{!}, \mathtt{?}, \mathtt{.}\}$, action type in $\{\mathsf{call}, \mathsf{ret}\}$, function name f, vector of arguments or return values $\vec{u}$, definition a, and use set A.

We typically elide the uninteresting parts of an action; missing parts are existentially quantified. For example, we write $\langle \mathtt{!}\mathsf{call}\, f\, \vec{u}\, a\, A\rangle$ to abbreviate $(\exists s)\langle s\,\mathtt{!}\mathsf{call}\, f\, \vec{u}\, a\, A\rangle$, and similarly for other abbreviations such as $\langle s\,\mathtt{!}\mathsf{call}\rangle$, $\langle \mathsf{call}\, f\rangle$, $\langle s\,\mathtt{!}\, f\rangle$ and $\langle \mathtt{!}\rangle$.

The thread identifier identifies the thread that performed the action.

As in Jeffrey and Rathke [2005], call and return actions include a *polarity*. Actions containing a “?” are *input*; those containing “!” are *output*; actions containing “.” are *internal*, as are memory actions. Input actions are offered by *quiescent* threads, whereas all others are initiated by *active* threads. Two actions are *complementary* if one is an input, the other an output and they are identical when action names and “?” and “!” are ignored. If $\alpha \in \{\langle \mathtt{!}\rangle, \langle \mathtt{?}\rangle\}$, we say α *is I/O*.

Actions $\langle \mathtt{!}\mathsf{call}\, f\rangle$ and $\langle \mathtt{?}\mathsf{ret}\, f\rangle$ occur in the traces of components that do *not* define f; whereas $\langle \mathtt{?}\mathsf{call}\, f\rangle$, $\langle \mathtt{!}\mathsf{ret}\, f\rangle$, $\langle \mathtt{.}\,\mathsf{call}\, f\rangle$ and $\langle \mathtt{.}\,\mathsf{ret}\, f\rangle$ occur those that *do*. Action $\langle \mathtt{?}\mathsf{call}\rangle$ represents a call from outside the component, whereas $\langle \mathtt{.}\,\mathsf{call}\rangle$ represents a call from the component to itself. Thus, input and output actions cause a shift across the boundary of the component for that thread, whereas the internal actions do not.

Call actions include the vector of actual parameters. Return actions include a vector of return values. Several examples require multiple return values. An obvious generalization would be to support first-class tuples, but this would complicate the presentation.

The action names decorating actions are used to specify ordering properties (Section 5). Each action *defines* a unique action name a. For the purposes of defining traces and trace composition, these names are mere decorations: we identify traces up to the renaming of action names. In $\langle \mathtt{?}\, A\rangle$, the set A contains names defined by “!” actions and represents an order relied upon by the component. In $\langle \mathtt{!}\ A\rangle$, the set A contains names defined by “?” actions and represents an order guaranteed the component. In $\langle \mathtt{.}\ A\rangle$, the set A contains names defined by “.” actions and represent the interaction of two components, one which relies upon A and one which guarantees it. In operationally generated traces, A is empty for any $\langle \mathtt{!}\ A\rangle$ or $\langle \mathtt{.}\ A\rangle$; these sets or nonempty when working with specification interfaces.

Definition 4.1 (Trace). For any given thread, define a *single-threaded balanced trace* to be one generated by the following grammar.

$$\mathrm{B} ::= \mathrm{A} \mid \mathrm{Q} \qquad \text{(Single-threaded balanced trace)}$$

$$
\begin{array}{llr}
\mathrm{A} ::= \langle .\,\mathsf{call}\, f\rangle\ \mathrm{A}\ \langle .\,\mathsf{ret}\, f\rangle \mid \mathrm{A}\,\mathrm{A} \mid \varepsilon & & \text{(Active trace)} \\
\quad \mid \langle !\,\mathsf{call}\, f\rangle\ \mathrm{Q}\ \langle ?\mathsf{ret}\, f\rangle \mid \mathrm{M} & & \\
\mathrm{Q} ::= \langle ?\mathsf{call}\, f\rangle\ \mathrm{A}\ \langle !\,\mathsf{ret}\, f\rangle \mid \mathrm{Q}\,\mathrm{Q} \mid \varepsilon & & \text{(Quiescent trace)} \\
\mathrm{M} \in \mathit{Mem} & & \text{(Memory action)}
\end{array}
$$

(We elide uninteresting metavariables within actions. Because they are single-threaded, all actions have the same thread name.)

A *balanced trace* is any interleaving of single-threaded balanced traces with distinct thread names. A *trace* is a trace of actions that is well-formed and is also a prefix of a balanced trace. Let σ, ρ, π range over traces. □

We give an inductive characterization of traces in Appendix A.

We expose, and nest, calls and returns as with VPLs [Alur and Madhusudan 2009]. As seen from the grammar, prefixes of single-threaded balanced trace are divided into two *polarities*: quiescent and active. By convention, ε is quiescent. For all other traces, the polarity is determined by the first action of the trace: if it is $\langle ?\mathsf{call}\rangle$, then the trace is quiescent; otherwise the trace is active.

Traces have three forms of bracketing, indexed by thread: call/return, input/output and output/input. (Internal actions provide no interesting bracketing other than call/return.) In the trace $\langle s?\mathsf{call}\ f\rangle\langle s!\mathsf{call}\ g\rangle\langle s?\mathsf{ret}\ g\rangle\langle s!\mathsf{ret}\ f\rangle$, the call/return matches are $\langle s?\mathsf{call}\ f\rangle/\langle s!\mathsf{ret}\ f\rangle$ and $\langle s!\mathsf{call}\ g\rangle/\langle s?\mathsf{ret}\ g\rangle$; the input/output matches are $\langle s?\mathsf{call}\ f\rangle/\langle s!\mathsf{call}\ g\rangle$ and $\langle s?\mathsf{ret}\ g\rangle/\langle s!\mathsf{ret}\ f\rangle$; the output/input match is $\langle s!\mathsf{call}\ g\rangle/\langle s?\mathsf{ret}\ g\rangle$.

Here are some further examples: $\langle s!\rangle\langle s?\rangle$ is a trace, but $\langle s!\rangle\langle s!\rangle$ is not. $\langle s!\rangle\langle t?\rangle\langle s?\rangle$ is a trace, but $\langle s!\rangle\langle s?\rangle\langle s?\rangle$ is not. $\langle s?\rangle\langle s.\rangle$ is a trace, but $\langle s!\rangle\langle s.\rangle$ is not.

Definition 4.2. Define the function *thrd* to return the thread name occurring inside an action and *thrds* to return the set of threads in a sequence of actions. Similarly, define the partial functions *fun* and *funs* to return the function name. For example, if $\alpha = \langle s!\mathsf{call}\ f\ \vec{u}\ a\ A\rangle$, then $\mathit{thrd}(\alpha) = s$ and $\mathit{fun}(\alpha) = f$.

Given a trace σ, define the *thread projection* $\sigma|_s$ of that trace, which includes only the actions attributed to thread s; this is always a prefix of a single-threaded balanced trace. Define the following functions over traces.

$$
\begin{aligned}
\mathit{intern}(\alpha_1\cdots\alpha_n) &\triangleq \{f \mid \exists i.\ \alpha_i = \langle ?\mathsf{call}\ f\rangle \text{ or } \alpha_i = \langle .\,\mathsf{call}\ f\rangle\} \\
&\quad \cup \{s \mid (\sigma|_s) \neq \varepsilon \text{ is an active trace}\} \setminus \{\mathsf{tinit}, \mathsf{tcom}\} \\
\mathit{extern}(\alpha_1\cdots\alpha_n) &\triangleq \{f \mid \exists i.\ \alpha_i = \langle !\,\mathsf{call}\ f\rangle\} \\
&\quad \cup \{s \mid (\sigma|_s) \neq \varepsilon \text{ is an quiescent trace}\}
\end{aligned}
$$

These definitions lift to trace sets via set union. When interpreted over trace sets, *intern* identifies the functions and threads defined by the component, whereas *extern* identifies the functions and threads mentioned in a component, but not defined by it.

A trace σ is *coherent* if $\mathit{intern}(\sigma) \cap \mathit{extern}(\sigma) = \emptyset$. We assume that all traces are coherent. We also assume other well-formedness criteria, detailed in Appendix A.

A set Σ of traces is *coherent* if $\mathit{intern}(\Sigma) \cap \mathit{extern}(\Sigma) = \emptyset$. Note that this is stronger than requiring only that each individual trace be coherent. Let Φ, Ψ range over coherent sets of traces.

A trace is *sequential* if it can be extended in such a way that every $\langle s\texttt{?}\rangle$ is followed by actions exclusively by s, up to a terminating $\langle s\texttt{!}\rangle$. A trace set is *sequential* if it contains only sequential traces.

A trace set is an *interface* if it contains only I/O actions. □

5 Memory actions and memory orders

Our approach is parametric with respect to the specific memory model considered. For concreteness, we will consider four models here: seq, hb, tso and pso. To keep the formalism simple, we assume that (1) memory stores only integers, (2) atomics provide the only form of synchronization and (3) components are specified as sets of functions, variables and threads.

Let $z \in \mathit{Reg}$ range over registers (local variables), $x, y \in \mathit{DataVar}$ over data variables and $w \in \mathit{SyncVar}$ over synchronization variables. We use the general term *variable* to include data variables and synchronization variables, but not registers. Memory actions are as follows.

$$\begin{array}{rlll} \alpha, \gamma ::= \cdots & \mid \langle s\ \mathsf{wr}\ x\ u\ a\rangle & \mid \langle s\ \mathsf{rd}\ x\ u\ a\rangle & \mid \langle \mathsf{com}\ s\ x\ a\rangle \\ & \mid \langle s\ \underline{\mathsf{wr}}\ w\rangle & \mid \langle s\ \underline{\mathsf{rd}}\ w\rangle & \mid \langle s\ \underline{\mathsf{cas}}\ w\rangle \end{array}$$

For data variables, the actions record writes, reads and commits. For synchronization variables, the actions record releases, acquires and compare-and-sets. Action names (metavariable a, as before) are used to record relations between data actions. Commit actions are used by buffering models, such as tso and pso, to indicate the point at which a write is moved from the local buffer to main memory. Non-buffering models, such as seq and hb, have no commit actions.

Neither initializations nor commits are performed by the program, but by the underlying operational machinery. Initialization actions are normal writes attributed to the reserved pseudo-thread “tinit”. Commit actions are only performed by the reserved pseudo-thread “tcom”; thus we simply define $\mathit{thrd}(\langle \mathsf{com}\ s\ x\ a\rangle) = \mathsf{tcom}$. In $\langle \mathsf{com}\ s\ x\ a\rangle$, the identifiers s and x are redundant with the corresponding $\langle s\ \mathsf{wr}\ x\ u\ a\rangle$.

Synchronization variables carry memory effects whereas data variables do not. Registers are used to write programs, but are not shared between threads; thus, we do not require actions relating to registers.

The name a is *defined* in $\langle \mathsf{wr}\ a\rangle$ and *used* in $\langle \mathsf{rd}\ a\rangle$ and $\langle \mathsf{com}\ a\rangle$. We expect that every write action is committed at most once and that the redundant information in read and commit actions should match the corresponding write. In addition, initialization writes by thread “tinit” must appear at the beginning of a trace. These bookkeeping requirements are included in the notion of *well-formed* trace, formalized in Appendix A. Most of the requirements are unsurprising. We note only that well-formedness does *not* require that a read be proceeded by the matching write, since this is not true under all of the models we consider.

Example 5.1. Consider the following traces, each containing actions from three different threads (eliding initialization and commit actions).

$$\langle \mathsf{s\ wr\ x\ a}\rangle\langle \mathsf{t\ wr\ x\ b}\rangle\langle \mathsf{r\ rd\ x\ b}\rangle \qquad \text{(a)}$$

$$\langle \mathsf{t\ wr\ x\ b}\rangle\langle \mathsf{s\ wr\ x\ a}\rangle\langle \mathsf{r\ rd\ x\ b}\rangle \tag{b}$$

$$\langle \mathsf{s\ wr\ x\ a}\rangle\langle \mathsf{r\ rd\ x\ b}\rangle\langle \mathsf{t\ wr\ x\ b}\rangle \tag{c}$$

$$\langle \mathsf{s\ wr\ x\ a}\rangle\langle \mathsf{s\ wr\ y\ b}\rangle\langle \mathsf{t\ wr\ x\ c}\rangle\langle \mathsf{t\ wr\ y\ d}\rangle\langle \mathsf{r\ rd\ y\ d}\rangle\langle \mathsf{r\ rd\ x\ a}\rangle \tag{d}$$

– Under seq, reads and writes are atomic; thus, a read must be fulfilled by the previous write. Only trace (a) is allowable; the others require that a read see a stale write.
– Under tso, writes are placed in a buffer which is not visible to other threads; for any given thread, the buffered writes are committed to main memory in FIFO order, but the order between threads is nondeterministic. Thus, traces (a) and (b) are allowable.
– pso is similar to tso, except that each thread has a separate buffer for each variable. Thus, traces (a), (b) and (d) are allowable.
– Under hb, a write may be seen by a reader even before it is generated by the writer. Thus, all four executions are allowable. □

Example 5.2. Consider the following unsynchronized implementation of a one place buffer (on the left) and client (on the right).

```
var y=0                                   var x=0
fun put (z){y=z}                          thrd s {x=1;put(3);wait(4);let z′=x}
fun wait (z){do skip until y==z}          thrd t {wait(3);x=2;put(4)}
```

Ignoring initialization and commits, here is a single trace of the library and of the client, each in isolation. (The label sets decorating return actions are specification elements. Those on the library output actions are *guarantees*, whereas those on client input actions are *relies*.)

$$\begin{array}{ll}
\langle \mathsf{s?call\ put\ 3\ a}\rangle\langle \mathsf{s\ wr\ y\ 3}\rangle\langle \mathsf{s!ret}\ \emptyset\rangle & \langle \mathsf{s\ wr\ x\ 1}\rangle\langle \mathsf{s!call\ put\ 3\ a}\rangle\langle \mathsf{s?ret}\ \emptyset\rangle \\
\quad\langle \mathsf{t?call\ wait\ 3\ b}\rangle\langle \mathsf{t\ rd\ y\ 3}\rangle\langle \mathsf{t!ret}\ \{\mathsf{a}\}\rangle & \quad\langle \mathsf{t!call\ wait\ 3\ b}\rangle\langle \mathsf{t?ret}\ \{\mathsf{a}\}\rangle\langle \mathsf{t\ wr\ x\ 2}\rangle \\
\quad\quad\langle \mathsf{t?call\ put\ 4\ c}\rangle\langle \mathsf{t\ wr\ y\ 4}\rangle\langle \mathsf{t!ret}\ \emptyset\rangle & \quad\quad\langle \mathsf{t!call\ put\ 4\ c}\rangle\langle \mathsf{t?ret}\ \emptyset\rangle \\
\quad\quad\quad\langle \mathsf{s?call\ wait\ 4\ d}\rangle\langle \mathsf{s\ rd\ y\ 4}\rangle\langle \mathsf{s!ret}\ \{\mathsf{c}\}\rangle & \quad\quad\quad\langle \mathsf{s!call\ wait\ 4\ d}\rangle\langle \mathsf{s?ret}\ \{\mathsf{c}\}\rangle\langle \mathsf{s\ rd\ x\ 1}\rangle
\end{array}$$

Composing the traces, we have the following trace (on the left), which, if we elide "." actions, is equivalent to the trace on the right.

$$\begin{array}{ll}
\langle \mathsf{s\ wr\ x\ 1}\rangle\langle \mathsf{s.call\ put\ 3\ a}\rangle\langle \mathsf{s\ wr\ y\ 3}\rangle\langle \mathsf{s.ret}\ \emptyset\rangle & \langle \mathsf{s\ wr\ x\ 1}\rangle\langle \mathsf{s\ wr\ y\ 3}\rangle \\
\quad\langle \mathsf{t.call\ wait\ 3\ b}\rangle\langle \mathsf{t\ rd\ y\ 3}\rangle\langle \mathsf{t.ret}\ \{\mathsf{a}\}\rangle\langle \mathsf{t\ wr\ x\ 2}\rangle & \quad\langle \mathsf{t\ rd\ y\ 3}\rangle\langle \mathsf{t\ wr\ x\ 2}\rangle \\
\quad\quad\langle \mathsf{t.call\ put\ 4\ c}\rangle\langle \mathsf{t\ wr\ y\ 4}\rangle\langle \mathsf{t.ret}\ \emptyset\rangle & \quad\quad\langle \mathsf{t\ wr\ y\ 4}\rangle\langle \mathsf{s\ rd\ y\ 4}\rangle \\
\quad\quad\quad\langle \mathsf{s.call\ wait\ 4\ d}\rangle\langle \mathsf{s\ rd\ y\ 4}\rangle\langle \mathsf{s.ret}\ \{\mathsf{c}\}\rangle\langle \mathsf{s\ rd\ x\ 1}\rangle & \quad\quad\quad\langle \mathsf{s\ rd\ x\ 1}\rangle
\end{array}$$

Ignoring calls and returns, under what circumstances should such a trace be allowed?

On the one hand, it is clearly *not* allowed under sequential semantics, since $\langle \mathsf{s\ rd\ x\ 1}\rangle$ does not see the most recent write. On the other hand, it is clearly *allowed* under a happens-before semantics, since there is no synchronization between thread s and t.

For tso and pso, the situation is less obvious. In fact, pso will allow the trace, but tso will not. The difference is that tso enforces an ordering between $\langle \mathsf{t\ wr\ x\ 2}\rangle$ and $\langle \mathsf{t\ wr\ y\ 4}\rangle$, whereas pso does not.

Moving from the combined trace back to the trace of the library in isolation, for each memory model, we may ask "does the library implementation meets its specification?" In this case, the answer is positive for seq and tso, and negative for pso and hb.

Similarly, moving from the combined trace back to the trace of the client in isolation, for each memory model, we may ask "is the final client read valid?" For this question, the answers are reversed: valid for pso and hb, and invalid for seq and tso. □

To formalize these properties, we introduce a notion of memory ordering, which is derivable from a trace. Recall that tinit is a reserved name.

Definition 5.3. The partial function *var* is undefined for commit and nonmemory actions and otherwise returns the variable mentioned: $var(\alpha) \triangleq x$ if $\alpha \in \{\langle \mathsf{wr}\, x\rangle, \langle \mathsf{rd}\, x\rangle\}$; $var(\alpha) \triangleq w$ if $\alpha \in \{\langle \underline{\mathsf{wr}}\, w\rangle, \langle \underline{\mathsf{rd}}\, w\rangle, \langle \underline{\mathsf{cas}}\, w\rangle\}$; and $var(\alpha)$ is undefined otherwise.

