Types for Relaxed Memory Models

Matthew Goto   Radha Jagadeesan   Corin Pitcher   James Riely

DePaul University

Abstract

Multicore computers implementing weak memory models are mainstream, yet type-based analyses of these models remain rare. We help fill this gap. We not only prove the soundness of a type system for a weak execution model, but we also show that interesting properties of that model can be embedded in the types themselves.

We argue that correspondence assertions can be used in a programming discipline that captures happens-before relationships, which are the basis for reasoning about weak memory in Java. This programming discipline is flexible and can be statically enforced. We present several examples from `java.util.concurrent` and prove the static semantics sound with respect to an execution model based on Java's memory model.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Theory, Languages

1. Introduction

The C++ memory model posits that programs with data races have undefined semantics [Boehm and Adve 2008]. This model is incompatible with languages such as Java, which allow data races but seek to ensure safety. The Java Memory Model (JMM) [Manson, Pugh, and Adve 2005] is a *speculative model* intended to validate the executions generated by modern processors and compilers while banning *thin-air* reads, and thus to ensuring type safety. To our knowledge, the safety argument has only been made informally.

The *happens-before* relation is crucial to understanding shared-memory concurrent Java programs. The Java APIs contain extensive comments describing the happens-before relation, such as the following: "Actions in a thread prior to placing an object into any concurrent collection happen-before actions subsequent to the access or removal of that element from the collection in another thread." There is no method for describing happens-before relations formally.

Consider the following tiny example, which shows that, without proper synchronization, it is possible for a thread to read stale values. We adopt the following notational conventions: `s` and `t` are thread identifiers; `p` is an object identifier; the process `s[`M`]` denotes a thread with identifier `s`, executing statement M; the process $A \,|\, B$ denotes the parallel composition of threads A and B.

```
class C1 { int f; int g; }
s[p.f=42; p.g=27;] |
t[val y=p.g; val x=p.f; if((y!=0)&&(x==0)){†}]
```

There are two threads sharing object `p` inhabiting class `C1` (notation `p:C1`). All fields are initially set to `0`. Object `p` has two fields. Thread `s` updates `p.f` then `p.g`. Thread `t` reads the fields in the reverse order. Because there is no synchronization, the program has data races on both fields.

In a *sequentially consistent* execution [Lamport 1979], threads must appear to execute sequentially in an interleaved fashion. Since `s` must write `p.f` before `p.g`, it is impossible for `t` to read an object from `p.g` and subsequently read `0` from `p.f`; thus the statement † is unreachable in a sequentially consistent semantics.

However, compilers may re-order unrelated writes and reads, and thus the JMM semantics is designed to allow executions in which † is reachable in example `C1`.

Synchronization eliminates data races, shrinking the set of visible values. For example, in the following variant of example `C1`, field `g` is declared as `volatile`. There are happens-before edges from a write of a volatile to every subsequent read. Assume `p:C2`.

```
class C2 { int f; volatile int g; }
s[p.f=42; p.g=27;] |
t[val y=p.g; val x=p.f; if((y!=0)&&(x==0)){†}]
```

Because of the synchronization though `p.g`, † becomes unreachable in the JMM for `C2`. Such happens-before reasoning is subtle: in the variant where `f` is volatile but `g` is not, † is reachable. The reasoning may also fail in the presence of other threads.

In this paper we propose a *a compositional program discipline for happens-before* using *correspondence assertions* [Woo and Lam 1992; Gordon and Jeffrey 2003]. As a byproduct, we present a proof of type safety for concurrent Java.

Organization. In Section 2, we sketch the semantics of our memory model and present correspondence assertions as a tool for reasoning about happens-before relations. Section 3 is a brief detour to complete the description of the semantics; we introduce speculation and describe its implications for type soundness. In Section 4, we provide intuitions about the type system via examples. For readability, Sections 1–4 sometimes use Java syntax. For simplicity of formal development, the remainder of the paper uses a more restrictive language.

In Section 5, we define the static type system, which is used to validate class definitions. In Section 6, we define soundness using an operational semantics. In Section 7, we describe the soundness proof, highlighting the most interesting cases of the inductive invariant maintained by the operational semantics. In Section 8, we summarize related work and conclude.

Details of the soundness proof, including the typing rules for processes used in subject reduction, are available in the full version of the paper.

TLDI'12, January 28, 2012, Philadelphia, PA, USA.
Copyright © 2012 ACM 978-1-4503-1120-5/12/01…$10.00

Acknowledgements. We gratefully acknowledge the detailed comments of the anonymous referees on this and a previous version of the paper. This research was supported by NSF grant #0916741.

2. Happens-before semantics using correspondence assertions

We provide a rough introduction to our operational semantics and describe how correspondence assertions can be integrated to reason about happens-before relations. The semantics is essentially copied from our previous paper [Jagadeesan, Pitcher, and Riely 2010], although it has been adapted to handle atomics.