From a trace $\sigma = \alpha_1 \cdots \alpha_n$, we derive several relations.

- $i <^{\sigma}_{\mathsf{rf}} j$ if $\alpha_i = \langle \mathsf{wr}\, a\rangle,\ \alpha_j = \langle \mathsf{rd}\, a\rangle$ *(reads-from relation)*
- $i <^{\sigma}_{\mathsf{cb}} j$ if $\alpha_i = \langle \mathsf{wr}\, a\rangle,\ \exists \ell < j.\ \alpha_\ell = \langle \mathsf{com}\, a\rangle$ *(committed-before relation)*
- $i <^{\sigma}_{\mathsf{ext}} j$ if $\alpha_i, \alpha_j \in \{\langle\,!\rangle, \langle ?\rangle, \langle\,.\,\rangle\}$, $\alpha_i = \langle\ a\rangle$ and $\alpha_j = \langle\ A \cup \{a\}\rangle$ *(external order)*
- $i <^{\sigma}_{\mathsf{init}} j$ if $i < j,\ thrd(\alpha_i) = \mathsf{tinit} \neq thrd(\alpha_j)$ *(init order)*
- $i <^{\sigma}_{\mathsf{thrd}} j$ if $i < j,\ thrd(\alpha_i) = thrd(\alpha_j) \notin \{\mathsf{tinit}, \mathsf{tcom}\}$ *(thread order)*
- $i <^{\sigma}_{\mathsf{var}} j$ if $i < j,\ var(\alpha_i) = var(\alpha_j)$ *(variable order)*
- $i <^{\sigma}_{\mathsf{sync}} j$ if $i < j,\ \alpha_i \in \{\langle \underline{\mathsf{wr}}\, w\rangle, \langle \underline{\mathsf{cas}}\, w\rangle\},\ \alpha_j \in \{\langle \underline{\mathsf{rd}}\, w\rangle, \langle \underline{\mathsf{cas}}\, w\rangle\}$
- $i <^{\sigma}_{\mathsf{wr}} j$ if $i' < j',\ \alpha_{i'} = \langle \mathsf{com}\, a\rangle,\ \alpha_{j'} = \langle \mathsf{com}\, b\rangle,\ \alpha_i = \langle \mathsf{wr}\, x\, a\rangle,\ \alpha_j = \langle \mathsf{wr}\, x\, b\rangle$

Here, $<_{\mathsf{sync}}$ is *synchronization order* and $<_{\mathsf{wr}}$ is *(unbuffered) write order*.
Using these relations, we define four *memory* orders and two *commit* orders.

- Define $<^{\sigma}_{\mathsf{seq}}$ to be the transitive closure of $(<^{\sigma}_{\mathsf{thrd}} \cup <^{\sigma}_{\mathsf{ext}} \cup <^{\sigma}_{\mathsf{init}} \cup <^{\sigma}_{\mathsf{var}})$.
- Define $<^{\sigma}_{\mathsf{hb}}$ to be the transitive closure of $(<^{\sigma}_{\mathsf{thrd}} \cup <^{\sigma}_{\mathsf{ext}} \cup <^{\sigma}_{\mathsf{init}} \cup <^{\sigma}_{\mathsf{sync}})$.
- Define $<^{\sigma}_{\mathsf{tso}}$ to be the least transitive relation that includes $(<^{\sigma}_{\mathsf{ext}} \cup <^{\sigma}_{\mathsf{init}} \cup <^{\sigma}_{\mathsf{sync}})$ and satisfies the following, where $\sigma = \alpha_1 \cdots \alpha_n$.
  (1) If $thrd(\alpha_i) \neq thrd(\alpha_j)$ then $i <^{\sigma}_{\mathsf{tso}} j$ whenever $i <^{\sigma}_{\mathsf{rf}} j$ or $i <^{\sigma}_{\mathsf{wr}} j$.
  (2) If $thrd(\alpha_i) = thrd(\alpha_j)$ then $i <^{\sigma}_{\mathsf{tso}} j$ whenever $i < j$, $\alpha_i \neq \langle \mathsf{com}\rangle$, $\alpha_j \neq \langle \mathsf{com}\rangle$, and either (a) $\alpha_i \neq \langle \mathsf{wr}\rangle$, (b) $\alpha_j \neq \langle \mathsf{rd}\rangle$, or (c) $\alpha_i = \langle \mathsf{wr}\, a\rangle$, $\alpha_j = \langle \mathsf{rd}\, a\rangle$ and $i <^{\sigma}_{\mathsf{cb}} j$.
- Define $<^{\sigma}_{\mathsf{pso}}$ similarly to $<^{\sigma}_{\mathsf{tso}}$, replacing clause (b) with (b′) and adding (d):
  (b′) $\alpha_j \notin \{\langle \mathsf{rd}\rangle, \langle \mathsf{wr}\rangle\}$, (d) $\alpha_j = \langle \mathsf{wr}\, x\rangle$ and $\alpha_i \in \{\langle \mathsf{rd}\rangle, \langle \mathsf{wr}\, x\rangle\}$.
- Define $i <^{\sigma}_{\mathsf{compso}} j$ whenever $i < j$ and one of the following holds.
  (1) $\exists a.\ \alpha_i = \langle \mathsf{wr}\, a\rangle$ and $\alpha_j = \langle \mathsf{com}\, a\rangle$. (2) $\exists a, s, t.\ s \neq t$, $\alpha_i = \langle \mathsf{com}\, s\, a\rangle$ and $\alpha_j = \langle t\ \mathsf{rd}\, a\rangle$. (3) $\exists s.\ \alpha_i = \langle \mathsf{com}\, s\rangle$ and $\alpha_j \in \{\langle s\ \underline{\mathsf{wr}}\rangle, \langle s\ \underline{\mathsf{cas}}\rangle\}$. (4) $\exists i' < j' < i.\ \exists a, b.\ \alpha_{i'} = \langle \mathsf{wr}\, a\rangle$, $\alpha_{j'} = \langle s\,!\mathsf{call}\, b\rangle$, $\alpha_i = \langle \mathsf{com}\, a\rangle$, $\alpha_j = \langle ?\mathsf{ret}\, B\rangle$ and $b \in B$. (5) $\exists x.\ \alpha_i = \langle \mathsf{com}\, x\rangle$ and $\alpha_j = \langle \mathsf{com}\, x\rangle$.
- Define $<^{\sigma}_{\mathsf{comtso}}$ similarly to $<^{\sigma}_{\mathsf{compso}}$, adding (6) $\exists s.\ \alpha_i = \langle \mathsf{com}\, s\rangle$ and $\alpha_j = \langle \mathsf{com}\, s\rangle$.

Let $\mathscr{W}$ range over the memory orders in $\{\mathsf{seq}, \mathsf{hb}, \mathsf{tso}, \mathsf{pso}\}$. □

The memory orders relate actions that affect the visibility of values. The (nontransitive) commit orders, $<^{\sigma}_{\mathsf{comtso}}$ and $<^{\sigma}_{\mathsf{compso}}$, relate commit actions to conflicting actions.

All four memory orders include $<_{\mathsf{ext}}$, which specifies orderings guaranteed by the environment, and $<_{\mathsf{init}}$, which specifies initialization. Initial writes are performed by the reserved thread "tinit". For traces of interfaces (which include only I/O actions), the four memory orders coincide.

The definitions of $<_{\mathsf{seq}}$ and $<_{\mathsf{hb}}$ are standard. Relative to hb, clause (1) of the definition of $<_{\mathsf{tso}}$ captures tso's stronger inter-thread dependencies, and clause (2) captures tso's weaker intra-thread dependencies. Two actions of the same thread are ordered unless the first is a write and the second is a read; in this case, they are ordered if the write is committed before the read. With respect to tso, the definition of $<_{\mathsf{pso}}$ removes the ordering between writes of different variables by the same thread.

For each $\mathscr{W}$, we define an operational semantics. The order-theoretic properties that require are $\mathscr{W}$-consistency (no stale reads) and $\mathscr{W}$-closure (no stalled threads).

A trace is $\mathscr{W}$-consistent if none of its read actions are matched with stale writes.

Definition 5.4. Trace $\sigma = \alpha_1 \cdots \alpha_n$ is *$\mathscr{W}$-consistent* if $<^{\sigma}_{\mathscr{W}}$ is antisymmetric and $\forall i, j \in [1, n].\ \alpha_j = \langle \mathsf{rd}\ x \rangle$ and $i <^{\sigma}_{\mathsf{rf}} j$ imply $j \not<^{\sigma}_{\mathscr{W}} i$ and $(\nexists k.\ \alpha_k = \langle \mathsf{wr}\ x \rangle$ and $i <^{\sigma}_{\mathscr{W}} k <^{\sigma}_{\mathscr{W}} j)$. A semantic function is *$\mathscr{W}$-consistent* if every trace it produces is $\mathscr{W}$-consistent. □

A trace set is $\mathscr{W}$-closed if, whenever σ is an allowed trace, then any interleaving consistent with $<^{\sigma}_{\mathscr{W}}$ is also allowed. For example, the following trace is seq-closed, but not tso-, pso- or hb-closed: $\langle \mathsf{tinit\ wr\ x} \rangle \langle \mathsf{s\ wr\ y} \rangle \langle \mathsf{t\ wr\ y} \rangle$.

Definition 5.5. Trace $\rho = \gamma_1 \cdots \gamma_n$ is a *$\mathscr{W}$-permutation* of $\sigma = \alpha_1 \cdots \alpha_n$ via δ, if δ is an injective total function in $[1, n] \to [1, n]$ such that $\forall i, j \in [1, n]$. we have that (1) $\alpha_i \neq \langle ? \rangle$ implies $\gamma_{\delta(i)} = \alpha_i$, (2) $\alpha_i = \langle ?\ A \rangle$ implies $\gamma_{\delta(i)} = \alpha_i \{\![{}^{B}\!/\!{}_{A}]\!\}$ and $B \subseteq A$, (3) $\mathit{thrd}(\alpha_i) = \mathsf{tinit}$ implies $\delta(i) = i$, (4) $i <^{\sigma}_{\mathsf{thrd}} j$ iff $\delta(i) <^{\rho}_{\mathsf{thrd}} \delta(j)$, (5) $i <^{\sigma}_{\mathscr{W}} j$ iff $\delta(i) <^{\rho}_{\mathscr{W}} \delta(j)$, and (6) $i < j$ iff $\delta(i) < \delta(j)$ whenever $\exists w.\ w = \mathit{var}(\alpha_i) = \mathit{var}(\alpha_j)$. When $\mathscr{W} = \mathsf{tso}$, we additionally require (7) $i <^{\sigma}_{\mathsf{comtso}} j$ iff $\delta(i) <^{\rho}_{\mathsf{comtso}} \delta(j)$, and similarly for pso. □

Definition 5.6. Trace set Φ is *$\mathscr{W}$-closed* if whenever $\sigma \in \Phi$ and ρ is an $\mathscr{W}$-permutation of σ, then $\rho \in \Phi$. A semantic function is *$\mathscr{W}$-closed* if every set it produces is $\mathscr{W}$-closed.□

6 Components

Components, M, N, are built using abstractions, Λ, and expressions, C, D. A component declares variables (with an initial value), threads (with an initial expression) and functions (with an abstraction). In addition to base components, there are component constructors for composition and restriction.

$$\begin{aligned}
\Lambda &::= (\vec{z})\{C\} \\
C, D &::= u \mid z \mid x \mid w \mid x\texttt{=}C \mid w\texttt{=}C \mid w\texttt{.cas}(C,D) \mid \mathsf{let}\ \vec{z}\texttt{=}C\texttt{;}D \mid \cdots \\
M, N &::= M \parallel N \mid M \setminus f \mid \mathsf{var}\ x_1\texttt{=}u_1\texttt{;} \cdots \mathsf{var}\ x_\ell\texttt{=}u_\ell\texttt{;}\ \ \mathsf{atomic}\ w_1\texttt{=}v_1\texttt{;} \cdots \mathsf{atomic}\ w_m\texttt{=}v_m\texttt{;} \\
&\qquad\qquad\quad \mathsf{thrd}\ s_1\ C_1\texttt{;} \cdots \mathsf{thrd}\ s_n\ C_n\texttt{;}\ \ \mathsf{fun}\ f_1\ \Lambda_1 \cdots \mathsf{fun}\ f_j\ \Lambda_j
\end{aligned}$$

Data variables are introduced by the keyword var; synchronization variables are introduced by the keyword atomic; registers are introduced by abstractions and let;-expressions. When unspecified, variables initially hold 0. It is important to note that the formal parameters to a function are registers, not shared variables. We require that each component uniquely declare every function and thread name that occurs within it. Variables that are declared in more than one subcomponent are shared, allowing the possibility of interference.

Definition 6.1. A component is *well formed* if (1) it contains at most one declaration for each thread and function name, and (2) all declarations of a variable agree on the initial value. Two components are *compatible* if their composition is well formed. □

Henceforth we consider only well formed components.

For a base component $M =$ “$\mathsf{var}\ \vec{x}{=}\vec{u};\mathsf{atomic}\ \vec{w}{=}\vec{v};\mathsf{thrd}\ \vec{s}\ \vec{C};\mathsf{fun}\ \vec{f}\ \vec{\Lambda}$”, define *funs* $(M) \triangleq \vec{f}$ and $\mathit{thrds}(M) \triangleq \vec{s}$. For aggregate components, define $\mathit{funs}(M \parallel N) = \mathit{funs}(M) \cup \mathit{funs}(N)$ and $\mathit{funs}(M \setminus f) = \mathit{funs}(M)$, and similarly for *thrds*. Note that *funs* returns the set of functions defined by a component, regardless of whether those functions are restricted. For a well formed component $M \parallel N$, we have that $\mathit{funs}(M) \cap \mathit{funs}(N) = \emptyset$.

Definition 6.2. For each memory order $<_{\mathscr{W}}$, Appendix C provides a corresponding operational semantics, defined as a partial function $\mathscr{O}_{\mathscr{W}}$. If $\mathit{thrds}(M) \cap S = \emptyset$ then $\mathscr{O}_{\mathscr{W}}[\![M]\!](S)$ returns a set traces that is coherent, $\mathscr{W}$-consistent and $\mathscr{W}$-closed. □

In $\mathscr{O}_{\mathscr{W}}[\![M]\!](S)$, the threads of $\mathit{thrds}(M)$ are initially active in the component (and quiescent in the environment) whereas the threads of S are initially active in the environment (and quiescent in the component). The operational semantics are unsurprising. We comment only on the role of commit actions. These have a clear operational interpretation under tso and pso; however, both seq- and hb-consistency ignore commit actions. Both $\mathscr{O}_{\mathsf{seq}}$ and $\mathscr{O}_{\mathsf{hb}}$ generate a commit action immediately after each write. This ensures that $\mathscr{O}_{\mathsf{seq}}$ traces are tso-consistent; we do not attempt to interpret $\mathscr{O}_{\mathsf{hb}}$ traces under tso.

To understand the examples, it is important to understand how the operational semantics generates actions from expressions involving memory. (1) Register writes do not create actions; neither do reads. (2) Data variable writes create $\langle \mathsf{wr} \rangle$ actions; reads create $\langle \mathsf{rd} \rangle$ actions. $\langle \mathsf{com} \rangle$ actions are generated immediately after a write in seq and hb; they are generated nondeterministically by tso and pso. (3) Synchronization variable writes create $\langle \underline{\mathsf{wr}} \rangle$ actions; reads create $\langle \underline{\mathsf{rd}} \rangle$ actions. Successful cas operations create $\langle \underline{\mathsf{cas}} \rangle$ actions; unsuccessful cas operations do not create actions.

7 Linearizability

Linearizability is defined in terms of I/O permutations.

Definition 7.1. Write $\alpha \approx \gamma$ if either $\alpha = \gamma$ or $\alpha = \langle\,!\ A \rangle$ and $\gamma = \alpha\{\![{}^{B}/_{A}]\!\}$.

Trace $\sigma = \alpha_1 \cdots \alpha_n$ has *I/O-permutation* $\rho = \gamma_1 \cdots \gamma_m$ via δ, if δ is an injective partial function over $[1,n] \to [1,m]$ such that

- $\forall i \in [1,n].$ if α_i is I/O then $\exists k \in [1,m].\ \alpha_i \approx \gamma_k$ and $\delta(i) = k$, and
- $\forall k \in [1,m].$ if γ_k is I/O then $\exists i \in [1,n].\ \alpha_i \approx \gamma_k$ and $\delta(i) = k$. □

Definition 7.2 (Linearizability). Define $\Phi \vDash_{\mathscr{W}} \Psi$ if every $\sigma = \alpha_1 \cdots \alpha_n \in \Phi$ has an I/O permutation $\rho = \gamma_1 \cdots \gamma_m \in \Psi$ via δ, such that

- $\forall i,j \in [1,n].$ if α_i, α_j are I/O and $\delta(i) <^{\rho}_{\mathscr{W}} \delta(j)$ then $i <^{\sigma}_{\mathscr{W}} j$, and
- $\forall i,j \in [1,n].$ if α_i, α_j are I/O and $i <^{\sigma}_{\mathscr{W}} j$ then $\delta(i) < \delta(j)$. □

The first condition ensures that the orderings required by the specification are preserved in the implementation. The last condition ensures that the ordering on I/O actions in the implementation is reflected in the specification. As the next example illustrates, this is different from the traditional requirement that the ordering on non-overlapping I/O actions be reflected in the specification.

Example 7.3. As a simple example, consider the following unsynchronized counter.

```
var x;
fun inc() { let tmp=x; tmp=tmp+1; x=tmp; return tmp }          (Inc)
```

At first glance, we might expect this implementation to satisfy a specification which requires that the return values be non-decreasing; that is, we expect traces of form

$$\langle \mathsf{s?call\ inc}\rangle\langle \mathsf{s!ret}\ u_0\rangle\ \langle \mathsf{t?call\ inc}\rangle\langle \mathsf{t!ret}\ u_1\rangle\ \langle \mathsf{r?call\ inc}\rangle\langle \mathsf{r!ret}\ u_2\rangle\cdots$$

where $u_i \geq u_{i-1}$. Although this specification contains no ordering on actions, the implementation does not satisfy it, for seq, tso or pso, due to the lack of synchronization. To see this, consider a call by one thread with overlapping and following calls by another.

Our results allow us to consider whether the implementation satisfies the specification if clients are constrained so that threads do not synchronize and each thread may call inc() at most once. In this case, we can answer affirmatively for all four models.

To illuminate the definition of linearizability, consider the following traces. (We elide the commit actions that immediately follow each write.) Inc generates the first trace under all memory models, but the second, only under hb.

$$\langle \mathsf{t?call\ inc}\rangle\langle \mathsf{s?call\ inc}\rangle\langle \mathsf{s\ rd\ x\ 0\ init}\rangle\langle \mathsf{s\ wr\ x\ 1\ a}\rangle\langle \mathsf{s!ret\ 1}\rangle\langle \mathsf{t\ rd\ x\ 1\ a}\rangle\langle \mathsf{t\ wr\ x\ 2\ b}\rangle\langle \mathsf{t!ret\ 2}\rangle$$
$$\langle \mathsf{t?call\ inc}\rangle\langle \mathsf{t\ rd\ x\ 1\ a}\rangle\langle \mathsf{t\ wr\ x\ 2\ b}\rangle\langle \mathsf{t!ret\ 2}\rangle\ \langle \mathsf{s?call\ inc}\rangle\langle \mathsf{s\ rd\ x\ 0\ init}\rangle\langle \mathsf{s\ wr\ x\ 1\ a}\rangle\langle \mathsf{s!ret\ 1}\rangle$$

For each $\mathscr{W} \in \{\mathsf{seq}, \mathsf{tso}, \mathsf{pso}\}$, the first trace is linearizable under $\models_{\mathscr{W}}$, whereas the second trace is not. The write and subsequent read of the shared variable creates order between threads (condition (2c) and (2d) for tso and pso) and thus we have $\langle \mathsf{t?call\ inc}\rangle <_{\mathscr{W}} \langle \mathsf{s!ret\ 1}\rangle$ in the second trace. This causes the last clause of Definition 7.2 to fail.

Touching a shared data variable creates no ordering under hb, and therefore both traces are linearizable under $\models_{\mathsf{hb}}$. This would not be the case if we were to adopt the traditional requirement for linearizability: that the order of non-overlapping method calls be respected. This would also not be the case if the last clause of Definition 7.2 required $\delta(i) <^{\rho}_{\mathscr{W}} \delta(j)$ rather than $\delta(i) < \delta(j)$, since $(<^{\rho}_{\mathscr{W}})$ is the empty relation for every specification trace ρ. □

Example 7.4. Suppose we have an implementation trace of the form

$$\langle \mathsf{s?call\ inc}\rangle\langle \mathsf{s!ret}\ u_0\ \mathsf{a}\ \emptyset\rangle\ \langle \mathsf{t?call\ inc}\ \{\mathsf{a}\}\rangle\langle \mathsf{t!ret}\ u_1\ \mathsf{b}\rangle\ \langle \mathsf{r?call\ inc}\ \{\mathsf{b}\}\rangle\langle \mathsf{r!ret}\ u_2\rangle\cdots$$

where the client has imposed ordering between each method return and the subsequent call. The definition of linearizability requires that the specification have exactly the same use sets, and thus the same client ordering. In this case, the specification may be more constrained. For example, it might require that $u_i > u_{i-1}$. □

Example 7.5. The following example is drawn from java.lang.String.hashCode. The specification requires that every call to hashCode return the same value. The implementation has a benign write-write data race.

```
var hash;
fun hashCode() { let h=hash;                                   (Hash)
  if h!=0 then { return h } else { let h=42; hash=h; return h } }
```

Here, we set hash to 42; in a real implementation, the value is derived from immutable fields of the object. hash is always set to the same value, regardless of the number of threads that call hashCode simultaneously. The intended sequential interface specification for Hash is:

(⟨s?call hashCode⟩⟨s!ret hashCode 42⟩)*

Hash satisfies its sequential specification under all memory models. □

We consider two implementations of an atomic pair, inspired by an example in [Burckhardt et al. 2012]. The specification requires that the get return the pair of values specified by the preceding set:

(⟨s?call set (u, v) a⟩⟨s!ret⟩ (⟨t?call get⟩⟨t!ret (u, v) {a}⟩)*)*

Example 7.6. The first implementation is fully synchronized using locks.

```
var x1; var x2; atomic lock;
fun set(z1,z2) { do skip until lock.cas(0,1); x1=z1; x2=z2; lock=0 }          (Pair1)
fun get() { do skip until lock.cas(0,1); let z1=x1; let z2=x2; lock=0; return z1,z2 }
```

Pair1 is linearizable under all memory models. The cas on the atomic variable provides the required order relation. The linearization point can be chosen to be the successful cas operation in both the methods. The specification also requires an order relationship from the call of set to the return of get as seen in the subsequence ⟨s?call set (v_1, v_2) a⟩ $\cdots$ ⟨t!ret get (v_1, v_2) {a}⟩. The order from the write of the atomic variable lock in set to the successful cas on lock in get establishes this relationship in the implementation. □

Example 7.7. The second implementation uses locking for set, but not get. The version variable i is odd if and only if there is a write in progress.

```
var x1; var x2; var i; atomic lock;
fun set(z1,z2) { do skip until lock.cas(0,1); i++; x1=z1; x2=z2; i++; lock=0 }
fun get() { while (1){ let j=i; if even(j) then let z1=x1; let z2=x2;          (Pair2)
                                    if j==i then return z1,z2 } }
```

Pair2 exemplifies a publication idiom characteristic of tso, allowing data races between writes and reads. Pair2 is also not linearizable under pso or hb.

Pair2 is linearizable under tso. A candidate linearization point for set is the first increment of i. The linearization point for get is the successful check of the counter i. Pair1 and Pair2 share the same specification, so the specification requires the same order relationship from the call of set to the return from get. The second condition of the definition of $<_{\mathsf{tso}}$ on the counter i, from the write in set to the read in get, yields the required order. Neither pso nor hb provide this ordering. □

Example 7.8. The next example is an “active” component, which implements an asynchronous function handler. This can be seen as a simplified thread pool, with a single, one-shot thread. Let v' be the result of performing the operation *op* on v.

$$\langle\mathsf{s?call\ send\ v\ a}\rangle\langle\mathsf{s!ret\ true}\rangle\ (\ (\ \langle\mathsf{t?call\ get}\rangle\langle\mathsf{t!ret\ v'}\ \{\mathsf{a}\}\rangle\ )\ |\ \langle\mathsf{r?call\ send\ u}\rangle\langle\mathsf{r?ret\ false}\rangle\ )^*$$

The first call to send succeeds, and calls to get return a value derived from its parameter. Subsequent calls to send return false.

```
var x; var y; atomic lock; atomic start; atomic stop;
fun send(z){ do { if (start==1) then return false } until lock.cas(0,1);
             x=z; start=1; return true }                                  (Async)
fun get()   { do skip until stop==1; return y }
thread wrk  { do skip until start==1; y=op(x); stop=1 }
```

Async satisfies its sequential specification for all four memory models.

A candidate linearization point for send is the successful cas or reading start==1, depending on which path is taken. The linearization point for the worker thread wrk and get is the point of exit from the loops, via the variables start and stop, respectively. The specification requires an order relationship as seen in the subsequence $\langle\mathsf{s?call\ send\ v\ a}\rangle \cdots \langle\mathsf{t!ret\ v'}\ \{\mathsf{a}\}\rangle$. The implementation establishes this by combining two order relations yielded by atomic variables: start links send to wrk and stop links wrk to get. □

Example 7.9. Async can be generalized to a thread pool which satisfies interface traces such as the following, where let v' be the result of performing some computation on v and j is a job identifier.

$$\langle\mathsf{s?call\ send\ v\ a}\rangle\langle\mathsf{s!ret\ j}\rangle\langle\mathsf{r?call\ get\ j}\rangle\langle\mathsf{r!ret\ v'}\ \{\mathsf{a}\}\rangle$$

If the thread pool generates unique job identifiers, then it should be able to guarantee the happens-before relation given in the specification.

We describe an implementation parameterized on a bounded buffer and map. The bounded buffer holds waiting jobs and the map holds waiting results. Due to the complexity of the possible interleavings, we give exemplary traces rather than complete specifications. The implementation is straightforward.

The bounded buffer is an adaption of Buffer given in the introduction. To accomodate the example, the buffer holds pairs of values. If the buffer is FILO, then the sequential interface will include traces such as the following.

$$\langle\mathsf{s?call\ bput}\ (1,10)\ \mathsf{a}\rangle\langle\mathsf{s!ret}\rangle\ \langle\mathsf{t?call\ bput}\ (1,10)\ \mathsf{b}\rangle\langle\mathsf{t!ret}\rangle$$
$$\langle\mathsf{r?call\ bget}\rangle\langle\mathsf{r!ret}\ (1,10)\ \{\mathsf{b}\}\rangle\ \langle\mathsf{q?call\ bget}\rangle\langle\mathsf{q!ret}\ (1,10)\ \{\mathsf{a}\}\rangle$$

Note that the same value is put twice, by different threads. The use sets in the get actions indicate the FILO order, even though the values do not.

The map is similar. Here is an example showing a value that is retrieved twice.

$$\langle\mathsf{s?call\ mput}\ (1,10)\ \mathsf{a}\rangle\langle\mathsf{s!ret}\rangle\ \langle\mathsf{t?call\ mput}\ (1,10)\ \mathsf{b}\rangle\langle\mathsf{t!ret}\rangle$$
$$\langle\mathsf{r?call\ mget\ 1}\rangle\langle\mathsf{r!ret\ 10}\ \{\mathsf{b}\}\rangle\ \langle\mathsf{q?call\ mget\ 1}\rangle\langle\mathsf{q!ret\ 10}\ \{\mathsf{b}\}\rangle$$

Assuming a bounded buffer and map, the general thread pool has traces such as the following. For clarity, we show the function name on return actions.

$$\langle\mathsf{s?call\ send\ v\ a}\rangle\ \ \langle\mathsf{s!call\ bput}\ (\mathsf{v},\mathsf{j})\ \mathsf{b}\rangle\langle\mathsf{s?ret\ bput}\rangle\ \ \langle\mathsf{s!ret\ send\ j}\rangle$$
$$\langle\mathsf{wrk!call\ bget}\rangle\langle\mathsf{wrk?ret\ bget}\ (\mathsf{v},\mathsf{j})\ \{\mathsf{b}\}\rangle\langle\mathsf{wrk!call\ mput}\ (\mathsf{j},\mathsf{v}')\ \mathsf{c}\rangle\langle\mathsf{wrk?ret\ mput}\rangle$$
$$\langle\mathsf{r?call\ get\ j}\rangle\ \ \langle\mathsf{r!call\ mget\ j}\rangle\langle\mathsf{r?ret\ mget\ v'}\ \{\mathsf{c}\}\rangle\ \ \langle\mathsf{r!ret\ get\ v'}\ \{\mathsf{a}\}\rangle$$

The first line shows a client calling send with argument v. The thread pool creates a new job id j, stores the job in the buffer and returns j. Subsequently, the second line show a worker thread retrieving the job from the buffer, computing v′, and storing the result in the map. Finally, the third line shows a client thread retrieving the result using a call to get j; in response, the thread pool retrieves j from the map and returns the corresponding value. In $\langle$!ret {a}$\rangle$, the decoration is a guarantee, similar to the decorations in previous examples: the thread pool guarantees that there will be memory effects between the call and corresponding return.

Consider the projection of this trace of the thread pool on the methods of the bounded buffer. We get:

$$\langle \text{s!call bput (v, j) b}\rangle\langle\text{s?ret bput}\rangle\ \ \langle\text{wrk!call bget}\rangle\langle\text{wrk?ret bget (v, j) \{b\}}\rangle$$

The sequence of calls to the buffer methods, and the values returned by them, line up with the trace of the buffer presented above. Furthermore, so do the label sets. In $\langle$s!call bput (v, j) b$\rangle$ and $\langle$wrk?ret bget (v, j) {b}$\rangle$, the label b indicates an assumption made by the thread pool on the bounded buffer. In the matching actions, $\langle$s?call bput (v, j) b$\rangle$ and $\langle$wrk!ret bget (v, j) {b}$\rangle$), the label b indicates a guarantee provided by the bounded buffer interface to the thread pool. Here one can recognize the semantic ingredients necessary for a full higher-order multiplicative linear logic of interfaces, perhaps in the style of Interaction Categories [Abramsky et al. 1996]. In this paper, however, we do not pursue this further. □

8 Proving Linearizability

We explore methods to quarantine data race free programs from the subtleties of relaxed memory models. First, we define a component to be *locally sequential consistent* when its SC traces provide a complete description of all its traces—or in the terminology of [Filipovic et al. 2010], when the set of its SC traces is an operational refinement of all of its traces.

Definition 8.1. Define $\sigma \sim_{\mathscr{W}} \rho$ when (1) $\sigma = \sigma_0\gamma_1\sigma_1\cdots\gamma_n\sigma_n$ and $\rho = \rho_0\gamma_1\rho_1\cdots\gamma_n\rho_n$ for some $\vec{\sigma}$, $\vec{\rho}$, $\vec{\gamma}$ such that each $\vec{\sigma}$ and $\vec{\rho}$ contains only write and commit actions, and (2) for every read action α, $\sigma\alpha$ is $\mathscr{W}$-consistent if and only if $\rho\alpha$ is $\mathscr{W}$-consistent.

A set of traces Φ is *locally sequentially consistent (LSC)* for $\mathscr{W}$ if

$$\forall\sigma \in \Phi.\ \exists\sigma' \in \Phi.\ \sigma \sim_{\mathscr{W}} \sigma' \text{ and } \sigma' \text{ is seq-consistent.} \qquad \square$$

Intuitively, a set is LSC if every trace can be matched by a seq-consistent trace in the set, where all non-write/non-commit actions must match exactly and in the same order (condition 1), and the reads available at the end are the same (condition 2).

Example 8.2. Inc is not LSC for any of the weak models. Hash is LSC for all four memory models. This demonstrates that LSC does not require the absence of data races.

Pair1 and Async are LSC for all four memory models; however, Pair2 is not LSC under any of the relaxed models. To see this, consider traces in which there is a completed call to set with parameters (1,1) and a subsequent call to get returning (1,1).

In every such trace, the write actions must occur before the call to get. Of these traces, choose one in which the loop in get initially fails because $\mathsf{i} \neq \mathsf{j}$. This trace will not be equivalent to any SC trace, since it must see a stale value. □

We describe a sufficient condition to establish that a set is LSC.

Definition 8.3. Actions *conflict* if one is a write to a data variable and the other is a read or write to the same variable. Trace $\sigma = \alpha_1 \alpha_1 \cdots \alpha_n$ is *locally data race free (LDRF)* if whenever α_i and α_j conflict then either $i <^{\sigma}_{\mathsf{hb}} j$ or $j <^{\sigma}_{\mathsf{hb}} i$. A set of traces is LDRF if every member is LDRF. □

Example 8.4. All of the examples from Section 7 are LDRF for all four memory models, with the exception of Inc, Hash and Pair2, which are not LDRF for any model. □

Proposition 8.5. Any trace that is LDRF and $\mathscr{W}$-consistent is also seq*-consistent.* □

Proposition 8.5 demonstrates that to establish that a component is LSC, it suffices to show that all of its traces are LDRF. This, in turn, can be established by various standard techniques for detecting data races. For tso, there is a weaker condition, "triangular race freedom", that suffices to establish that a component is LSC [Owens 2010].

In order to reason about a program using the SC semantics, we must ensure that the weak semantics is consistent with $\mathscr{O}_{\mathsf{seq}}$, in the sense that any seq-consistent trace generated by the weak semantics can also be generated by $\mathscr{O}_{\mathsf{seq}}$. All of the semantic functions we consider have this property.

Definition 8.6. A semantic function $\mathscr{S}$ is *consistent with* $\mathscr{O}_{\mathsf{seq}}$ if whenever $\sigma \in \mathscr{S}[\![M]\!]$ (S) and σ is seq-consistent then $\sigma \in \mathscr{O}_{\mathsf{seq}}[\![M]\!](S)$. □

LSC components can be quarantined. For LSC programs, it is sometimes possible to use traditional SC techniques to reason about linearizability, even in a relaxed setting. The restrictions should be unsurprising to readers familiar with [Filipovic et al. 2010], which states that "OSC observationally refines OSA iff OSC is linearizable with respect to OSA, assuming that client operations may use at *least one shared global variable*." For such programs, our results allow proof techniques developed in the SC setting to apply to relaxed models.

A trace is *strongly I/O-ordered for $\mathscr{W}$* if there is a $<_{\mathscr{W}}$ order between every input and output. Formally, $\sigma = \alpha_1 \cdots \alpha_n$ is strongly I/O-ordered for $\mathscr{W}$ if whenever α_i is input and α_j is output and $i < j$ then $i <^{\sigma}_{\mathscr{W}} j$[1]. Let *erase*$(\sigma)$ be the trace derived from σ by replacing every name set occurring in return actions by the empty set; this has the effect of removing all of the happens-before relations from an interface.

Proposition 8.7. Let $\mathscr{S}$ be a semantic function that is $\mathscr{W}$-consistent and consistent with $\mathscr{O}_{\mathsf{seq}}$. Let Ψ be a sequential interface. Let $\mathscr{S}[\![M]\!](S)$ be strongly I/O-ordered and LSC for $\mathscr{W}$. Then $\mathscr{O}_{\mathsf{seq}}[\![M]\!](S) \vDash_{\mathsf{seq}}$ erase(Ψ) implies $\mathscr{S}[\![M]\!](S) \vDash_{\mathscr{W}} \Psi$. □

[1] This corrects an error in the ESOP version, which reads "Formally, $\sigma = \alpha_1 \cdots \alpha_n$ is I/O-ordered for $\mathscr{W}$ if whenever α_i and α_j are input/output-bracketed then $i <^{\sigma}_{\mathscr{W}} j$." This condition is vacuous for operationally generated traces, since I/O bracketed actions must have the same thread, and therefore are always related by $<^{\sigma}_{\mathscr{W}}$.

Here $\mathscr{O}_{\mathsf{seq}}[\![M]\!](S) \vDash_{\mathsf{seq}} \mathit{erase}(\Psi)$ is similar to traditional linearizability. The use of *erase* (Ψ) ensures that the proof obligation is indeed the traditional one: ordering requirements are removed. I/O-ordering of the implementation and sequentiality of the specification are required to ensure that the order can be recovered.

In Corollary 10.4, we show that LSC clients can be isolated from the subtleties of relaxed memory used in the implementations of (even racy) libraries.

9 Composition

In order to state properties of linearizability, we must first define semantic versions of restriction and composition. Restriction is straightforward.