The semantics is based on the idea that writes do not affect a traditional store, but rather produce *actions*. When a read occurs, the resulting value can be determined by the visible writes. Reduction relates *processes*, which are snapshots of a running system. As a first approximation, one can view a process as a collection of object denotations and threads, surrounded by a sequence of actions. Consider the following reductions using `class Int {int f;}`, where `p:Int`.

$$\begin{aligned}&\texttt{p:Int | s[p.f=42; p.f=27;] | t[val x=p.f;]}\\ \to\ &\langle\texttt{s!p.f=42}\rangle(\texttt{p:Int | s[p.f=27;] | t[val x=p.f;]})\\ \to\ &\langle\texttt{s!p.f=42}\rangle\langle\texttt{s!p.f=27}\rangle(\texttt{p:Int | s[] | t[val x=p.f;]})\end{aligned}$$

After the first reduction, `t` can read 42. After the second reduction, `t` can read either 42 or 27. The read operation is nondeterministic in the latter case.

Java does not have thread or object literals, and so writing Java programs to generate such processes is a bit of a chore. In this case, we must write initialization code that creates the shared object and the threads:

```
class S { Int p; S(Int p) {this.p=p;}
  void run() {p.f=42; p.f=27;}
}
class T { Int p; T(Int p) {this.p=p;}
  void run() {val x=p.f} // using untyped declarations
}
class Main {
  static void main() {
    val p = new Int();
    new S(p).start();
    new T(p).start();
} }
```

For expository purposes, we consistently elide this bootstrap code, looking instead at a literal expression of the resulting process.

Once running, a process context defines a sequence of actions (those leading up to the context hole) and a happens-before relation over that sequence. For example, if

$$\mathbb{C} = \langle\texttt{s!p.f=42}\rangle\langle\texttt{s!p.f=27}\rangle(\texttt{p:Int |}\ [\![-]\!])$$

then there are two write actions in the sequence, with $\langle\texttt{s!p.f=42}\rangle$ happening-before $\langle\texttt{s!p.f=27}\rangle$. The same context can provide different values to different threads. For example, $\mathbb{C}$ justifies reading either 42 or 27 from `p.f` for thread `t`; whereas $\mathbb{C}$ can only justify reading 27 from `p.f` for thread `s`. We formalize this by adding an action for the context hole, labeled by the identifier of the thread within the hole that is attempting to read.

The sequence of actions from a single thread defines *program order*. The sequence of actions on a single synchronization variable defines *synchronization order*. Happens-before is defined to be the transitive closure of program order and synchronization order. A write action is available to a read unless it happens-before another write action that in turn happens-before the read; the "other" write *intervenes* between the first write and the read, hiding the first write.

We discuss synchronization order using Java's *atomic* objects, rather than volatiles. A volatile field can be viewed as a final atomic upon which we only call `get` and `set`. In addition, atomics have get-and-set and compare-and-swap methods, which can be used to implement locks.

Synchronization order specifies that a "set" on an atomic happens-before any subsequent "get." We model this using synchronization actions of the form $\langle \ell\,\texttt{!}\,s\texttt{:}\,j\rangle$, where ℓ the atomic object acted upon, s is the acting thread, and j is an *opcount*. If the opcount is odd, then the operation was a set. If the opcount is even, then the operation was a get. The opcount is incremented when a set follows a get or a get follows a set. Initially, atomics are "set" with opcount `1`.

Reading an atomic is deterministic; thus we store the value held by an atomic with its denotation, rather than representing it using actions. The form of an atomic denotation is $\texttt{Atomic\{}v\texttt{;}j\texttt{\}}$, where v is the current value and j is the opcount.

Consider the following variant of example `C2` from the introduction.

```
p:Int | l:Atomic{0;1} |
s[p.f=42; l.set(27);] |
t[val y=l.get(); val x=p.f; if((x==0)&&(y!=0)){†}]
```

Just as for example `C1`, statement † is unreachable. For example, consider the case where thread `s` executes before `t`. In this case, the read of `p.f` occurs in the following context.

$$\begin{aligned}&\langle\texttt{s!p.f=42}\rangle\langle\texttt{s!l:1}\rangle\langle\texttt{t!l:2}\rangle\\ &\qquad(\texttt{p:Int | l:Atomic\{27;2\} | s[] |}\ [\![-]\!])\end{aligned}$$

The synchronization action $\langle\texttt{s!l:1}\rangle$ was created when `s` performed `l.set(27)`; this operation modified the value of the atomic, but not its opcount, resulting in `l:Atomic{27;1}`. The next synchronization action, $\langle\texttt{t!l:2}\rangle$, was created when `t` performed `l.get()`; this operation modified the opcount of the atomic, but not its value, resulting in `l:Atomic{27;2}`. Note that synchronization actions record the opcount, whereas write actions, such as $\langle\texttt{s!p.f=42}\rangle$, record the value written.

The above context has three happens-before edges, when interpreted as a context for `t`.

- program order from $\langle\texttt{s!p.f=42}\rangle$ to $\langle\texttt{s!l:1}\rangle$,
- synchronization order from $\langle\texttt{s!l:1}\rangle$ to $\langle\texttt{t!l:2}\rangle$, and
- program order from $\langle\texttt{t!l:2}\rangle$ to the hole.

Happens-before is also affected by thread creation. For example, the bootstrap code given above will evaluate to the following context, holding process `s[···] |t[···]`.

$$\begin{aligned}&(\nu\texttt{m})(\nu\texttt{p})(\nu\texttt{s})(\nu\texttt{t})\\ &\quad\langle\texttt{m!p.f=0}\rangle\langle\texttt{m!l:1}\rangle\langle\texttt{m!k:1}\rangle\langle\texttt{s!l:2}\rangle\langle\texttt{t!k:2}\rangle\\ &\qquad(\texttt{m:Main | p:Int | s:S | t:T |}\ [\![-]\!])\end{aligned}$$

Given the semantics of `start`, there are synchronization edges from `m` to the threads that it creates, here manifest in the synchronization variables `l` and `k`, which will never be used again. Because of these synchronization edges, the write of the initial value of `p.f` happens-before any actions performed by `s` or `t`, since these threads are part of the process filling the hole in the context.

Many concurrent algorithms use locks. When locks are implemented using atomics, the release of a lock happens-before the subsequent acquire.

In this paper we reason about happens-before relations using *correspondence assertions* [Woo and Lam 1992; Gordon and Jeffrey 2003]. Using this technique, a programmer may annotate a program with `begin` and `end` statements. The execution of an end statement is *correct* if it occurs *after* the corresponding begin. The primary goal of static analysis is to ensure that all ends are correct.

In a well-typed program, therefore, end statements can be erased without changing the meaning of the program.

Correspondences have no computational effect. The particular form of correspondence is chosen in order to specify some property of interest. We wish to reason about happens-before relations in code built using atomics and other mechanisms, and we wish to do so in a way that might be useful for API documentation.

Recall the example from `java.util.concurrent` quoted in the introduction: "Actions in a thread prior to placing an object into any concurrent collection happen-before actions subsequent to the access or removal of that element from the collection in another thread." Directly capturing such statements requires dependent types, as well as a selection of analyses based on locality [Sewell 1998], confinement [Zhao, Palsberg, and Vitek 2006], linearity [Hawblitzel 2005], and effects [Lucassen and Gifford 1988]. Such powerful type systems provide strong guarantees at the cost of additional programmer effort that we do not wish to mandate.

Our goal has been to develop a type system that captures some property of happens-before while retaining simple types. To achieve this, we propose a rather weak notion of correspondence.

In our minimalist approach, the creation of an object `p` by a thread `s` begins a correspondence, creating a begin action $\langle$`s!p`$\rangle$. At any point, a program can attempt to end the correspondence using the statement `end p`. If $\langle$`s!p`$\rangle$ happens-before the hole in the context where `end p` occurs, then we say that this occurrence of `p` is *correctly published*, and the end acts as a no-op; otherwise the thread becomes *stuck*. For example, consider the following process.

$$\langle \texttt{s!p} \rangle\, (\texttt{s[end p; }\cdots\texttt{] | t[end p; }\cdots\texttt{]})$$

Thread `s` evaluates to `s[`$\cdots$`]` whereas thread `t` is stuck: the begin in context $\langle$`s!p`$\rangle$ $[\![-]\!]$ only happens-before the hole when the hole is filled by `s`. A stuck program is erroneous; typing ensures that well-typed programs cannot get stuck.

This form of specification is weak in the sense that a programmer cannot choose the point at which a begin occurs; begins always coincide with object creation. The programmer is also restricted to end on a single object reference, not a tuple.

Additionally, Java allows the null value wherever an object reference can occur, and we make no attempt to countermand this design decision. As in FJ [Igarashi, Pierce, and Wadler 2001], the definition of stuckness is tuned to the power of the type system, which cannot prevent null or casting errors. By definition, null errors and casting errors do not cause a *stuck* thread, simply an *irreducible* one. Stuckness is precisely defined in Section 6.2.

Nonetheless, our correspondences can be used to associate arbitrary program points by creating a dummy object to denote the begin statement, communicating that object, and then performed an end using the received value. The correspondence is only meaningful, however, if the received value is that which is expected. Since begins are performed on *every* object, we have factored out any data dependencies and therefore removed the need for dependent types.

To understand a concurrent data structure, one must understand both the data structure invariants and the memory effects. Our correspondence should be read as capturing the necessary memory effects under the supposition that the (informal) data structure invariants hold.

In general, shared fields do not preserve safety; that is, a thread may get stuck if it performs an end on a value read from a shared field, even if the writer could have performed the end. For example, consider the following process.

$$\langle \texttt{s!o} \rangle\ \langle \texttt{s!p.f=o} \rangle\ (\texttt{s[end o;] | t[val x=p.f; end x;]})$$

Thread `s` has created and written `o`. The residual `s[end o;]` is safe, reducing to `s[]`; however the residual `t[end o;]` is stuck.