Definition 9.1. Let $\mathit{incalls}(\alpha_1 \cdots \alpha_n) \triangleq \{f \mid \exists i.\ \alpha = \langle ?\mathsf{call}\ f\rangle\}$.
Then $\Phi \setminus F \triangleq \{\sigma \in \Phi \mid \mathit{incalls}(\sigma) \cap F = \emptyset\}$. □

Definition 9.2. An action sequence π is a *collapsed interleaving*[2] of σ and ρ if there exists a π' such that (1) all actions of tinit occur at the beginning of π', (2) π' is an interleaving of σ and ρ, and (3) π is derived from π' by (3a) replacing every subsequence $\langle s\,!\mathsf{call}\ f\ \vec{u}\ a\ A\rangle\langle s?\mathsf{call}\ f\ \vec{u}\ a\ A\rangle$ by $\langle s\,.\,\mathsf{call}\ f\ \vec{u}\ a\ A\rangle$, (3b) replacing every subsequence $\langle s\,!\,\mathsf{ret}\ \vec{u}\ a\ A\rangle\langle s?\mathsf{ret}\ \vec{u}\ a\ A\rangle$ by $\langle s\,.\,\mathsf{ret}\ \vec{u}\ a\ A\rangle$, (3c) replacing every action $\langle ?\ A\rangle$ from σ by $\langle ?\ (A \cup B)\rangle$, where B is any subset of the preceding actions of ρ, and (3d) repeating (3c) swapping σ and ρ. □

Definition 9.3 (Composition). Let $\mathit{intern}(\Phi) = H$ and $\mathit{intern}(\Psi) = G$. If $H \cap G = \emptyset$, then define $\Phi \otimes \Psi$ to be the set of traces, π, such that $\mathit{extern}(\pi) \cap (H \cup G) = \emptyset$, and π is a collapsed interleaving of some $\sigma \in \Phi$ and $\rho \in \Psi$. □

In Appendix B, we provide an inductive characterization of composition and discuss its properties.

Example 9.4. Here are some single threaded examples to illustrate the definition. We elide the thread identifier. $\{\langle ?\mathsf{call\ f}\rangle\} \otimes \{\langle ?\mathsf{call\ f}\rangle\}$ and $\{\langle \mathsf{wr}\rangle\} \otimes \{\langle \mathsf{wr}\rangle\}$ are undefined because their *intern* overlap; the first pair on f, the second, on the thread identifier.

Composition forces complete synchronization on invocations of functions that are defined in either component, but permits interleaving of invocations of functions that are undefined in both components. Let $\mathscr{C}$ perform prefix closure.

$$\begin{aligned}
&\mathscr{C}\{\langle ?\mathsf{call\ f\ 0}\rangle\langle !\mathsf{ret}\rangle\} \otimes \mathscr{C}\{\langle \mathsf{wr}\rangle\} &&= \mathscr{C}\{\langle \mathsf{wr}\rangle\}\\
&\mathscr{C}\{\langle ?\mathsf{call\ f\ 0}\rangle\langle !\mathsf{ret}\rangle\} \otimes \mathscr{C}\{\langle !\mathsf{call\ f\ 1}\rangle\} &&= \mathscr{C}\{\varepsilon\}\\
&\mathscr{C}\{\langle ?\mathsf{call\ f\ 0}\rangle\langle !\mathsf{ret}\rangle\} \otimes \mathscr{C}\{\langle !\mathsf{call\ f\ 0}\rangle\langle ?\mathsf{ret}\rangle\} &&= \mathscr{C}\{\langle .\,\mathsf{call\ f\ 0}\rangle\langle .\,\mathsf{ret}\rangle\}\\
&\mathscr{C}\{\langle ?\mathsf{call\ f\ 0}\rangle\langle !\mathsf{ret}\rangle\} \otimes \mathscr{C}\{\langle ?\mathsf{call\ g\ 0}\rangle\langle !\mathsf{ret}\rangle\} &&= \mathscr{C}\{\langle ?\mathsf{call\ g}\rangle\langle !\mathsf{ret\ 0}\rangle\langle ?\mathsf{call\ f}\rangle\langle !\mathsf{ret\ 0}\rangle,\\
& && \qquad \langle ?\mathsf{call\ f}\rangle\langle !\mathsf{ret\ 0}\rangle\langle ?\mathsf{call\ g}\rangle\langle !\mathsf{ret\ 0}\rangle\}
\end{aligned}$$

[2] Conditions (3c) and (3d) are missing from the ESOP version, and thus the tensor generates incomplete sets. The incomplete tensor fails to validate compositionality of the operational semantics. See Definition 10.2.

Consider the following traces, where α_{11}–α_{32} are arbitrary memory actions. Both the first and second traces include calls to f, which is defined by third trace. The first trace also includes a call to g, which is defined by the second trace.

$$\alpha_{11}\langle\textsf{!call g}\rangle\langle\textsf{?ret}\rangle\alpha_{12}\langle\textsf{!call f}\rangle\langle\textsf{?ret}\rangle$$
$$\langle\textsf{?call g}\rangle\alpha_{21}\langle\textsf{!call f}\rangle\langle\textsf{?ret}\rangle\alpha_{22}\langle\textsf{!ret}\rangle$$
$$\langle\textsf{?call f}\rangle\alpha_{31}\langle\textsf{!ret}\rangle\langle\textsf{?call f}\rangle\alpha_{32}\langle\textsf{!ret}\rangle$$

The first two compose to $\alpha_{11}\langle\textsf{.call g}\rangle\alpha_{21}\langle\textsf{!call f}\rangle\langle\textsf{?ret}\rangle\alpha_{22}\langle\textsf{.ret}\rangle\alpha_{12}\langle\textsf{!call f}\rangle\langle\textsf{?ret}\rangle$.
Composing the second and third gives $\langle\textsf{?call f}\rangle\alpha_{31}\langle\textsf{!ret}\rangle\langle\textsf{?call g}\rangle\alpha_{21}\langle\textsf{.call f}\rangle\alpha_{32}\langle\textsf{.ret}\rangle\alpha_{22}\langle\textsf{!ret}\rangle$ and $\langle\textsf{?call g}\rangle\alpha_{21}\langle\textsf{.call f}\rangle\alpha_{31}\langle\textsf{.ret}\rangle\alpha_{22}\langle\textsf{!ret}\rangle\langle\textsf{?call f}\rangle\alpha_{32}\langle\textsf{!ret}\rangle$.
Composing the first and third gives $\alpha_{11}\langle\textsf{!call g}\rangle\langle\textsf{?call f}\rangle\alpha_{31}\langle\textsf{!ret}\rangle\langle\textsf{?ret}\rangle\alpha_{12}\langle\textsf{.call f}\rangle\alpha_{32}\langle\textsf{.ret}\rangle$. Composing all three gives

$$\alpha_{11}\langle\textsf{.call g}\rangle\alpha_{21}\langle\textsf{.call f}\rangle\alpha_{31}\langle\textsf{.ret}\rangle\alpha_{22}\langle\textsf{.ret}\rangle\alpha_{12}\langle\textsf{.call f}\rangle\alpha_{32}\langle\textsf{.ret}\rangle. \qquad \square$$

Example 9.5. For any single trace, the order of cross-thread actions is fixed. Thus, composing $\langle\textsf{s?call f}\rangle\langle\textsf{t wr}\rangle$ and $\langle\textsf{s!call f}\rangle$ produces only $\langle\textsf{s.call f}\rangle\langle\textsf{t wr}\rangle$. □

10 Properties of linearizability

We present the results using the most general client. More general results can be found in the appendix.

Definition 10.1 (Interference freedom). Two components are *interference free* if they are compatible (Definition 6.1) and declare disjoint variables. □

Definition 10.2 (Compositionality). A semantic function $\mathscr{S}$ is *compositional* if
(1) $\mathscr{S}\llbracket M \setminus F\rrbracket(S) = \mathscr{S}\llbracket M\rrbracket(S) \setminus F$ and,
(2) $\mathscr{S}\llbracket M \parallel N\rrbracket(S) \subseteq \mathscr{S}\llbracket M\rrbracket(S) \otimes \mathscr{S}\llbracket N\rrbracket(S)$, whenever M and N are interference free.□

Proposition 10.3 (Abstraction). Let $\mathscr{S}$ be coherent, compositional and $\mathscr{W}$-closed. Let M_L and M_C be interference free. If $\mathscr{S}\llbracket M_L\rrbracket(S) \vDash_{\mathscr{W}} \Psi_L$ and $\mathscr{S}\llbracket M_C\rrbracket(S) \otimes \Psi_L \vDash_{\mathscr{W}} \Psi_C$ then $\mathscr{S}\llbracket M_C \parallel M_L\rrbracket(S) \vDash_{\mathscr{W}} \Psi_C$. □

Consider the Lock discussed in the introduction. If we are given that (1) the lock implements its specification (that is, $\mathscr{S}\llbracket\mathsf{Lock}\rrbracket(\mathsf{S}) \vDash_{\mathscr{W}} \Psi_{\mathsf{lock}}$) and (2) the one place buffers implements its specification when it uses the lock specification (that is, $\mathscr{S}\llbracket\mathsf{Buffer}\rrbracket(\mathsf{S}) \otimes \Psi_{\mathsf{lock}} \vDash_{\mathscr{W}} \Psi_{\mathsf{buf}}$), then the theorem allows us to deduce that the implementation of the buffer realizes its specification ($\mathscr{S}\llbracket\mathsf{Buffer} \parallel \mathsf{Lock}\rrbracket(\mathsf{S}) \vDash_{\mathscr{W}} \Psi_{\mathsf{buf}}$).

Corollary 10.4 (Quarantining weakness). Let $\mathscr{S}$ be coherent, compositional and $\mathscr{W}$-closed. Let M_L and M_C be interference free. Let Ψ_L and Ψ_C be sequential interfaces. Suppose $\Psi_L = erase(\Psi_L)$, $\mathscr{S}\llbracket M_C\rrbracket(S)$ is LSC and either (1) $erase(\Psi_C) = \Psi_C$ or (2) $\mathscr{S}\llbracket M_C\rrbracket(S)$ is I/O-ordered. If $\mathscr{S}\llbracket M_L\rrbracket(S) \vDash_{\mathscr{W}} \Psi_L$ and $\mathscr{S}\llbracket M_C\rrbracket(S) \otimes \Psi_L \vDash_{\mathsf{seq}} \Psi_C$ then $\mathscr{S}\llbracket M_C \parallel M_L\rrbracket(S) \vDash_{\mathscr{W}} \Psi_C$. □

Corollary 10.4 demonstrates that well-synchronized clients (that do not depend on the library for synchronization), are not affected by data races in the library. Consider the unsynchronized counter Inc from Examples 7.3-7.4. A fully-synchronized client can safely use the library without regard to its data races; for example, a fully-synchronized counter can be built using the unsynchronized one.

11 Conclusion

This paper investigates reasoning about concurrent data structures, with a special focus on isolating the complexity wrought by relaxed memory models. We have presented an adaptation of linearizability that accounts for relaxed memory and provided ways to reason compositionally. Our treatment is parametric with respect to the memory model, with the required properties of the memory model confined to a couple of key properties. We have been able to address SC, TSO, PSO and (a variant of) the JMM in this style.

References

S. Abramsky, S. J. Gay, and R. Nagarajan. Interaction categories and the foundations of typed concurrent programming. In *NATO ASI DPD*, pages 35–113, 1996.

S. V. Adve and H.-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM*, 53:90–101, 2010.

S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29 (12):66–76, 1996.

R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.

M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, pages 55–66. ACM, 2011.

M. Batty, M. Dodds, and A. Gotsman. Library abstraction for C/C++ concurrency. In *POPL*, 2013. To appear.

H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, pages 68–78. ACM, 2008.

S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In *ESOP*, pages 87–107, 2012.

I. Filipovic, P. O'Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. *Theoretical Comp. Sci.*, 411:4379–4398, 2010.

A. Gotsman and H. Yang. Linearizability with ownership transfer. In *CONCUR*, volume 7454 of *LNCS*, pages 256–271, 2012.

A. Gotsman, M. Musuvathi, and H. Yang. Show no weakness: sequentially consistent specifications of TSO libraries. In *DISC'12*, 2012. To appear.

M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

R. Jagadeesan, C. Pitcher, and J. Riely. Generative operational semantics for relaxed memory models. In *ESOP*, pages 307–326, 2010.

A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. *Theoretical Comp. Sci.*, 338:17–63, 2005.

L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, pages 378–391, 2005.

S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, volume 6183 of *LNCS*, pages 478–503, 2010.

S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding power multiprocessors. In *PLDI*, pages 175–186. ACM, 2011.

J. Sevcík. *Program Transformations in Weak Memory Models*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 2008.

P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

SPARC, Inc. *The SPARC Architecture Manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

Sun Microsystems. http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/atomic/package-summary.html, 2004.

A Traces

We formalize the notion of well-formed trace from Sections 4 and 5. Recall Definition 4.1 of traces. Recall the syntax of actions (Sections 4 and 5).

$$\begin{aligned}\alpha, \gamma \ ::= {} & \langle s?\mathsf{call}\, f\, \vec{u}\, a\rangle \mid \langle s!\mathsf{call}\, f\, \vec{u}\, a\rangle \mid \langle s\,.\,\mathsf{call}\, f\, \vec{u}\, a\rangle \\ & \mid \langle s!\mathsf{ret}\, f\, \vec{u}\, A\rangle \mid \langle s?\mathsf{ret}\, f\, \vec{u}\, B\rangle \mid \langle s\,.\,\mathsf{ret}\, f\, \vec{u}\, A\, B\rangle \\ & \mid \langle s\ \mathsf{wr}\, x\, u\, a\rangle \mid \langle s\ \mathsf{rd}\, x\, u\, a\rangle \mid \langle \mathsf{com}\, s\, x\, a\rangle \\ & \mid \langle s\ \underline{\mathsf{wr}}\ w\rangle \mid \langle s\ \underline{\mathsf{rd}}\ w\rangle \mid \langle s\ \underline{\mathsf{cas}}\ w\rangle\end{aligned}$$

Define *defs*!, *uses*!, *defs*?, *uses*?, *defs*., *uses*., *wr-defs* and *wr-uses* as follows.

$$\mathit{defs}!(\alpha) \triangleq \begin{cases}\{a\} & \text{if } \alpha = \langle !\mathsf{call}\, a\rangle \\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathit{uses}!(\alpha) \triangleq \begin{cases}A & \text{if } \alpha \in \langle !\mathsf{ret}\, A\rangle \\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathit{defs}?(\alpha) \triangleq \begin{cases}\{a\} & \text{if } \alpha = \langle ?\mathsf{call}\, a\rangle \\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathit{uses}?(\alpha) \triangleq \begin{cases}A & \text{if } \alpha \in \langle ?\mathsf{ret}\, A\rangle \\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathit{defs}.(\alpha) \triangleq \begin{cases}\{a\} & \text{if } \alpha = \langle .\,\mathsf{call}\, a\rangle \\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathit{uses}.(\alpha) \triangleq \begin{cases}A & \text{if } \alpha \in \langle .\,\mathsf{ret}\, A\rangle \\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathit{wr\text{-}defs}(\alpha) \triangleq \begin{cases}\{a\} & \text{if } \alpha \in \{\langle \mathsf{wr}\, a\rangle\} \\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathit{wr\text{-}uses}(\alpha) \triangleq \begin{cases}\{a\} & \text{if } \alpha \in \{\langle \mathsf{rd}\, a\rangle, \langle \mathsf{com}\, a\rangle\} \\ \emptyset & \text{otherwise}\end{cases}$$

Let *defs* be the union of all the definitions in a trace or action, and similarly for *uses*.

Definition A.1. Define *def-use well-formed* traces as follows: ε is def-use well-formed. $\sigma\alpha$ is def-use well-formed if σ is def-use well-formed and $\sigma\alpha$ satisfies the following.

(a) $(\mathit{defs}(\sigma)) \cap (\mathit{defs}(\alpha)) = \emptyset$,
(b) $\mathit{defs}!(\sigma) \supseteq \mathit{uses}?(\alpha)$,
(c) $\mathit{defs}?(\sigma) \supseteq \mathit{uses}!(\alpha)$,
(d) $\mathit{defs}.(\sigma) \supseteq \mathit{uses}.(\alpha)$, and
(e) $\mathit{wr\text{-}defs}(\sigma) \supseteq \mathit{wr\text{-}uses}(\alpha)$.

Trace $\alpha_1 \cdots \alpha_n$ is *well formed* if it is def-use well-formed, coherent (Definition 4.2) and satisfies the following.

(f) if $\alpha_i = \langle \mathsf{com}\ a_i \rangle$ and $\alpha_j = \langle \mathsf{com}\ a_j \rangle$, then $a_i \neq a_j$.
(g) if $\alpha_i = \langle s_i\ \mathsf{wr}\ x_i\ a \rangle$ and $\alpha_j = \langle \mathsf{com}\ s_j\ x_j\ a \rangle$, then $s_i = s_j$ and $x_i = x_j$.
(h) if $\alpha_i = \langle \mathsf{wr}\ x_i\ u_i\ a \rangle$ and $\alpha_j = \langle \mathsf{rd}\ x_j\ u_j\ a \rangle$, then $x_i = x_j$ and $u_i = u_j$.
(i) if $\mathit{thrd}(\alpha_i) = \mathsf{tinit}$ then $\alpha_i \in \{\langle \mathsf{wr} \rangle, \langle \mathsf{com} \rangle\}$.
(j) if $\alpha_i = \langle \mathsf{tinit}\ \mathsf{wr}\ a \rangle$ then $\alpha_{i+1} = \langle \mathsf{com}\ a \rangle$ and $\forall j.\ \mathit{thrd}(\alpha_j) \neq \mathsf{tinit}$ implies $i < j$. □

Condition (f) ensures that each write has at most one commit. Conditions (g) and (h) manage the redundant information in traces. Conditions (i) and (j) constrains actions by the tinit thread: these may only be write and commit actions at the beginning of a trace; further, every initial write must be committed.

We assume that all traces are well-formed.

We now give an inductive characterization of traces using an automaton that maintains a separate stack for each thread. We ignore the action names annotating traces, which can be validated independently.

Definition A.2 (Inductive characterization of traces). Define *stack actions* as follows.

$$\zeta \ ::= \ \mathbf{A} \mid \mathbf{Q} \mid \ .f \mid \ \mathbf{!}f \mid \ ?f$$

Let μ range over maps from thread names to stacks of stack actions, and let *initmap* be the initial map, which maps all threads to an empty stack. In any trace it is not know whether a thread is initially active or quiescent until the first action of that thread is seen. We use the stack actions $\mathbf{A}$ and $\mathbf{Q}$ to indicate that initial state, once it is known. One of these markers always sits at the bottom of a nonempty stack; there is at most one marker, and it is never popped.

We access maps using the function $\mathit{push}(\mu, s, f) = \mu'$, the predicate $\mathit{empty}(\mu, s)$, and the partial functions $\mathit{pop}(\mu, s) = \mu'$ and $\mathit{top}(\mu, s) = \zeta$. These are defined the obvious way. Define the predicates *active* and *quiescent* as follows.

$$\mathit{active}(\mu, s) \triangleq \neg(\mathit{empty}(\mu, s)) \text{ and } (\mathit{top}(\mu, s) \in \{\mathbf{A}, .f, ?f\})$$
$$\mathit{quiescent}(\mu, s) \triangleq \neg(\mathit{empty}(\mu, s)) \text{ and } (\mathit{top}(\mu, s) \in \{\mathbf{Q}, \mathbf{!}f\})$$

For any pair (μ, s), exactly one of *empty*, *active* and *quiescent* holds.

Let σ range over sequences of actions. We define the partial function *accept* over these action sequences. The definition is by induction on σ, using a map to keep track of the trace processed thus far. If σ is a suffix of a trace, where the prefix is summarized by μ, then $\mathit{accept}(\sigma, \mu)$ returns the map that results from processing σ (continuing after μ). Otherwise, if σ is not the suffix of such a trace, then $\mathit{accept}(\sigma, \mu)$ is undefined. For the basis of the definition, we have

$$\mathit{accept}(\varepsilon, \mu) \triangleq \mu.$$

For the step, let $s = \mathit{thrd}(\alpha)$.

$\mathit{accept}(\alpha\sigma, \mu) \triangleq$

$$\begin{cases}
\mathit{accept}(\alpha\sigma, \mathit{push}(\mu, s, \mathsf{A})) & \text{if } \mathit{empty}(\mu, s) \text{ and } \alpha \in \mathit{Mem} \cup \{\langle \mathtt{.}\,\mathsf{call}\rangle, \langle \mathtt{!}\,\mathsf{call}\rangle\} \\
\mathit{accept}(\alpha\sigma, \mathit{push}(\mu, s, \mathsf{Q})) & \text{if } \mathit{empty}(\mu, s) \text{ and } \alpha \in \{\langle \mathtt{?}\mathsf{call}\rangle\} \\
\mathit{accept}(\sigma, \mu) & \text{if } \mathit{active}(\mu, s) \text{ and } \alpha \in \mathit{Mem} \\
\mathit{accept}(\sigma, \mathit{push}(\mu, s, \mathtt{!}f)) & \text{if } \mathit{active}(\mu, s) \text{ and } \alpha = \langle \mathtt{!}\,\mathsf{call}\ f\rangle \\
\mathit{accept}(\sigma, \mathit{push}(\mu, s, \mathtt{.}f)) & \text{if } \mathit{active}(\mu, s) \text{ and } \alpha = \langle \mathtt{.}\,\mathsf{call}\ f\rangle \\
\mathit{accept}(\sigma, \mathit{pop}(\mu, s)) & \text{if } \mathit{active}(\mu, s) \text{ and } \alpha = \langle \mathtt{!}\,\mathsf{ret}\ f\rangle \text{ and } \mathit{top}(\mu, s) = \mathtt{?}f \\
\mathit{accept}(\sigma, \mathit{pop}(\mu, s)) & \text{if } \mathit{active}(\mu, s) \text{ and } \alpha = \langle \mathtt{.}\,\mathsf{ret}\ f\rangle \text{ and } \mathit{top}(\mu, s) = \mathtt{.}f \\
\mathit{accept}(\sigma, \mathit{push}(\mu, s, \mathtt{?}f)) & \text{if } \mathit{quiescent}(\mu, s) \text{ and } \alpha = \langle \mathtt{?}\mathsf{call}\ f\rangle \\
\mathit{accept}(\sigma, \mathit{pop}(\mu, s)) & \text{if } \mathit{quiescent}(\mu, s) \text{ and } \alpha = \langle \mathtt{?}\mathsf{ret}\ f\rangle \text{ and } \mathit{top}(\mu, s) = \mathtt{!}f
\end{cases} \qquad \square$$

The rules fall into three categories, depending upon whether the stack for s is empty, or the top of the stack indicates that the thread is active, or quiescent.

It is not known whether the thread is initially active or quiescent until the first symbol is seen. At the point, the rules for the empty stack record the decision. The stack is never empty again. Note that the rules for the empty stack do not consume the token α. This is important, since the `!` actions are available to active threads, but make the thread quiescent, and symmetrically for `?` actions.

The rules for active and quiescent threads slavishly follow the grammar of Definition 4.1, which they implement.

Proposition A.3. $\mathit{accept}(\sigma, \mathit{initmap})$ is defined exactly when σ is a trace.

PROOF SKETCH. Suppose that σ is a trace; we show that $\mathit{accept}(\sigma)$ is defined. (The other direction is similar.) Let $\mathit{thrds}(\sigma) = \{s_1, \ldots, s_n\}$. Since σ is a trace, it must be an interleaving of $\sigma|_{s_1}, \ldots, \sigma|_{s_n}$. Further, it must be that each $\sigma|_{s_i}$ is a single-threaded balanced trace, as given by the grammar for B in Definition 4.1. For any single thread, *accept* is a push down automata accepting the grammar of B. □

B Properties of composition

Composition is defined in Section 9. In this section we provide basic machinery for dealing with composition. We give an alternative characterization of traces and composition, and define projection.

B.1 Composition

We sometimes write $\sigma \otimes \rho$ for $\{\sigma\} \otimes \{\rho\}$.

Proposition B.1. Composition is commutative and associative with identity $\{\varepsilon\}$.

PROOF SKETCH. Commutativity and identity are straightforward from Definition 9.3. We sketch associativity. Composition is defined only for sets with disjoint *intern*. For $i, j \in \{1, 2, 3\}$, let $\mathit{intern}(\Phi_i) = H_i$, such that for $i \neq j$, we have $H_i \cap H_j = \emptyset$. Consider an action sequence π such that π is a collapsed interleaving of some σ_1, σ_2 and σ_3, for $\sigma_i \in \Phi_i$, and π is a trace such that $\mathit{extern}(\pi) \cap (H_1 \cup H_2 \cup H_3) = \emptyset$. Because the *intern* sets are disjoint, the input actions of each trace have distinct function names. Thus, it does not matter if we collapse σ_1 and σ_2 first, or σ_2 and σ_3. Thus both $(\Phi_1 \otimes \Phi_2) \otimes \Phi_3$ and $\Phi_1 \otimes (\Phi_2 \otimes \Phi_3)$ give the same sets of action sequences. □

Definition B.2 (Inductive characterization of composition). Let $\mathit{intern}(\sigma) \subseteq H$ and $\mathit{intern}(\rho) \subseteq G$. If $H \cap G = \emptyset$, then define $\sigma\ {}_{H}{\otimes}_{G}\ \rho$ inductively as follows:

$$\frac{}{\varepsilon \in (\varepsilon\ {}_{H}{\otimes}_{G}\ \varepsilon)} \qquad \frac{\pi \in (\sigma\ {}_{H}{\otimes}_{G}\ \rho)}{\pi \in (\rho\ {}_{G}{\otimes}_{H}\ \sigma)} \qquad \frac{\pi \in (\sigma\ {}_{H}{\otimes}_{G}\ \rho)}{\alpha\pi \in ((\alpha\sigma)\ {}_{H}{\otimes}_{G}\ \rho)}\ \begin{array}{l} \mathit{thrd}(\alpha) \in H \\ \alpha \notin \{\langle ? \rangle, \langle ! \rangle\} \end{array}$$

$$\frac{\pi \in (\sigma\ {}_{H\cup\{s\}}{\otimes}_{G}\ \rho)}{\alpha'\pi \in ((\alpha\sigma)\ {}_{H}{\otimes}_{G}\ \rho)}\ \begin{array}{ll} s \notin H \cup G & B \subseteq \mathit{defs}!(\rho) \\ \alpha = \langle s?\, A \rangle & \alpha' = \alpha\{\![{}^{A\cup B}/_{A}]\!\} \end{array}$$

$$\frac{\pi \in (\sigma\ {}_{H\setminus\{s\}}{\otimes}_{G}\ \rho)}{\alpha\pi \in ((\alpha\sigma)\ {}_{H}{\otimes}_{G}\ \rho)}\ \begin{array}{l} s \in H \text{ and } f \notin G \\ \alpha = \langle s!f \rangle \end{array}$$

$$\frac{\pi \in (\sigma\ {}_{H\setminus\{s\}}{\otimes}_{G\cup\{s\}}\ \rho)}{\langle s\,.\,\mathsf{call}\ f\ \vec{u}\ a \rangle\pi \in ((\alpha\sigma)\ {}_{H}{\otimes}_{G}\ (\gamma\rho))}\ \begin{array}{l} s \in H \\ \alpha = \langle s!\mathsf{call}\ f\ \vec{u}\ a \rangle \\ \gamma = \langle s?\mathsf{call}\ f\ \vec{u}\ a \rangle \end{array}$$

$$\frac{\pi \in (\sigma\ {}_{H\setminus\{s\}}{\otimes}_{G\cup\{s\}}\ \rho)}{\langle s\,.\,\mathsf{ret}\ \vec{u}\ A\ B \rangle\pi \in ((\alpha\sigma)\ {}_{H}{\otimes}_{G}\ (\gamma\rho))}\ \begin{array}{l} s \in H \\ \alpha = \langle s!\mathsf{ret}\ \vec{u}\ A \rangle \\ \gamma = \langle s?\mathsf{ret}\ \vec{u}\ B \rangle \end{array}$$

□

Reading a derivation from bottom to top, the definition processes a trace from left to right, removing an action from the front of the trace at each step. Note that the functions in H and G remain fixed, only the threads change; these record the active threads on the current suffix of the initial trace. To see that this is necessary, consider the composition of ⟨s?f⟩⟨s!f⟩ and ⟨s?g⟩⟨s!g⟩. Composition allows ⟨s?f⟩⟨s!f⟩⟨s?g⟩⟨s!g⟩ but forbids ⟨s?f⟩⟨s?g⟩⟨s!f⟩⟨s!g⟩

Suppose $\sigma\ {}_{H}{\otimes}_{G}\ \rho$ is defined and $\sigma|_s$ is nonempty. By induction on the definition, one can show that if $s \in H$ then $\sigma|_s$ is an active trace; otherwise $\sigma|_s$ is quiescent. An invariant of the definition is that H and G are disjoint: it is not possible for both $\sigma|_s$ and $\rho|_s$ to be active, but it is possible for them to both be quiescent; in this case, the environment is active.

The rule for $\alpha = \langle s? \rangle$ indicates that a thread is activated by input. (Given the invariants, it is not possible that $s \in H$, since $\alpha\sigma$ is a quiescent trace.) Similarly, the rule for $\alpha = \langle s!f \rangle$ indicates that a thread is made quiescent by output. The last two rules describe the synchronization of input and output, which transfer the active thread from one side to the other.

We use the functions *make*! and *make*? in the discussion below.

Definition B.3. Define the functions *make*? and *make*! over actions. These functions are the identity function for any action but "." actions, in which case they replace the "." with either "?" or "!", eliding use sets as necessary. For example, *make*! $(\langle s\,.\,\mathsf{ret}\ \vec{u}\ A\ B \rangle) = \langle s!\mathsf{ret}\ \vec{u}\ A \rangle$ and $\mathit{make}?(\langle s\,.\,\mathsf{ret}\ \vec{u}\ A\ B \rangle) = \langle s?\mathsf{ret}\ \vec{u}\ B \rangle$. □

Definition B.4. We say that a trace σ is *properly initialized* if all of the actions from thread tinit are at the beginning. □

Proposition B.5. Let $\mathit{intern}(\Phi) = H$ *and* $\mathit{intern}(\Psi) = G$*. If* $H \cap G = \emptyset$*, then*

$$\Phi \otimes \Psi = \{\pi \mid \exists\sigma \in \Phi.\ \exists\rho \in \Psi.\ \pi \in (\sigma\ {}_H{\otimes}_G\ \rho) \text{ and } \pi \text{ is properly initialized}\}.$$

PROOF SKETCH. From right to left is a straightforward induction on the derivation of $\sigma\ {}_H{\otimes}_G\ \rho$, using the invariants outlined above.

We give the argument from left to right. By induction on π, we show that for all Φ and Ψ such that $\mathit{intern}(\Phi) = H$, $\mathit{intern}(\Psi) = G$, $H \cap G = \emptyset$, and $\pi \in \Phi \otimes \Psi$, we have that $\exists\sigma \in \Phi.\ \exists\rho \in \Psi.\ \pi \in (\sigma\ {}_H{\otimes}_G\ \rho)$.

In the case that $\pi = \varepsilon$, the result is immediate, taking $\sigma = \rho = \varepsilon$.

In the case that $\pi = \alpha\pi'$, then either (1) $\sigma = \alpha\sigma'$, (2) $\rho = \alpha\rho'$, (3) $\sigma = \mathit{make}!(\alpha)\sigma'$ and $\rho = \mathit{make}?(\alpha)\rho'$, or (4) $\sigma = \mathit{make}?(\alpha)\sigma'$ and $\rho = \mathit{make}!(\alpha)\rho'$. In any case, the proof obligation for π' will follow by induction.

(1a) Suppose α is not an I/O action. Then $\alpha\sigma'$ is active for $\mathit{thrd}(\alpha)$ and therefore $\mathit{thrd}(\alpha) \in H$. Thus, the rule for $\alpha \notin \{\langle ?\rangle, \langle !\rangle\}$ applies.

(1b) Suppose α is an input action. Then π is quiescent for $\mathit{thrd}(\alpha)$ and therefore, by the *extern* clause of Definition 9.3, $\mathit{thrd}(\alpha) \notin (H \cup G)$. Thus, the rule for $\alpha = \langle s?\rangle$ applies.

(1c) Suppose α is an output action. Then π is an output for f and therefore, by the *extern* clause of Definition 9.3, $f \notin (H \cup G)$. Thus, the rule for $\alpha = \langle s!f\rangle$ applies.

(3) In this case one of the final two rules applies.

Using the rule for symmetry, (2) follows from (1) and (4) follows from (3). □

B.2 Projection

Composition also supports a notion of *component projection*, or decomposition.

Example B.6. The definition can be understood from the following examples.

$$\langle \mathsf{s.call\ f}\rangle\langle \mathsf{s.ret}\rangle{\downarrow_{\{\mathsf{s}\}}} = \langle \mathsf{s!call\ f}\rangle\langle \mathsf{s?ret}\rangle \qquad \langle \mathsf{s.call\ f}\rangle\langle \mathsf{s.ret}\rangle{\downarrow_{\{\mathsf{s},\mathsf{f}\}}} = \langle \mathsf{s.call\ f}\rangle\langle \mathsf{s.ret}\rangle$$

$$\langle \mathsf{s.call\ f}\rangle\langle \mathsf{s.ret}\rangle{\downarrow_{\{\mathsf{f}\}}} = \langle \mathsf{s?call\ f}\rangle\langle \mathsf{s!ret}\rangle \qquad \langle \mathsf{s.call\ f}\rangle\langle \mathsf{s.ret}\rangle{\downarrow_{\emptyset}} = \varepsilon$$

□

Recall the definition of collapsed interleaving from Definition 9.2.

Definition B.7 (Projection). When $\mathit{extern}(\pi) \cap H = \emptyset$, define $\pi{\downarrow_H} = \sigma$ such that for some ρ, we have that π is a collapsed interleaving of σ and ρ, and $\mathit{extern}(\sigma) \cap H = \mathit{intern}(\rho) \cap H = \emptyset$. We lift the definition to sets of traces via union. □

We write $\pi{\downarrow_\Psi}$ as shorthand for $\pi{\downarrow_{\mathit{intern}(\Psi)}}$, and similarly $\Phi{\downarrow_\Psi}$ for $\Phi{\downarrow_{\mathit{intern}(\Psi)}}$.

Proposition B.8. (a) When $\mathit{extern}(\pi) \cap H = \emptyset$*,* $\pi{\downarrow_H}$ *is unique.*

(b) If $\mathit{intern}(\Phi) \cap \mathit{intern}(\Psi) = \emptyset$ *then* $(\Phi \otimes \Psi){\downarrow_\Phi} \subseteq \Phi$*.*

(c) If $\mathit{intern}(\Phi) = H \cup G$ *and* $H \cap G = \emptyset$ *then* $(\Phi{\downarrow_H}) \otimes (\Phi{\downarrow_G}) = \Phi$*.*

(d) $\Phi{\downarrow_\Phi} = \Phi$*.*

PROOF SKETCH. (a)–(c) are immediate from the definitions.

For (d), the direction $\Phi{\downarrow_{\Phi}} \supseteq \Phi$, follows by taking $\sigma = \pi \in \Phi$ and $\rho = \varepsilon$. For the reverse, we show that any ρ meeting the requirements must be ε.

Suppose that ρ is active and that its initial action is by thread s. Then $s \in \mathit{intern}$ (ρ) and therefore $s \in \mathit{intern}(\pi)$ and $s \in \mathit{intern}(\Phi)$. (Recall that a collapsed interleaving includes all the action of ρ, except I/O actions. I/O actions may be converted to "`.`" actions; nevertheless, "`.`" actions are also *intern*.) But this contradicts the requirement that $\mathit{intern}(\rho) \cap H = \emptyset$, since, in this case, $H = \mathit{intern}(\Phi)$.

Suppose that ρ is quiescent and that its initial action is an input call on function f. Then $f \in \mathit{intern}(\rho)$ and, by reasoning as above, $f \in \mathit{intern}(\Phi)$, again leading to a contradiction. □

Definition B.9 (Inductive characterization of projection). Recall the maps from Definition A.2 and define *actor* as follows.

$$\mathit{actor}(\mu, s) = \begin{cases} s, & \text{if } \mathit{empty}(\mu, s) \text{ or } \mathit{top}(\mu, s) \text{ is } \mathsf{A} \text{ or } \mathsf{Q} \\ f, & \text{if } \mathit{top}(\mu, s) \text{ is } \texttt{.}f,\ \texttt{?}f, \text{ or } \texttt{!}f \end{cases}$$

For the basis of the definition, we have the following.

$$\varepsilon{\downarrow}_H^{\mu} \triangleq \varepsilon$$

For the inductive step, we do a case analysis. Define

$$(\alpha\pi){\downarrow}_H^{\mu} \triangleq \rho\,(\pi{\downarrow}_H^{\mathit{accept}(\alpha,\mu)})$$

where $\eta = \mathit{actor}(\mu, \mathit{thrd}(\alpha))$ and

$$\rho = \begin{cases} \mathit{make}\texttt{!}(\alpha) & \text{if } \eta \in H \text{ and } \alpha = \langle \texttt{.}f \rangle \text{ and } f \notin H \\ \alpha & \text{if } \eta \in H \text{ otherwise} \\ \mathit{make}\texttt{?}(\alpha) & \text{if } \eta \notin H \text{ and } \alpha = \langle \texttt{.}f \rangle \text{ and } f \in H \\ \alpha & \text{if } \eta \notin H \text{ and } \alpha = \langle \texttt{?}f \rangle \text{ and } f \in H \\ \varepsilon & \text{if } \eta \notin H \text{ otherwise.} \end{cases}$$

□

The map μ is updated as each token is processed, using the *accept* function from Definition A.2.