Atomics do preserve safety. Consider the following program, derived from the previous one by replacing `p.f` with atomic `l`.

$$\langle \texttt{s!o} \rangle\ \langle \texttt{s!l:1} \rangle\ (\texttt{l:Atomic\{o;1\} |}$$
$$\texttt{s[end o;] | t[val x=l.get(); end x;]})$$

In this case, the residual `t[end o;]` is executed in the context

$$\langle \texttt{s!o} \rangle\ \langle \texttt{s!l:1} \rangle\ \langle \texttt{t!l:2} \rangle\ [\![-]\!]$$

and therefore is not stuck.

Example `C1` from the introduction can be rewritten to use correspondences as follows, where `p:C1'`

```
class C1' { Object f; Object g; }
s[p.f=new Object(); p.g=new Object();] |
t[val y=p.g; val x=p.f;
  if(y!=null){end y; if(x==null){†}}]
```

In `C1'`, the end assertion can get stuck, since there is no happens-before edge from `s` to `t`.

Example `C2` can also be written in this form, where `p:C2'`

```
class C2' { Object f; final Atomic g; }
s[p.f=new Object(); p.g.set(new Object()));] |
t[val y=p.g.get(); val x=p.f;
  if(y!=null){end y; if(x==null){†}}]
```

In `C2'`, the end assertion cannot get stuck, and statement † is unreachable. If the reads of `t` occur in the reverse order, then `t` could become stuck and statement † could be unreachable.

3. Speculation

A substantial technical hurdle in proving soundness of any type system for the JMM is the treatment of speculation. We first describe our semantics for speculation, then discuss the implications for the proof. See our previous paper [Jagadeesan, Pitcher, and Riely 2010] for details.

While the JMM is clearly designed with type safety in mind, it does not lend itself to standard proofs using subject reduction. The JMM validates an execution using a sequence of *commitments* and a corresponding sequence of *validating executions:* a commitment C is a set of actions resolving some read-write data races; in a sequence of valid executions, execution E_i must justify C_i using C_{i-1}, subject to various coherence criteria. The coherence criteria for the JMM are notoriously subtle, and various revisions have been proposed [Cenciarelli, Knapp, and Sibilio 2007; Sevcík and Aspinall 2008; Sevcík 2008]. These coherence criteria complicate the normal inductive argument for soundness, perhaps accounting for the fact that no soundness proof has been published as yet (to the best of our knowledge).

For our soundness proof, we adopt the semantics proposed in our previous work, which handles speculation quite differently. Our semantics allows every execution allowed by the JMM, with the exception of some programs that contain both synchronization actions and data races. As a result of these subtle differences, our semantics validates many common program transformations that the JMM was meant to validate, but does not.

In addition to the mechanisms present in the previous section for fields and atomics, the semantics allows *single threaded action rewriting* and *speculation*.

Single threaded action rewriting is easy to understand. We allow dynamic transformations to the action sequence generated by a single thread, as long as this does not introduce new behaviors. For example, it is permitted to rewrite $\langle$`s!p.f=42`$\rangle\langle$`s!p.f=27`$\rangle$ to $\langle$`s!p.f=27`$\rangle$, removing the value 42, which may be visible to concurrent threads. The converse transformation is not sound, however, since it introduces the value 42 out of thin-air.

Speculation is far more subtle. Let "$\uparrow$x" stand for `return x` in the following example.

$$\texttt{s[val x=p.g; p.f=27; }\uparrow\texttt{x] | t[val y=p.f; p.g=y;]}$$

In any sequentially consistent execution, `s` must read the initial value 0 from `p.g`, resulting in `s[`$\uparrow$`0]`. A speculative semantics also allows the result `s[`$\uparrow$`27]`. In an implementation, such an execution may be the result of cache effects or optimizations that reorder the independent statements of `s`.

The basic idea of speculation is to allow `s` to read the value 27 from `p.g` before it is written by `t`. Clearly, we are playing with knives. It is common to impose two sanity checks on speculation. First, it should not create data out of "thin air". Second, any program without data races should only have sequentially consistent executions. Our semantics guarantees both properties by restricting valid speculations. We refer to our previous paper for full details, providing only a flavor of the ideas here.

Speculation is not performed actively *by* a process. Rather, a speculation happens *to* a process. Speculation is performed, nondeterministically, by the reduction relation itself. For example, let A be the original pair of threads given above. We have the following "reduction," which speculates that `t` will write 27 to `p.g`.

$$A \;\rightarrow\; \big((\top\Rightarrow A) \;[\!]\; (\langle\texttt{t?p.g=27}\rangle \Rightarrow A)\big)$$

Speculative "actions" are marked with a query (`?`), whereas concrete action are marked with a bang (`!`). The reduction creates two copies of the original process, which are executed in separate universes with separate copies of the state. The left copy is called the *initial* process; the right, the *final* process.