The "actor" of a thread is the function on top of the call stack, if any, and the thread itself otherwise. In the inductive step, there are two groups of rules, depending upon whether the actor is in H or not.

A "`.`" action is converted to a "`!`" if the actor is in H, but the function is not. A "`.`" action is converted to a "?" if the actor is in not in H, but the function is. Other dot actions are processed like any other action: If the actor is in H, then all of its actions are included in the projection. If the actor is not, then only the "?" actions serviced by H are included. Such "?" actions transfer control to the component; all other actions belong to the environment.

Proposition B.10. $\Phi{\downarrow_H} = \bigcup_{\pi\in\Phi} \pi{\downarrow_H^{\mathit{initmap}}}$.

PROOF SKETCH. The result follows from: $\forall \pi \in \Phi.\ \{\pi\}{\downarrow_H} = \{\pi{\downarrow_H^{\mathit{initmap}}}\}$. In both directions, this follows by induction on the length of π, where μ records the history of the trace seen thus far. From right to left, the ρ required by Definition B.7 can be constructed by modifying the definition of ${\downarrow_H^{\mu}}$ to generate both traces. □

B.3 Order preserving projection

A subtrace derived by projection may not include all the order of the original trace.

Example B.11. In

$$\langle \mathsf{s\ wr\ x}\rangle\langle \mathsf{s\,.\,call\ f}\rangle\langle \mathsf{s}\ \underline{\mathsf{wr}}\ \mathsf{w}\rangle\langle \mathsf{t\,.\,call\ g}\rangle\langle \mathsf{t}\ \underline{\mathsf{rd}}\ \mathsf{w}\rangle\langle \mathsf{t\,.\,ret}\rangle\langle \mathsf{t\ rd\ x}\rangle,$$

we have $\langle \mathsf{s\ wr\ x}\rangle <_{\mathsf{hb}} \langle \mathsf{t\ rd\ x}\rangle$. However, when projecting this thread to $\{\mathsf{s}, \mathsf{t}\}$, we have

$$\langle \mathsf{s\ wr\ x}\rangle\langle \mathsf{s\,!\,call\ f}\rangle\langle \mathsf{t\,!\,call\ g}\rangle\langle \mathsf{t?ret}\rangle\langle \mathsf{t\ rd\ x}\rangle,$$

which lacks any order between $\langle \mathsf{s\ wr\ x}\rangle$ and $\langle \mathsf{t\ rd\ x}\rangle$.

If we "saturate" the original trace, we arrive at

$$\langle \mathsf{s\ wr\ x}\rangle\langle \mathsf{s\,.\,call\ f\ a}\rangle\langle \mathsf{s}\ \underline{\mathsf{wr}}\ \mathsf{w}\rangle\langle \mathsf{t\,.\,call\ g\ b}\rangle\langle \mathsf{t}\ \underline{\mathsf{rd}}\ \mathsf{w}\rangle\langle \mathsf{t\,.\,ret}\ \emptyset\ \{\mathsf{a}\}\rangle\langle \mathsf{t\ rd\ x}\rangle.$$

In this case, the projection,

$$\langle \mathsf{s\ wr\ x}\rangle\langle \mathsf{s\,!\,call\ f\ a}\rangle\langle \mathsf{t\,!\,call\ g\ b}\rangle\langle \mathsf{t?ret}\ \{\mathsf{a}\}\rangle\langle \mathsf{t\ rd\ x}\rangle,$$

includes the desired order. □

We identify a class of traces which do have this property.

Definition B.12 (Saturated). Let $\prec$ be a trace-indexed family of relations as in Definition 7.2. We say that $\pi = \gamma_1 \cdots \gamma_n$ is *$\prec$-saturated* if $\forall i, j \in [1, n].\ \gamma_i = \langle\,.\,\mathsf{call}\ a\rangle$, $\gamma_j = \langle\,.\,\mathsf{ret}\ A\ B\rangle$, $i \prec^{\pi} j$ implies $a \in B$. □

Definition B.13 (Projection with map). Suppose $\pi = \gamma_1 \cdots \gamma_n$, $\sigma = \alpha_1 \cdots \alpha_m$ and $\delta : [1, n] \to [1, m]$ is a monotone[3] injective partial function. We write $\sigma = \pi{\downarrow_H^{\delta}}$ to indicate that $\{\sigma\} = \{\pi\}{\downarrow_H}$ and $\forall i \in \mathit{dom}(\delta).\ \alpha_{\delta(i)} \in \{\gamma_i, \mathit{make}?(\gamma_i), \mathit{make}\,!\,(\gamma_i)\}$. □

Lemma B.14. Suppose π is $\prec$-saturated and $\sigma = \pi{\downarrow_H^{\delta}}$. Then $\forall i, j \in \mathit{dom}(\delta).\ i \prec^{\pi} j$ iff $\delta(i) \prec^{\sigma} \delta(j)$.

PROOF SKETCH. Immediate. □

[3] A partial function δ is monotone if $\forall i \in \mathit{dom}(\delta).\ i < j$ implies $\delta(i) < \delta(j)$.

Definition B.15 (Saturate/unsaturate). We say that a trace $\pi = \gamma_1 \cdots \gamma_n$ is *unsaturated* if $\forall i \in [1, n].\ \gamma_i = \langle\, .\, \mathsf{ret}\, A\, B \rangle$ implies $A = B = \emptyset$.

For any $\pi = \gamma_1 \cdots \gamma_n$, define $\mathit{saturate}_{\prec}(\pi) = \gamma'_1 \cdots \gamma'_n$ to be the $\prec$-saturated trace with the smallest rely sets such that $\forall i \in [1, n].\ \gamma'_i$ is either the same as γ_i or is derived from $\gamma_i = \langle\, .\, \mathsf{ret}\, A\, B \rangle$ by replacing B by $B' \supseteq B$. (This is unique up to renaming of actions.)

For any $\pi = \gamma_1 \cdots \gamma_n$, define $\mathit{unsaturate}(\pi) = \gamma'_1 \cdots \gamma'_n$, where $\forall i \in [1, n].\ \gamma'_i$ is either the same as γ_i or is derived from $\gamma_i = \langle\, .\, \mathsf{ret}\, A\, B \rangle$ by replacing B by $\emptyset$. □

For a memory model $\mathscr{W}$, we write $\mathit{saturate}_{\mathscr{W}}$ for $\mathit{saturate}_{<_{\mathscr{W}}}$.

Lemma B.16. Let $\pi' = \mathit{saturate}_{\prec}(\pi)$. Then $i \prec^{\pi'} j$ iff $i \prec^{\pi} j$.

PROOF SKETCH. Immediate. □

Lemma B.17. If π is unsaturated, then $\pi = \mathit{unsaturate}(\mathit{saturate}_{\prec}(\pi))$.

PROOF SKETCH. Immediate. □

C Operational models

In this section we define operational models that generate trace sets appropriate for seq, tso, pso and hb. We present the first three models in their familiar form, using a store and thread-specific buffers that map variables to values. hb is distinctly less operational in flavor and requires a different treatment to capture the examples of interest. We have chosen a variant of the JMM that we developed in prior work [Jagadeesan, Pitcher, and Riely 2010]; unlike the other three semantics, the JMM does not have a traditional store.

Much of the machinery has little to do with the memory model, and we present this first, using the processes appropriate for seq, tso, pso. We divide the presentation into three subsections. We give the general framework in Section C.2, the I/O rules in Section C.3 and the memory rules in Section C.4. We adapt the definitions to the JMM in Section C.5.

C.1 Syntax

We present the simplest language that can encode the examples of interest. It is a while language with integers as the only values. For simplicity, we do not include binders for variables; they have global scope. When necessary, the effect of scoping is achieved by restricting attention to components with disjoint variables.

Expressions and *evaluation contexts* are defined as follows, where $op \in Op$ is an *operator*, whose semantics is given by a function $[\![op]\!]$ from value tuples to values. The forms that are shaded in the syntax for expression are not allowed in (static) components; they are allowed only in (dynamic) processes, defined below.

$$\begin{aligned}
C, D, E \;&::= u \mid z \mid \colorbox{lightgray}{$\mathsf{test}\; f$} \mid \colorbox{lightgray}{$f\,\texttt{.}\{C\}$} \mid \colorbox{lightgray}{$f\,\texttt{!}\{C\}$} \\
&\mid op(E) \mid f(E) \mid E_1, \ldots, E_n \mid \mathsf{let}\; z_1, \ldots, z_n = E; C \\
&\mid C; D \mid \mathsf{if}\; E\; \mathsf{then}\; C\; \mathsf{else}\; D \mid \mathsf{while}\; E\; \mathsf{do}\; C \mid \mathsf{return}\; E \\
&\mid x \mid w \mid x\texttt{=}E \mid w\texttt{=}E \mid w\,\texttt{.}\mathsf{cas}(E, D) \\
\mathbb{C}, \mathbb{D}, \mathbb{E} \;&::= [-] \mid f\,\texttt{.}\{\mathbb{C}\} \mid f\,\texttt{!}\{\mathbb{C}\} \\
&\mid op(\mathbb{E}) \mid f(\mathbb{E}) \mid \vec{u}, \mathbb{E}, \vec{E} \mid \mathsf{let}\; \vec{z} = \mathbb{E}; C \\
&\mid \mathbb{C}; D \mid \mathsf{if}\; \mathbb{E}\; \mathsf{then}\; C\; \mathsf{else}\; D \mid \mathsf{return}\; \mathbb{E} \\
&\mid x\texttt{=}\mathbb{E} \mid w\texttt{=}\mathbb{E} \mid w\,\texttt{.}\mathsf{cas}(\mathbb{E}, D) \mid w\,\texttt{.}\mathsf{cas}(u, \mathbb{E})
\end{aligned}$$

We use infix notation for operators when customary. Let "skip" be syntax sugar for the value 0, let "do C until E" be syntax sugar for "C;while not(E) do C", and let "x++" be syntax sugar for "x=x+1". A function without a return will get stuck; in addition, a return will be stuck unless it is followed by an unreachable expression. In examples, we elide returns with uninteresting results and write return E rather than return E;skip.

The notable design choices deal with multiple return values and the treatment of parameters and local variables.

Some examples require multiple return values; to accomodate this, we include simple tuple constructors and destructors. Tuples cannot be stored in memory, nor bound to local variables. The language is untyped, and therefore execution may become stuck if there is a arity mismatch.

Parameters and local variables are given a semantics via substitution. We treat local variables as "static single assignment"; local variables bound within a loop are reassigned on each iteration. The alternative solution, which carries local variables as mutable state, is notationally burdensome. The substitution semantics requires a few tricks so that there is no loss of expressiveness. First, in the dynamics, while expands to an if, creating two copies of the loop code before the condition is evaluated. Second, we allow a return to terminate the execution of a loop, potentially returning values from local variables. Our treatment of return is chosen for expedience rather than elegance, as will become clear below.

C.2 Memory-independent semantics for seq, tso and pso

When a component is loaded, the result is a *process*, which includes the variables and threads of each component in the program.

The exact syntax of processes varies depending upon the memory model. Functions are immutable; variables and threads are mutable. For uniformity, we define the rules for seq, tso and pso to include a buffer with each thread. In the case of seq, the buffer will remain empty. tso and pso have different rules for removing items from the buffer.

Definition C.1. The syntax of buffers is as follows.

$$\beta \ ::= x_1{=}u_1@a_1, \ldots, x_n{=}u_n@a_n$$

We write the empty buffer as $\emptyset$. Buffer lookup is a partial function from data variables to values, written $\beta(x)$. For the empty buffer, $\emptyset(x)$ is undefined for all x. Otherwise,

$$(\beta, y{=}u@a)(x) \ \triangleq \ \begin{cases} u@a & \text{if } x = y \\ \beta(x) & \text{otherwise.} \end{cases} \qquad \square$$

Processes and process contexts are defined as follows.

$$\begin{aligned} P, Q \ &::= \mathsf{var}\, x{=}u@a \ \mid \ \mathsf{atomic}\, w{=}u \ \mid \ \mathsf{thrd}\, s\, \beta\, C \ \mid \ P \mid Q \\ \mathbb{P} \ &::= [-] \ \mid \ \mathbb{P} \mid Q \ \mid \ Q \mid \mathbb{P} \end{aligned}$$

In defining evaluation, we separate elements which are under a component's control from those which are not. Memory and threads are stored in the process, the remaining bookkeeping data is stored in an *environment*. As a process evolves, so does its environment.

The environment keeps track of function definitions (stored in the component M) and the set A of action names that have been previously used to annotate actions. For use by tso and pso, it also keeps track of the state of the process buffers when calls are made to the environment. This allows us to ensure that certain buffered writes have been committed before a call to the environment returns.

To formalize this, let ζ range over partial maps over $Act \to Thrd \times 2^{Act}$. We write the empty map as $\emptyset$ and map extension as $\zeta, a : (s, B)$.

Environments have the following form.

$$\Delta, \Gamma \ ::= \ (M, A, \zeta)$$

We now define several notations using environments. First, we define the functions *lbls*, *top* and *visible*. For buffers, define $\mathit{lbls}(\beta)$ as follows.

$$\mathit{lbls}(x_1{=}u_1\mathtt{@}a_1,\ \ldots,\ x_n{=}u_n\mathtt{@}a_n) \triangleq \{a_1,\ \ldots,\ a_n\}$$

For evaluation contexts, define $\mathit{top}(\eta, \mathbb{C})$ as follows.

$$\begin{aligned}
\mathit{top}(\eta, [\text{–}]) &\triangleq \eta \\
\mathit{top}(\eta, f\,\mathtt{.\{}\mathbb{C}\mathtt{\}}) &\triangleq \mathit{top}(f, \mathbb{C}) \\
\mathit{top}(\eta, f\,\mathtt{!\{}\mathbb{C}\mathtt{\}}) &\triangleq \mathit{top}(f, \mathbb{C}) \\
\mathit{top}(\eta, \ldots\mathbb{C}\ldots) &\triangleq \mathit{top}(\eta, \mathbb{C}), \qquad \text{otherwise}
\end{aligned}$$

For components, define $\mathit{visible}(\eta, M)$ as follows.

$$\begin{aligned}
\mathit{visible}(\eta, M) &\triangleq \mathit{funs}(M) && \text{if } M \text{ is a base component} \\
\mathit{visible}(\eta, M \parallel N) &\triangleq \mathit{visible}(\eta, M) \cup \mathit{visible}(\eta, N) && \\
\mathit{visible}(\eta, M \setminus f) &\triangleq \mathit{visible}(\eta, M) && \text{if } \eta \in \mathit{thrds}(M) \cup \mathit{funs}(M) \\
\mathit{visible}(\eta, M \setminus f) &\triangleq \mathit{visible}(\eta, M) \setminus \{f\} && \text{if } \eta \notin \mathit{thrds}(M) \cup \mathit{funs}(M)
\end{aligned}$$

We define the following notation for environments, where $\Delta = (M, A, \zeta)$.

$$\begin{aligned}
\mathit{funs}(\Delta) &\triangleq \mathit{funs}(M) \\
\Delta, a &\triangleq (M, A \cup \{a\}, \zeta) && \text{if } a \notin A \\
\Delta, a : (s, \beta) &\triangleq (M, A \cup \{a\}, (\zeta, a : (s, \mathit{lbls}(\beta)))) && \text{if } a \notin A \\
\mathit{thrds}(\Delta, A) &\triangleq \{s \mid \exists a \in A.\ \zeta(a) = (s, \_)\} \\
\mathit{lbls}(\Delta, A) &\triangleq \{b \mid \exists a \in A.\ \zeta(a) = (\_, B) \text{ and } b \in B\} \\
\Delta(f, s, \mathbb{C}) &\triangleq \Lambda \qquad \text{if } (\mathsf{fun}\ f\ \Lambda) \in M \text{ and } f \in \mathit{visible}(\mathit{top}(s, \mathbb{C}), M)
\end{aligned}$$

The partial function Δ, a adds a to the action name set of Δ; it is undefined if a already occurs in the set. $\Delta, a : (s, \beta)$ does the same, but additionally adds information to ζ, which is then recoverable using *thrds* and *lbls*. The partial function $\Delta(f, s, \mathbb{C})$ returns the abstraction for f, if it is defined by M and visible by $\mathit{top}(s, \mathbb{C})$.

We define a trivial structural equivalence over processes as the least equivalence relation satisfying the following rules.

$$\begin{gathered}
P \mid P' \equiv P' \mid P \\
P \mid (P' \mid P'') \equiv (P \mid P') \mid P''
\end{gathered}$$

One can easily show that structural equivalence is a congruence for processes; that is, $P \equiv P'$ implies $P \mid Q \equiv P' \mid Q$ and $Q \mid P \equiv Q \mid P'$.

For each $\mathscr{W}$, we define an evaluation relation, written $P \xrightarrow[\Delta/\Delta']{\alpha}_{\mathscr{W}} P'$, indicating that under model $\mathscr{W}$, process P in environment Δ may evolve to process P' in environment Δ' by executing α. Evaluation is defined to be the least relation that satisfies the axioms of Figures 1–3 and is preserved by structural equivalence. Formally, the structural rule is as follows.

$$P \xrightarrow[\Delta/\Delta']{\alpha}_{\mathscr{W}} P' \quad \text{if } P \equiv Q \xrightarrow[\Delta/\Delta']{\alpha}_{\mathscr{W}} Q' \equiv P'$$

$$
\begin{array}{rcll}
\mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[\mathsf{while}\ E\ \mathsf{do}\ C]\big] & \xrightarrow{\varepsilon}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[\mathsf{if}\ E\ \mathsf{then}\ (C;\mathsf{while}\ E\ \mathsf{do}\ C)\ \mathsf{else}\ 0]\big] & \\
\mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[\mathsf{if}\ u\ \mathsf{then}\ C\ \mathsf{else}\ D]\big] & \xrightarrow{\varepsilon}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[D]\big] & \text{if } u=0 \\
\mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[\mathsf{if}\ u\ \mathsf{then}\ C\ \mathsf{else}\ D]\big] & \xrightarrow{\varepsilon}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[C]\big] & \text{if } u\neq 0 \\
\mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[\mathsf{let}\ \vec{z}\mathsf{=}\vec{u};D]\big] & \xrightarrow{\varepsilon}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[D\{\!\![\vec{u}/\vec{z}]\!\!\}]\big] & \\
\mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[u;D]\big] & \xrightarrow{\varepsilon}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[D]\big] & \\
\mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[op(\vec{u})]\big] & \xrightarrow{\varepsilon}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ \_\,\_\ \mathbb{C}[v]\big] & \text{if } [\![op]\!](\vec{u})=v \\
\mathbb{P}\big[\mathsf{thrd}\ s\ \_\ \mathbb{C}[f(\vec{u})]\big] & \xrightarrow[\Delta/\Delta,a]{\langle s\,.\,\mathsf{call}\ f\ \vec{u}\ a\ \emptyset\rangle}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \_\ \mathbb{C}[f\,.\,\{D\{\!\![\vec{u}/\vec{z}]\!\!\}\}]\big] & \text{if } \Delta(f,s,\mathbb{C})=(\vec{z})\{D\} \\
\mathbb{P}\big[\mathsf{thrd}\ s\ \_\ \mathbb{C}[f\,.\,\{\mathsf{return}\ \vec{u};D\}]\big] & \xrightarrow[\Delta/\Delta,a]{\langle s\,.\,\mathsf{ret}\ f\ \vec{u}\ a\ \emptyset\rangle}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \_\ \mathbb{C}[\vec{u}]\big] &
\end{array}
$$

Fig. 1: Generic rules ($\mathscr{W} \in \{\mathsf{seq}, \mathsf{tso}, \mathsf{pso}, \mathsf{hb}\}$)

$$
\begin{array}{rcl}
\mathbb{P}\big[\mathsf{thrd}\ s\ \beta\ \mathbb{C}[f(\vec{u})]\big] & \xrightarrow[\Delta/\Delta,a:(s,\beta)]{\langle s\,!\mathsf{call}\ f\ \vec{u}\ a\ \emptyset\rangle}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \beta\ \mathbb{C}[\mathsf{test}\ f]\big] \\
 & & \text{if } f\notin \mathit{funs}(\Delta) \\
\mathbb{P}\big[\mathsf{thrd}\ \vec{s}\ \vec{\beta}\ \vec{C}\ \mathsf{|}\ \mathsf{thrd}\ s\ \beta\ \mathbb{C}[\mathsf{test}\ f]\big] & \xrightarrow[\Delta/\Delta,a]{\langle s?\mathsf{ret}\ f\ \vec{u}\ a\ B\rangle}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \beta\ \mathbb{C}[\vec{u}]\ \mathsf{|}\ \mathsf{thrd}\ \vec{s}\ \vec{\beta}\ \vec{C}\big] \\
 & & \text{if } \{\vec{s},s\}\supseteq \mathit{thrds}(\Delta,B) \text{ and } \mathit{lbls}(\Delta,B)\cap \mathit{lbls}(\vec{\beta},\beta)=\emptyset \\
\mathbb{P}\big[\mathsf{thrd}\ \vec{s}\ \vec{\beta}\ \vec{C}\ \mathsf{|}\ \mathsf{thrd}\ s\ \beta\ \mathbb{C}[\mathsf{test}\ g]\big] & \xrightarrow[\Delta/\Delta,a]{\langle s?\mathsf{call}\ f\ \vec{u}\ a\ B\rangle}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \beta\ \mathbb{C}[f\,!\{D\{\!\![\vec{u}/\vec{z}]\!\!\}\};\mathsf{test}\ g]\big] \\
 & & \text{if } \{\vec{s},s\}\supseteq \mathit{thrds}(\Delta,B) \text{ and } \mathit{lbls}(\Delta,B)\cap \mathit{lbls}(\vec{\beta},\beta)=\emptyset \\
 & & \text{and } \Delta(f,s,\mathbb{C})=(\vec{z})\{D\} \\
\mathbb{P}\big[\mathsf{thrd}\ s\ \beta\ \mathbb{C}[f\,!\{\mathsf{return}\ \vec{u};D\}]\big] & \xrightarrow[\Delta/\Delta,a:(s,\beta)]{\langle s\,!\mathsf{ret}\ f\ \vec{u}\ a\ \emptyset\rangle}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \beta\ \mathbb{C}[\vec{u}]\big]
\end{array}
$$

Fig. 2: I/O rules ($\mathscr{W} \in \{\mathsf{seq}, \mathsf{tso}, \mathsf{pso}\}$)

$$
\begin{array}{rcll}
\mathbb{P}\big[\mathsf{atomic}\ w\mathsf{=}u\ \mathsf{|}\ \mathsf{thrd}\ s\ \emptyset\ \mathbb{C}[w\mathsf{=}v]\big] & \xrightarrow{\langle s\ \underline{\mathsf{wr}}\ w\rangle}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \emptyset\ \mathbb{C}[v]\ \mathsf{|}\ \mathsf{atomic}\ w\mathsf{=}v\big] & \\
\mathbb{P}\big[\mathsf{atomic}\ w\mathsf{=}u\ \mathsf{|}\ \mathsf{thrd}\ s\ \_\ \mathbb{C}[w]\big] & \xrightarrow{\langle s\ \underline{\mathsf{rd}}\ w\rangle}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \_\ \mathbb{C}[u]\ \mathsf{|}\ \mathsf{atomic}\ w\mathsf{=}u\big] & \\
\mathbb{P}\big[\mathsf{atomic}\ w\mathsf{=}u\ \mathsf{|}\ \mathsf{thrd}\ s\ \_\ \mathbb{C}[w\,.\,\mathsf{cas}(u',v)]\big] & \xrightarrow{\langle s\ \underline{\mathsf{rd}}\ w\rangle}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \_\ \mathbb{C}[0]\ \mathsf{|}\ \mathsf{atomic}\ w\mathsf{=}u\big] & \text{if } u\neq u' \\
\mathbb{P}\big[\mathsf{atomic}\ w\mathsf{=}u\ \mathsf{|}\ \mathsf{thrd}\ s\ \emptyset\ \mathbb{C}[w\,.\,\mathsf{cas}(u',v)]\big] & \xrightarrow{\langle s\ \underline{\mathsf{cas}}\ w\rangle}_{\mathscr{W}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \emptyset\ \mathbb{C}[1]\ \mathsf{|}\ \mathsf{atomic}\ w\mathsf{=}v\big] & \text{if } u=u'
\end{array}
$$

$$
\begin{array}{rcl}
\mathbb{P}\big[\mathsf{var}\ x\mathsf{=}u\mathsf{@}a\ \mathsf{|}\ \mathsf{thrd}\ s\ \_\ \mathbb{C}[x\mathsf{=}v]\big] & \xrightarrow[\Delta/\Delta,b]{\langle s\ \mathsf{wr}\ x\ v\ b\rangle}_{\mathsf{seq}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \_\ \mathbb{C}[v]\ \mathsf{|}\ \mathsf{var}\ x\mathsf{=}v\mathsf{@}b\big] \\
\mathbb{P}\big[\mathsf{var}\ x\mathsf{=}u\mathsf{@}a\ \mathsf{|}\ \mathsf{thrd}\ s\ \_\ \mathbb{C}[x]\big] & \xrightarrow{\langle s\ \mathsf{rd}\ x\ u\ a\rangle}_{\mathsf{seq}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \_\ \mathbb{C}[u]\ \mathsf{|}\ \mathsf{var}\ x\mathsf{=}u\mathsf{@}a\big]
\end{array}
$$

$$
\begin{array}{rcll}
\mathbb{P}\big[\mathsf{thrd}\ s\ \beta\ \mathbb{C}[x\mathsf{=}v]\big] & \xrightarrow[\Delta/\Delta,a]{\langle s\ \mathsf{wr}\ x\ v\ a\rangle}_{\mathsf{tso,pso}} & \mathbb{P}\big[\mathsf{thrd}\ s\ (\beta, x\mathsf{=}v\mathsf{@}a)\ \mathbb{C}[v]\big] & \\
\mathbb{P}\big[\mathsf{thrd}\ s\ \beta\ \mathbb{C}[x]\big] & \xrightarrow{\langle s\ \mathsf{rd}\ x\ u\ a\rangle}_{\mathsf{tso,pso}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \beta\ \mathbb{C}[u]\big] & \text{if } \beta(x)=u\mathsf{@}a \\
\mathbb{P}\big[\mathsf{var}\ x\mathsf{=}u\mathsf{@}a\ \mathsf{|}\ \mathsf{thrd}\ s\ \beta\ \mathbb{C}[x]\big] & \xrightarrow{\langle s\ \mathsf{rd}\ x\ u\ a\rangle}_{\mathsf{tso,pso}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \beta\ \mathbb{C}[u]\ \mathsf{|}\ \mathsf{var}\ x\mathsf{=}u\mathsf{@}a\big] & \text{if } \beta(x) \text{ undefined} \\
\mathbb{P}\big[\mathsf{var}\ x\mathsf{=}u\mathsf{@}a\ \mathsf{|}\ \mathsf{thrd}\ s\ (x\mathsf{=}v\mathsf{@}b, \beta)\ C\big] & \xrightarrow{\langle \mathsf{com}\ s\ x\ b\rangle}_{\mathsf{tso}} & \mathbb{P}\big[\mathsf{thrd}\ s\ \beta\ C\ \mathsf{|}\ \mathsf{var}\ x\mathsf{=}v\mathsf{@}b\big] & \\
\mathbb{P}\big[\mathsf{var}\ x\mathsf{=}u\mathsf{@}a\ \mathsf{|}\ \mathsf{thrd}\ s\ (\beta, x\mathsf{=}v\mathsf{@}b, \beta')\ C\big] & \xrightarrow{\langle \mathsf{com}\ s\ x\ b\rangle}_{\mathsf{pso}} & \mathbb{P}\big[\mathsf{thrd}\ s\ (\beta, \beta')\ C\ \mathsf{|}\ \mathsf{var}\ x\mathsf{=}v\mathsf{@}b\big] & \text{if } x\notin \mathit{dom}(\beta)
\end{array}
$$

Fig. 3: SC/TSO/PSO memory rules ($\mathscr{W} \in \{\mathsf{seq}, \mathsf{tso}, \mathsf{pso}\}$)

To avoid clutter, we elide information which is simply carried from the left side of evaluation to the right: we elide the subscript Δ/Δ from the arrow and write _ for thread names and buffers when they are unused.

We discuss Figures 2–3 in the following subsections. The axioms in Figure 1 are straightforward. Only the last two rules may be worthy of comment: we include labels on internal actions in order to simplify the definition decomposition. A return statement throws away the remainder of a frame. For example, $f\,.\,\{\mathsf{return}\ u; D\}$ reduces to u. As noted above, in examples we write $f\,.\,\{\mathsf{return}\ E\}$ rather than $f\,.\,\{\mathsf{return}\ E; \mathsf{skip}\}$. The return statement is also the *only* way to discard a frame. Every terminating control flow path in a function body should end in a return statement; otherwise execution will become stuck.

Define multistep evaluation as usual.

$$\begin{aligned}
&P \xRightarrow[\Delta/\Delta]{\varepsilon}_{\mathscr{W}} P \\
&P \xRightarrow[\Delta/\Delta'']{\alpha\sigma}_{\mathscr{W}} P'' \quad \text{if } P \xrightarrow[\Delta/\Delta']{\alpha}_{\mathscr{W}} P' \xRightarrow[\Delta'/\Delta'']{\sigma}_{\mathscr{W}} P''
\end{aligned}$$

We write $P \xRightarrow[\Delta]{\sigma}_{\mathscr{W}}$ when $P \xRightarrow[\Delta/\Delta']{\sigma}_{\mathscr{W}} P'$ for some Δ' and P'.

The *initial client* for threads S is a process containing the given threads in parallel, each initialized with an empty buffer and running expression $\mathsf{test\ main}$, where main is a reserved function name (that is, main is not defined by any component). The *initial process* of a component contains the variable and thread declarations for that component, where all variables are initialized to their given values. We assume a canonical order on action names; let ainit_i be the i^{th} name under this canonical order. The *initial environment* includes the action names used by the initial process. The *initial trace* uses the reserved thread name tinit. These are defined as follows.

Treating components as sets of declarations, we first define $\mathit{flat}(M \parallel N) \triangleq \mathit{flat}(M) \cup \mathit{flat}(N)$ and $\mathit{flat}(M \setminus f) \triangleq \mathit{flat}(M)$.

Let $\mathit{flat}(M) = \mathsf{var}\ \vec{x}\mathsf{=}\vec{u}; \mathsf{atomic}\ \vec{w}\mathsf{=}\vec{v}; \mathsf{thrd}\ \vec{t}\ \vec{D}; \mathsf{fun}\ \vec{f}\ \vec{\Lambda}$. We then define the following.

$$\begin{aligned}
\mathit{initEnv}(M) &\triangleq (M, \{\mathsf{ainit}_1, \ldots, \mathsf{ainit}_\ell\}, \emptyset) \\
\mathit{initTrace}(M) &\triangleq \langle \mathsf{tinit\ wr}\ x_1\ u_1\ \mathsf{ainit}_1\rangle\langle \mathsf{com\ tinit}\ x_1\ \mathsf{ainit}_1\rangle \cdots \\
&\qquad\quad \cdots \langle \mathsf{tinit\ wr}\ x_\ell\ u_\ell\ \mathsf{ainit}_\ell\rangle\langle \mathsf{com\ tinit}\ x_\ell\ \mathsf{ainit}_\ell\rangle \\
\mathit{initProc}(M,S) &\triangleq \mathsf{var}\ x_1\mathsf{=}u_1\mathsf{@ainit}_1 \mid \cdots \mid \mathsf{var}\ x_\ell\mathsf{=}u_\ell\mathsf{@ainit}_\ell \mid \\
&\ \ \mathsf{atomic}\ w_1\mathsf{=}v_1 \mid \cdots \mid \mathsf{atomic}\ w_m\mathsf{=}v_m \mid \\
&\ \ \mathsf{thrd}\ t_1\ \emptyset\ D_1 \mid \cdots \mid \mathsf{thrd}\ t_n\ \emptyset\ D_n \mid \\
&\ \ \mathsf{thrd}\ s_1\ \emptyset\ (\mathsf{test\ main}) \mid \cdots \mid \mathsf{thrd}\ s_n\ \emptyset\ (\mathsf{test\ main}) \\
\mathscr{O}_{\mathscr{W}}[\![M]\!](S) &\triangleq \{\sigma'\sigma \mid \sigma' = \mathit{initTrace}(M) \text{ and } (\mathit{initProc}(M,S)) \xRightarrow[\mathit{initEnv}(M)]{\sigma}_{\mathscr{W}}\}
\end{aligned}$$

Note that $\mathscr{O}_{\mathscr{W}}$ is a partial function: $\mathscr{O}_{\mathscr{W}}[\![M]\!](S)$ is undefined when $\mathit{thrds}(\mathscr{W}) \cap S \neq \emptyset$.

C.3 I/O semantics

The rules for I/O are given in Figure 2.

Fix an environment Δ. When a function is called, the result is a *frame*, $f\,.\,\{C\}$ or $f\,!\,\{C\}$, where f is the function name and C is the function body (with parameter substitution). A "." frame is created when the caller is internal to Δ. A "!" frame is created when the caller is external to Δ. The sequence of actions generated by the transition system between the creation of a "." frame and its removal is always an active trace; whereas the sequence generated between the creation of a "!" frame and its removal is always a quiescent trace.

A call to an external function (not defined in Δ) creates a test g. This changes context from inside Δ to outside; such tests can invoke functions from Δ before returning values in response to the initial external function call.

Example C.2. Suppose f is defined by a component and g is not. Eliding some notation, here are examples of the component calling itself, the component calling out, and a test calling in.

$$f\texttt{()} \xrightarrow{\langle \texttt{.call}\ f\rangle} f\texttt{.\{}\cdots\texttt{\}} \xrightarrow{\varepsilon} f\texttt{.\{return 0\}} \xrightarrow{\langle \texttt{.ret 0}\rangle} 0$$

$$g\texttt{()} \xrightarrow{\langle \texttt{!call}\ g\rangle} \mathsf{test}\ g \xrightarrow{\langle \texttt{?ret 42}\rangle} 42$$

$$\mathsf{test}\ g \xrightarrow{\langle \texttt{?call}\ f\rangle} f\texttt{!\{}\cdots\texttt{\};}\mathsf{test}\ g \xrightarrow{\varepsilon}\xrightarrow{\langle \texttt{!ret 0}\rangle} 0\texttt{;}\mathsf{test}\ g \xrightarrow{\varepsilon} \mathsf{test}\ g \qquad \square$$

C.4 Memory semantics for seq, tso and pso

The rules for memory actions are given in Figure 3.

The atomic operations are the same for all three models. Atomic writes and successful cas actions require that the thread's buffer be empty.

seq has one rule for reads and one for writes. The buffers are ignored under seq, and thus they remain empty.

tso and pso each have two rules for read and two for write. Only the second write rule distinguishes them. When a process performs a write, the value is first stored in a buffer. It can be read from there only by the thread that performed the write. At any moment, an action in the buffer may be committed. tso always commits the first buffered write, whereas pso may commit any write so long as there is not a prior write of the same variable. After the commit, any thread may read the value so long as that thread does not itself have a buffered write on the same variable.

The read operation is deterministic under tso and pso. Only the commit is nondeterministic.

Thus, the standard tso "fence" operation can be encoded as fence=0, where fence is a reserved atomic, which is never read. (To satisfy interference freedom, one can use a separate fence for each component.)

C.5 Semantics for hb

The central idea of an hb model is that a read may be satisfied by any write as long as there is no intervening write that "happens-between" the write and the read. These models are complicated by the fact that reads may match future writes. Without further constraint, hb models allow DRF programs to produce surprising results.

Example C.3. Consider the trace

$$\langle \mathsf{s\ rd\ x\ 1}\rangle\langle \mathsf{s\ wr\ y\ 1}\rangle\langle \mathsf{t\ rd\ y\ 1}\rangle\langle \mathsf{t\ wr\ x\ 1}\rangle.$$

This can be generated by the following program under hb.

```
var x; var y;
thread s { x; y=1 }
thread t { y; x=1 }
```

However, it is considered unacceptable that the following DRF program should generate the same trace.

```
var x; var y;
thread s { if x==1 then y=1 }
thread t { if y==1 then x=1 }
```

In this case, a read of 1 is considered a "thin air" read [Manson et al. 2005] □

The operational semantics that we present is a simplified variant of the semantics from our earlier paper [Jagadeesan et al. 2010]. It is an hb model that avoids thin-air reads. For DRF programs, our model coincides with the JMM. For programs without synchronization, our model allows more executions than the JMM. For programs with both data races and synchronization, our model differs from the JMM.