The initial and final processes evolve independently. Within the final process, any thread but `t` can read the speculative value. This allows reduction to the following.

$$\begin{array}{r}\top\Rightarrow \langle\texttt{s!p.f=27}\rangle\,\langle\texttt{t!p.g=27}\rangle\,(\texttt{s[}\uparrow\texttt{0]}\;\;\texttt{| t[]})\\ [\!]\;\langle\texttt{t?p.g=27}\rangle\Rightarrow \langle\texttt{s!p.f=27}\rangle\,\langle\texttt{t!p.g=27}\rangle\,(\texttt{s[}\uparrow\texttt{27] | t[]})\end{array}$$

Actions performed by both processes are allowed to trickle out of the speculation, in order. In addition, object denotations and threads can move in and out of the speculation, as long as there are identical copies on both sides. In the case of our example, this results in

$$\langle\texttt{s!p.f=27}\rangle\,\langle\texttt{t!p.g=27}\rangle\left(\begin{array}{r}\top\Rightarrow(\texttt{s[}\uparrow\texttt{0]}\;\;\texttt{| t[]})\\ [\!]\;\langle\texttt{t?p.g=27}\rangle\Rightarrow(\texttt{s[}\uparrow\texttt{27] | t[]})\end{array}\right).$$

When a speculation occurs in a context with a matching concrete write, the speculation may be removed, leaving only the final process; the initial process is used only to justify the action that removes the speculation. This is called *finalization*. For our running example, the result is

$$\langle\texttt{s!p.f=27}\rangle\,\langle\texttt{t!p.g=27}\rangle\,(\texttt{s[}\uparrow\texttt{27] | t[]}).$$

A *valid* execution is one in which every speculation can be finalized; non-valid executions may create values out of thin air, and, therefore, are not interesting. Nondeterminism ensures that the reduction relation captures all valid executions.

The soundness theorem establishes that typed programs cannot become stuck and therefore that every `end p` occurs in a context such that, for some s, $\langle s\texttt{!p}\rangle$ happens-before the context hole.

Soundness only applies to valid executions. The difficulty is to find an invariant that holds in the face of partial executions which have no valid extension. Such partial executions are caused by speculations which cannot be finalized, which we refer to as *traps*.

If a trap is the result of shape error, our type system can safely treat the final branch as untyped: we simply "turn off" typing in worlds that we know can never be exposed at top level.

Our type system includes effects, which formalize happens-before relations using correspondence assertions. Thus, in addition to shape errors, there are also *subtle* traps, such as speculations that occur without a thread holding appropriate locks. Our proof addresses this subtlety by being safely optimistic about effects: if the optimism is justified, then it has done no harm; if the optimism is not justified, then the speculation must be a trap. In either case, subject reduction holds. For further discussion, see Section 7.1.

It may be argued that we should simply stop the reduction relation from generating unfinalizable speculations. In general, however, unfinalizability is undecidable. Of course, we can rule out certain classes of speculation which are clearly unfinalizable, including shape errors and locking errors. However, we believe that in order to know that a speculation is unfinalizable, you must allow the speculation to occur, and then *prove* that it cannot be finalized.

4. Static approximation of happens-before

Recall from Section 2 that an occurrence of an object reference is *correctly published* if an end statement using that occurrence cannot become stuck. An end statement that cannot become stuck in any execution is *safe*. We introduce typing annotations to statically approximate correct publication.

Correct publication is a predicate on *occurrences* of object references, not on objects themselves: threads may have different happens-before relations to the begin action that marks the creation of an object. A copy of a correctly published reference need not itself be correctly published. In particular, correct publication may be lost when a reference is copied via a shared object. Communication using an atomics preserves correct publication, since an atomic read is related by happens-before to every write to the same atomic. However, communication using a non-atomic does not necessarily preserve correct publication.

A field or variable annotated as *reliable* (notation $\boxast$) must hold only correctly published references. Annotations may be attached to any field or variable, including method formal paramaters, the implicit parameter `this`, and return values. The key to safely approximating correct publication in the type system is to ban read-write data races on reliable fields. We consider three mechanisms for doing so: final fields, atomics and lock-protected fields. We discuss these three primitives in more detail below. In the following subsections, we discuss larger data structures using these primitives.

Finals fields. Final fields have a special semantics in the JMM and can be used to communicate correct publication. If field f is final and reliable, then $p.f$ is correctly published if the occurrence of p is correctly published.

Atomics. We take `Atomic` to be primitive in the dynamics, and provide it an enriched signature in the statics to indicate that it preserves happens-before relations. We first present a simplified version of the signature for `Atomic`.

```
class Atomic { ⊛ Object get();
               void set(⊛ Object v); ... }
```

In the case of get, the reliable annotation indicates that a client program can safely end on the returned value. In the case of set, the annotation requires that all values written to an atomic be correctly published. This restriction may be undesirable in some cases, and therefore we introduce polymorphism over annotations. The final form of the atomic interface is as follows.

```
class Atomic⟨χ⟩ { χ Object get(□this;);
                  void set(□this;χ Object v);  ... }
```

Here χ is an annotation variable, which may be instantiated to $\boxast$ or $\square$, indicating whether a variable is reliable. Reliable variables must hold only correctly published values, whereas variables that are not reliable variables may hold any value.

It is always safe to forget that a value is correctly published, thus losing the capability to end. Annotations therefore give rise to a natural form of subtyping with $⧆ <: \chi <: \square$.

In general, annotations are required on the implicit parameter `this`. Here the annotations indicate that correctly published values may be recovered from a `Atomic` that is not itself correctly published, that is, `this` is marked $\square$.

We elide $\square$ annotations except when they occur as actual parameters to generic classes. We also elide the `this` annotation in method declarations when it is $\square$`this`.

To typecheck example `C2'` from the end of Section 2 in our system, we must specify that the atomic held in `g` is reliable. The correct declaration of `C2'` is as follows.

```
class C2'' { Object f; final Atomic⟨⧆⟩ g; }
s[p.f=new Object(); p.g.set(new Object());] |
t[val y=p.g.get(); val x=p.f;
  if(y!=null){end y; if(x==null){†}}]
```

In this example, there are two classes of executions: those in which `t` gets null from `p.g`, and those in which `t` gets the object set by `s`. In the former case, the annotations on `Atomic` say nothing. In the latter case, the annotations indicate that the creation of the parameter to set happens-before the return of get. That is, happens-before relations are preserved by `Atomic` from a set to the *corresponding* get.

Although we take `Atomic` to be a primitive, one can imagine varying implementations. Implementing `Atomic` by simply reading and writing a field would violate the interface given above, since such an implementation would allow a write before a set to race against a read after the corresponding get. However, an implementation that always returns null would satisfy the interface trivially. More subtly, so would an implementation that created a fresh object for each get, or always returned the first object passed to set. In both cases, the happens-before relation that results is not the expected one. In the former case, we have that the body of get happens-before the return of get. In the latter case, we have that the first call to set happens-before the return of every subsequent call to get. In short, the annotations say nothing about behavior of `Atomic` as a *data structure*. As a data structure, `Atomic` requires that after the first set, every get action returns the value from the "previous" set. Such requirements are readily understood informally, but difficult to formalize in a way that appeals to nonspecialists.

Our goal has been to define a typing system that captures interesting properties of happens-before without commenting on data structure invariants. Our solution is to leave the notion of "corresponding actions" in the dynamics, using object identity. The resulting type system can be used alone or as a module in a larger system with additional tools to establish data structure invariants and confinement properties. Used alone, the statics provide conditional guarantees of properties of interest, such as the reachability of statement †. *If* values match up *and* the necessary confinement properties hold *then* the properties of interest can be inferred from the happens-before relation.

Lock-protected fields. Communicating correct publication on fields which are neither final nor atomic is rather more delicate, as one must guarantee the absence of read-write data races programmatically. The use of locks to avoid races has been studied by Abadi, Flanagan, and Freund [2006]. We adapt the simplest of their systems to our setting by allowing write-write data races on all fields and read-write races on fields that are not reliable. Consider the following locking variant, where `p:C3` and `l:Lock`. (Here synchronized statements do not create scopes, and therefore `y` remains visible.)

```
class C3 { Object f; ⧆Object g guardedby l; }
s[p.f=new Object();
  synchronized(l){p.g=new Object();}] |
t[synchronized(l){val y=p.g;} val x=p.f;
  if(y!=null){end y; if(x==null){†}}]
```

Just as in example `C2''` above, the end assertion is safe, and statement † is unreachable. In this case, the happens-before relation is carried on lock `l`. The type of `g` indicates that it is a conservative field guarded by `l`. A guarded field may only be accessed in a context that holds the corresponding lock, thus ensuring the necessary happens-before relations from lock acquisition to field read and from field write and lock release.

In general, we allow reads and writes to be guarded by different boolean combinations of locks, requiring that any set of locks satisfying the read guard must *overlap* with every set of locks satisfying the write guard. (Overlap is defined in Section 5.2 to capture the intuition of non-disjoint intersection.) This disallows read-write data races while still allowing write-write data races. Consider an example from Sevcík [2008] with write-write data races, but no read-write data races, where `p:D`, `l:Lock`, and `k:Lock` are shared.

```
class D { ⧆Object f rdguard (l && k) wrguard (l || k); }
s[synchronized (l){p.f=new Object();}] |
t[synchronized (k){p.f=new Object();}] |
u[synchronized (l){
  synchronized (k){⧆Object x=p.f;}}]
```

The annotations indicate that the read of `p.f` requires both locks `l` and `k`, whereas the write of `p.f` requires only one of them. This program typechecks in our system. While this example is contrived, the separate treatment of read and write capabilities on objects is critical to the design of efficient data structures to avoid unnecessary sequentialization (see Java's concurrent hash tables).

4.1 Dummy variables

Our technique uses object identity to reason about dynamic occurrences of program points via correspondence assertions. Occasionally, however, interfaces lack appropriate variables to annotate the happens-before relations of interest. In such cases, we require that the interface be augmented with *dummy variables*, which can be systematically erased at runtime. For example, consider the following methods of `Atomic`, with candidate annotations. (Recall that we elide the annotation for the implicit parameter `this` from method types when annotation is $\square$.)

```
class Atomic⟨χ⟩ { ...
 χ Object getAndSet(χ Object v);
 boolean compareAndSet(Object oldv, χ Object newv); }
```

Using the data structure invariants, the annotation of the `getAndSet` method captures exactly the right property: a return from `getAndSet` happens-after the previous set, and a call to `getAndSet` happens-before the subsequent get. The `compareAndSet` method has the same memory effect as `getAndSet`, however, its annotation only gives half of the story: the return value is unannotated. The situation can be remedied using a dummy variable in the signature of `compareAndSet`.

```
Pair⟨boolean, χ Object⟩
       compareAndSet(Object oldv, χ Object newv);
```

If `compareAndSet` returns false, the dummy variable returns null; otherwise, the dummy variable returns the previous value stored by the atomic, as in `getAndSet`.