The hb semantics does not require the $<_{\mathsf{rf}}$ relation, and therefore does not require the action names annotating read and write actions. We include these annotations only so that we have a single definition of well-formed traces.

Processes have no global store for data variables. Instead, processes are defined using *pseudo-actions*, κ, which record all of the information about past actions that is necessary to determine the visible values of data variables.

$$\kappa \;::=\; \langle s \;\mathsf{wr}\; x\; u\; a\rangle \;\mid\; \langle s\; \underline{\mathsf{cas}}\; w\; u\rangle \;\mid\; \langle s\; \underline{\mathsf{wr}}\; w\; u\rangle \;\mid\; \langle s\; \underline{\mathsf{rd}}\; w\rangle \;\mid\; \langle s\; \underline{\mathsf{wr}}\; a\rangle \;\mid\; \langle s\; \underline{\mathsf{rd}}\; A\rangle$$

Read actions need not be recorded as pseudo-actions. The pseudo-actions for atomic variables record the written value. The pseudo-action $\langle s\; \underline{\mathsf{wr}}\; a\rangle$ is generated by a $\langle s\,!\; a\rangle$ action; likewise, $\langle s\; \underline{\mathsf{rd}}\; A\rangle$ is generated by a $\langle s?\; A\rangle$ action.

Processes and process contexts are defined as follows.

$$\begin{aligned} P, Q \;&::=\; \mathsf{thrd}\; s\; C \;\mid\; P \mid Q \;\mid\; \kappa P \;\mid\; \langle s\; \mathsf{spec}\; x\; u\; a\rangle P \;\&\; Q \\ \mathbb{P} \;&::=\; [-] \;\mid\; \mathbb{P} \mid Q \;\mid\; Q \mid \mathbb{P} \;\mid\; \kappa\mathbb{P} \;\mid\; \langle s\; \mathsf{spec}\; x\; u\; a\rangle \mathbb{P} \;\&\; Q \;\mid\; \langle s\; \mathsf{spec}\; x\; u\; a\rangle Q \;\&\; \mathbb{P} \end{aligned}$$

In $\langle s\; \mathsf{spec}\; x\; u\; a\rangle P \;\&\; Q$, the speculation is visible to P but not Q; we call P the *final* process and Q the *initial* or *justifying* process.

A simple structural equivalence is not sufficient. We define the *structural order* ($P \triangleright Q$) to be the least preorder that satisfies the rules in Figure 4. We write $P \triangleright Q$ as shorthand for $P \triangleright Q$. The rules in the table are divided into three groups. The first group gives the congruence rule and the traditional rules for composition. The second group gives the rules for speculation, including those which involve both speculation and composition. Finally, the last group gives the rules for action prefixing; these include several rules for multiple actions from a single thread. Note that the rule for speculation does not require that the action name a be fresh; freshness is ensured by the write in the final branch of speculation. This allows speculation on a write which has already occurred:

$$\langle s\; \mathsf{wr}\; x\; u\; a\rangle P \triangleright \langle s\; \mathsf{spec}\; x\; u\; a\rangle (\langle s\; \mathsf{wr}\; x\; u\; a\rangle P) \;\&\; (\langle s\; \mathsf{wr}\; x\; u\; a\rangle P)$$

$$
\begin{array}{rll}
\mathbb{P}[Q \mid Q'] & \rhd \mathbb{P}[Q' \mid Q] & \\
\mathbb{P}[Q \mid (Q' \mid Q'')] & \rhd \mathbb{P}[(Q \mid Q') \mid Q''] & \\[1ex]
\mathbb{P}[Q] & \rhd \mathbb{P}[\langle s\ \mathsf{spec}\ x\ u\ a\rangle Q \mathbin{\&} Q] & \text{if } s \in \mathit{thrds}(Q) \\
\mathbb{P}[Q \mid \langle s\ \mathsf{spec}\rangle Q' \mathbin{\&} Q''] & \rhd \mathbb{P}[\langle s\ \mathsf{spec}\rangle (Q \mid Q') \mathbin{\&} (Q \mid Q'')] & \\
\mathbb{P}[\langle s\ \mathsf{spec}\ x\ u\ a\rangle (\vec{\kappa} Q) \mathbin{\&} (\vec{\kappa} Q')] & \rhd \mathbb{P}[\vec{\kappa} Q] \quad \text{if } \exists i.\ \kappa_i = \langle s\ \mathsf{wr}\ x\ u\ a\rangle & \\
& \text{and } \forall j.\ \kappa_j \neq \langle s\ \underline{\mathsf{cas}}\rangle \text{ and either } \kappa_j \neq \langle s\ \underline{\mathsf{rd}}\rangle \text{ or } \kappa_j = \langle s\ \underline{\mathsf{rd}}\ \emptyset\rangle & \\[1ex]
\mathbb{P}[Q \mid \kappa Q'] & \rhd \mathbb{P}[\kappa (Q \mid Q')] & \text{if } \mathit{thrd}(\kappa) \notin \mathit{thrds}(Q) \\
\mathbb{P}[\langle s\ \mathsf{wr}\ x\ u\rangle \langle s\ \mathsf{wr}\ y\ v\rangle Q] & \rhd \mathbb{P}[\langle s\ \mathsf{wr}\ y\ v\rangle \langle s\ \mathsf{wr}\ x\ u\rangle Q] & \text{if } x \neq y
\end{array}
$$

Fig. 4: HBC structural order

$$\rhd \langle s\ \mathsf{wr}\ x\ u\ a\rangle P$$

As for the other orders, evaluation is preserved by structural order.

$$P \xrightarrow[\Delta/\Delta']{\sigma}_{\mathsf{hb}} P' \quad \text{if } P \rhd Q \xrightarrow[\Delta/\Delta']{\sigma}_{\mathsf{hb}} Q' \rhd P'$$

Evaluation is defined using speculative evaluation contexts, $\mathbb{E}$, and the derived function $\mathbb{E}(s,x)$, which returns the set of values of x that are visible for s in the hole of $\mathbb{E}$.

Definition C.4. Define *extended pseudo-actions* and *speculative evaluation contexts* as follows.

$$
\begin{array}{rcl}
\varkappa & ::= & \kappa \mid \langle s\ \mathsf{spec}\ x\ u\ a\rangle \\
\mathbb{E} & ::= & [\text{–}] \mid \mathbb{E} \mid Q \mid Q \mid \mathbb{E} \mid \varkappa \mathbb{E}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{flatten}([\text{–}]) = \varepsilon & \mathit{flatten}(\mathbb{E} \mid Q) = \mathit{flatten}(\mathbb{E}) \\
\mathit{flatten}(\varkappa \mathbb{E}) = \varkappa\, \mathit{flatten}(\mathbb{E}) & \mathit{flatten}(Q \mid \mathbb{E}) = \mathit{flatten}(\mathbb{E})
\end{array}
$$

Let $\mathit{numActs}(w, Q)$ be the number of w-actions in Q, where an w-action is either $\langle s\ \underline{\mathsf{wr}}\ w\rangle$, $\langle s\ \underline{\mathsf{rd}}\ w\rangle$, or $\langle s\ \underline{\mathsf{cas}}\ w\rangle$, and similarly for $\mathit{numActs}(w, \mathbb{E})$ and $\mathit{numActs}(w, \vec{\varkappa})$. We say that $\mathbb{E}$ *enables* w if $\mathit{numActs}(w, \mathit{flatten}(\mathbb{E})) = \mathit{numActs}(w, \mathbb{E})$.

Define $\mathbb{E}(w)$ to return the last value written to w in $\mathbb{E}$. That is, $\mathbb{E}(w) = u$ if $\mathit{flatten}$ $(\mathbb{E}) = \varkappa_1 \cdots \varkappa_n$ and $\exists i.\ \varkappa_i \in \{\langle \underline{\mathsf{wr}}\ w\ u\rangle, \langle \underline{\mathsf{cas}}\ w\ u\rangle\}$ and $\forall j > i.\ \varkappa_j \notin \{\langle \underline{\mathsf{wr}}\ w\rangle, \langle \underline{\mathsf{cas}}\ w\rangle\}$.

Adapt the $<_{\mathsf{hb}}$ order to sequences of extended pseudo-actions in the obvious way. In particular, $\langle s\ \underline{\mathsf{wr}}\ a\rangle$ and $\langle s\ \underline{\mathsf{rd}}\ A\rangle$ are related by $<_{\mathsf{hb}}$ when $a \in A$.

Define $\mathbb{E}(s,x)$ to return the set of pairs $u@a$ such that $\mathit{flatten}(\mathbb{E}) = \varkappa_1 \cdots \varkappa_n$ and for some $0 \leq i \leq n$, both of the following are true.

- Either $\varkappa_i = \langle \mathsf{wr}\ x\ u\ a\rangle$ or $\varkappa_i = \langle t\ \mathsf{spec}\ x\ u\ a\rangle$ for some $t \neq s$.
- There exists no $\varkappa_j = \langle \mathsf{wr}\ x\rangle$ such that $i <_{\mathsf{hb}}^{\vec{\varkappa}} j <_{\mathsf{hb}}^{\vec{\varkappa}\langle s\ \mathsf{wr}\rangle} (n+1)$, where $\langle s\ \mathsf{wr}\rangle$ is an arbitrary non-synchronizing extended pseudo-action for thread s. □

The interesting rules for the evaluation relation are found in Figure 5.

Most of the operational rules are morally the same as those for seq; however, the form is different. For example, the seq rule for sequencing is $\mathsf{thrd}\ s\ \beta\ \mathbb{C}[u;D] \xrightarrow{\varepsilon}_{\mathsf{seq}}$ $\mathsf{thrd}\ s\ \beta\ \mathbb{C}[D]$. For hb, we write this as follows.

$$\mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[u;D]\big] \xrightarrow[\Delta/\Delta']{\varepsilon}_{\mathsf{hb}} \mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[D]\big]$$

$$\begin{array}{rcll}
\mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[f\,(\vec{u})]\big] & \xrightarrow[\Delta/\Delta,a]{\langle s\,\mathsf{!call}\ f\ \vec{u}\ a\ \emptyset\rangle}_{\mathsf{hb}} & \mathbb{E}\big[\langle s\ \underline{\mathsf{wr}}\ a\rangle\mathsf{thrd}\ s\ \mathbb{C}[\mathsf{test}\ f]\big] & \text{if } \Delta(f) \text{ undefined} \\
\mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[\mathsf{test}\ f]\big] & \xrightarrow[\Delta/\Delta,a]{\langle s\,\mathsf{?ret}\ f\ \vec{u}\ a\ B\rangle}_{\mathsf{hb}} & \mathbb{E}\big[\langle s\ \underline{\mathsf{rd}}\ B\rangle\mathsf{thrd}\ s\ \mathbb{C}[\vec{u}]\big] & \\
\mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[\mathsf{test}\ g]\big] & \xrightarrow[\Delta/\Delta,a]{\langle s\,\mathsf{?call}\ f\ \vec{u}\ a\ B\rangle}_{\mathsf{hb}} & \mathbb{E}\big[\langle s\ \underline{\mathsf{rd}}\ B\rangle\mathsf{thrd}\ s\ \mathbb{C}[f\,\mathsf{!}\{D\{\![\vec{u}/\vec{z}]\!\}\}\mathsf{;}\,\mathsf{test}\ g]\big] & \text{if } \Delta(f,s,\mathbb{C}) = (\vec{z})\{D\} \\
\mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[f\,\mathsf{!}\{\mathsf{return}\ \vec{u}\mathsf{;}D\}]\big] & \xrightarrow[\Delta/\Delta,a]{\langle s\,\mathsf{!ret}\ f\ \vec{u}\ a\ \emptyset\rangle}_{\mathsf{hb}} & \mathbb{E}\big[\langle s\ \underline{\mathsf{wr}}\ a\rangle\mathsf{thrd}\ s\ \mathbb{C}[\vec{u}]\big] & \\
\mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[x\mathsf{=}v]\big] & \xrightarrow[\Delta/\Delta,a]{\langle s\ \mathsf{wr}\ x\ v\ a\rangle}_{\mathsf{hb}} & \mathbb{E}\big[\langle s\ \mathsf{wr}\ x\ v\ a\rangle\mathsf{thrd}\ s\ \mathbb{C}[v]\big] & \\
\mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[x]\big] & \xrightarrow{\langle s\ \mathsf{rd}\ x\ u\ a\rangle}_{\mathsf{hb}} & \mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[u]\big] & \text{if } u\mathsf{@}a \in \mathbb{E}(s,x) \\
\mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[w\mathsf{=}v]\big] & \xrightarrow{\langle s\ \underline{\mathsf{wr}}\ w\rangle}_{\mathsf{hb}} & \mathbb{E}\big[\langle s\ \underline{\mathsf{wr}}\ w\ v\rangle\ \mathsf{thrd}\ s\ \mathbb{C}[v]\big] & \text{if } \mathbb{E} \text{ enables } w \\
\mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[w]\big] & \xrightarrow{\langle s\ \underline{\mathsf{rd}}\ w\rangle}_{\mathsf{hb}} & \mathbb{E}\big[\langle s\ \underline{\mathsf{rd}}\ w\rangle\ \ \mathsf{thrd}\ s\ \mathbb{C}[u]\big] & \text{if } \mathbb{E} \text{ enables } w \text{ and } \mathbb{E}(w) = u \\
\mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[w\mathsf{.cas}(u,v)]\big] & \xrightarrow{\langle s\ \underline{\mathsf{rd}}\ w\rangle}_{\mathsf{hb}} & \mathbb{E}\big[\langle s\ \underline{\mathsf{rd}}\ w\rangle\ \ \mathsf{thrd}\ s\ \mathbb{C}[0]\big] & \text{if } \mathbb{E} \text{ enables } w \text{ and } \mathbb{E}(w) \neq u \\
\mathbb{E}\big[\mathsf{thrd}\ s\ \mathbb{C}[w\mathsf{.cas}(u,v)]\big] & \xrightarrow{\langle s\ \underline{\mathsf{cas}}\ w\rangle}_{\mathsf{hb}} & \mathbb{E}\big[\langle s\ \underline{\mathsf{cas}}\ w\ v\rangle\mathsf{thrd}\ s\ \mathbb{C}[1]\big] & \text{if } \mathbb{E} \text{ enables } w \text{ and } \mathbb{E}(w) = u
\end{array}$$

$$\frac{\mathbb{E}\big[\langle s\ \mathsf{spec}\ x\ u\ a\rangle P\big] \xrightarrow[\Delta/\Delta']{\alpha}_{\mathsf{hb}} \mathbb{E}\big[\langle s\ \mathsf{spec}\ x\ u\ a\rangle P'\big]}{\mathbb{E}\big[\langle s\ \mathsf{spec}\ x\ u\ a\rangle P\ \mathsf{\&}\ Q\big] \xrightarrow[\Delta/\Delta']{\alpha}_{\mathsf{hb}} \mathbb{E}\big[\langle s\ \mathsf{spec}\ x\ u\ a\rangle P'\ \mathsf{\&}\ Q\big]}\ \text{if } \alpha \text{ is not I/O}$$

$$\frac{\mathbb{E}\big[Q\big] \xrightarrow[\Gamma/\Gamma']{\alpha}_{\mathsf{hb}} \mathbb{E}\big[Q'\big]}{\mathbb{E}\big[\langle s\ \mathsf{spec}\ x\ u\ a\rangle P\ \mathsf{\&}\ Q\big] \xrightarrow[\Delta/\Delta]{\varepsilon}_{\mathsf{hb}} \mathbb{E}\big[\langle s\ \mathsf{spec}\ x\ u\ a\rangle P\ \mathsf{\&}\ Q'\big]}\ \text{if } \alpha \text{ is not I/O}$$

$$\frac{\mathbb{E}\big[\langle s\ \mathsf{spec}\ x\ u\ a\rangle P\big] \xrightarrow[\Delta/\Delta']{\alpha}_{\mathsf{hb}} \mathbb{E}\big[\langle s\ \mathsf{spec}\ x\ u\ a\rangle P'\big] \quad \mathbb{E}\big[Q\big] \xrightarrow[\Gamma/\Gamma']{\alpha}_{\mathsf{hb}} \mathbb{E}\big[Q'\big]}{\mathbb{E}\big[\langle s\ \mathsf{spec}\ x\ u\ a\rangle P'\ \mathsf{\&}\ Q\big] \xrightarrow[\Delta/\Delta']{\alpha}_{\mathsf{hb}} \mathbb{E}\big[\langle s\ \mathsf{spec}\ x\ u\ a\rangle P'\ \mathsf{\&}\ Q'\big]}\ \text{if } \alpha \text{ is I/O}$$

Fig. 5: HBC memory rules

The rules of Figure 1 and the rules for $\langle\mathsf{?call}\rangle$ and $\langle\mathsf{!ret}\rangle$ actions from Figure 2 are translated to hb in this way.

Given this definition of evaluation ($\longrightarrow_{\mathsf{hb}}$) and multistep evaluation ($\Longrightarrow_{\mathsf{hb}}$) are defined as before. The semantic function only includes traces where the end process is speculation-free.

$$\begin{array}{rl}
\mathit{initProc}(M,S) \triangleq & \langle\mathsf{tinit}\ \mathsf{wr}\ x_1\ u_1\ \mathsf{ainit}_1\rangle \cdots \langle\mathsf{tinit}\ \mathsf{wr}\ x_\ell\ u_\ell\ \mathsf{ainit}_\ell\rangle \\
& \langle\mathsf{tinit}\ \underline{\mathsf{wr}}\ w_1\ v_1\rangle \cdots \langle\mathsf{tinit}\ \underline{\mathsf{wr}}\ w_m\ v_m\rangle \\
& \quad \big(\mathsf{thrd}\ t_1\ D_1 \mid \cdots \mid \mathsf{thrd}\ t_n\ D_n \mid \\
& \quad\ \ \mathsf{thrd}\ s_1\ (\mathsf{test\ main}) \mid \cdots \mid \mathsf{thrd}\ s_n\ (\mathsf{test\ main})\big) \\
\mathsf{hb}[\![M]\!](S) \triangleq & \{\sigma'\sigma \ \mid\ \sigma' = \mathit{initTrace}(M) \text{ and } (\mathit{initProc}(M,S)) \xRightarrow[\mathit{initEnv}(M)]{\sigma}_{\mathscr{W}} P' \\
& \text{and } P' \text{ is speculation-free.}\}
\end{array}$$