The unannotated interface of `BinaryLatch` (adapting Java's `CountDownLatch`) is:

```
class BinaryLatch { void await(); void countDown(); }
```

The documented memory effect is that a return of `await` happens-after the corresponding call to `countDown`. Applying our methodology leads to the interface

```
class BinaryLatch⟨χ⟩ { χ Object await();
                       void countDown(χ Object v); }
```

where `await` returns the value passed to the last call to `countDown`. Similar changes can be made systematically throughout the APIs.

```
class Condition⟨χ⟩ { χ Object await();
                     void signal(χ Object); }
class Semaphore⟨χ⟩ { χ Object acquire();
                     void release(χ Object v); }
```

The annotations on `Condition` indicate the memory effect that a return of `await` happens-after the corresponding call to `signal`, and similarly for the (binary) `Semaphore`.

4.2 Building data structures

From the primitive constructs with special memory effects—final fields, atomic fields, and lock-protected fields—it is possible to construct useful concurrent data structures. The happens-before relations created by these data structures are key to concurrent programing. Indeed, *every* data structure provided by the `java.util.concurrent` API specifies such memory consistency effects. Our typing annotations are a lightweight mechanism for making these memory effects amenable to the static analysis.

For example consider `BlockingQueue` from `java.util.concurrent`. The Java APIs specify the behavior of a `BlockingQueue` in two parts.

- The *data structure* specification states that there is a one-to-one correspondence between invocations of the `take` and the `put` methods, such that a `take` returns the `Object` that is added into the data structure by its *corresponding* `put`.
- The *memory consistency* specification states that "actions in a thread prior to placing an object into a `BlockingQueue` happen-before actions subsequent to the access or removal of that element from the `BlockingQueue` in another thread" .

In our system, the `BlockingQueue` is annotated as follows.

```
class BlockingQueue⟨χ⟩ { void put(χ Object o);
                         χ Object take(); ... }
```

As with `Atomic`, the annotations do not capture the data structure invariant, and only very roughly approximate the memory consistency specification. Nevertheless, they provide valuable information. Consider the following variant of our running example, where `p:C4` and `q:BlockingQueue⟨⧆⟩`.

```
class C4 { Object f; Object g; }
s[p.f=new Object(); p.g=new Object();
  q.put(new Object());] |
t[val z=q.take(); val y=p.g; val x=p.f;
  end z; if((y!=null)&&(x==null)){†}]
```

The call to `take` blocks until `put` begins execution. Assuming that `q` is initially empty and that there are no other threads, then `t` must take the object that is put by `s`. In this case, the ⧆ annotation tells us that the call to put happens-before the return of get. By thread-local reasoning, we can conclude that the writes by `s` must happen-before the call to put, and also that the reads by `t` must happen-after the return of take. Reasoning by transitivity, we can conclude that statement † is unreachable.

Formally, our system allows the inclusion of "`end z`" in thread `t`, indicating that the call of put happened-before the return of take. To reason about the reachability of †, one must also reason about thread-local happens-before relations and the confinement `q` and `p`. Our system captures the crucial *cross-thread* reasoning required about happens-before, leaving the local reasoning to informal methods or modular formal methods.

In the rest of this section, we present both non-blocking and lock-based examples which typecheck in our system, providing typed implementations of the interfaces discussed above.

4.3 Non-blocking algorithms

The basic building block for writing non-blocking algorithms is the `Atomic` class. For example, one can implement the `BinaryLatch` interface using an `Atomic`.

```
class BinaryLatch⟨χ⟩ {
  final Object init;
  final Atomic⟨χ⟩ witness = new Atomic();
  BinaryLatch() {
    witness.set(new Object());
    init = witness.get(); }
  χ Object await()  {
    while (witness.get() == init) ;
    return witness.get(); }
  void countDown(χ Object o) {
    witness.compareAndSet(init, o); } }
```

Informally, one can reason that the return of `await` must happen-after the call to `compareAndSet` which must happen-after the corresponding call to `countDown`. Thus, the implementation validates the interface of `BinaryLatch`.

This guarantee is established formally by typechecking the code in our system. The type system uses the `Atomic` interface to deduce that the return value of `await` may be annotated by χ. The body of `countDown` typechecks because the second argument of `compareAndSet` carries annotation χ. The set of the witness in the constructor typechecks because newly created references are viewed as initially created with a ⧆ annotation.

Our type system can validate Treiber's algorithm for concurrent stacks, from [Goetz, Peierls, Bloch, Bowbeer, Holmes, and Lea 2005], with the following interface.

```
class ConcurrentStack⟨χ⟩ { χ Object pop();
                           void push(χ Object v); }
```

When `ConcurrentStack⟨⧆⟩` satisfies the usual data structure invariant, the absence of a write-read data race on the ⧆ reference transmitted from the `push` to the `pop` establishes the required happens-before edge between a return of `pop` and the corresponding call of `push`. This annotated interface is validated by the following implementation. (We elide the dummy return value of `compareAndSet`.)

```
class Node⟨χ⟩ {
  final χ Object item;
  final ⧆ Node⟨χ⟩ next;
  Node(χ Object item,⧆ Node⟨χ⟩ next) {
    this.item = item; this.next = next; } }
class ConcurrentStack⟨χ⟩ {
  final Atomic⟨⧆⟩ top = new Atomic⟨⧆⟩();
  void push(χ Object item) {
    ⧆ Node⟨χ⟩ oldHead;
    ⧆ Node⟨χ⟩ newHead;
    do { oldHead = top.get();
         newhead = new Node(item,oldHead);
    } while (!top.compareAndSet(oldHead, newHead)); }
  χ Object pop() {
    ⧆ Node⟨χ⟩ oldHead;
    ⧆ Node⟨χ⟩ newHead;
    do { oldHead = top.get();
```

```
        if (oldHead == null) return null;
        newHead = oldHead.next;
    } while (!top.compareAndSet(oldHead, newHead));
    return oldHead.item; } }
```

The annotation parameter of the ConcurrentStack (and Node) classes matches that on the items that are stored in the stack. On the other hand, the code maintains the invariant that all references to the nodes on the stack carry $\boxast$ annotation. This invariant is used to satisfy the requirement that the second arguments of the compare-AndSet carry $\boxast$ annotation. In order to maintain the invariant at the creation of new stack nodes in the push method, the Node.next field must be final.

4.4 Programs with locks

Our locks are based on Java locks, with some different conventions to simplify the semantics. Synchronization can only be performed using block-based synchronization on an instance of the Lock class. We take both Lock and the associated Condition class to be primitives; however, one can safely think of these as being implemented using atomic objects.

Locks can be used to validate the following implementation of the enriched interface for binary semaphores discussed above.

```
class Semaphore⟨χ⟩ {
  final Lock l = new ReentrantLock();
  final Condition⟨χ⟩ open = new Condition(l);
  χ Object witness guardedby l;
  χ Object acquire() {
    χ Object temp;
    synchronized(l) {
      while (witness == null) {
        witness = open.await(); }
      temp = witness;
      witness = null; }
    return temp; }
  void release(χ Object o) {
    synchronized(l) {
      if (witness == null) {
        witness = o; open.signal(o); } } }
```

The required guarantee follows from typing. The witness field is guarded by lock l, and our system ensures that all accesses to witness happen only when this lock has been acquired. The assignment to witness in acquire typechecks because of the Condition interface. In typechecking countDown, the Condition interface forces the argument of release to have χ annotation. The two cases of control flow for the witness returned by acquire are merged via the annotations on witness and temp and their typechecking.

5. Typing classes

We present the syntax and type rules for classes, using the annotations described in the previous section. In the next section, we describe the safety theorem: starting from a *well-typed* class table, (small step) *evaluation* of the *bootstrap process* cannot get *stuck*, and therefore, in particular, every end must always happen-after the creation of the corresponding object.

The statement of safety requires that we define well-typed classes, the evaluation relation, the bootstrap process and the stuck predicate. The proof of safety follows by defining well-typed *processes*, demonstrating that the bootstrap process is well-typed, and proving preservation and progress.

To simplify the semantics, we use a more restrictive language than that of the previous section. These changes are made only for conciseness of exposition: they do not alter the expressive power of the language or its type system. We make all local variables immutable and implement iteration via recursion. We make all sequencing explicit; for example `return this.f.g` must be written `val x=this.f; return x.g`.

The operational semantics follows our previous paper [Jagadeesan, Pitcher, and Riely 2010], which is in turn based on the JMM. In order to capture the properties of interest, we have enriched our previous semantics to include final fields, as well as primitive support for the Atomic and Condition classes. Unlike our previous work, we require block-structured use of Locks, which simplifies typing. Since local variables are immutable, we add a return value to these synchronization blocks, so that they can provide results to the thread without requiring the use of a mutable field.

5.1 Syntax

Let x and y range over variable names (including the reserved variable this), d over class names (including the reserved classes Object, Atomic, Lock and Cond), f and g over field names, and m over method names (including the reserved methods start and run).

Class types may be parameterized by annotations. Let χ range over annotation variables and γ range over the ground annotations ($\gamma ::= \boxast \mid \square$). An *annotation*, a, b, c, is either an annotation variable or a ground annotation ($a,b,c ::= \chi \mid \gamma$). Class types, D, have the form $d\langle\vec{a}\rangle$. A type may be a class type or a base type ($T,S ::= D \mid bt$). We routinely drop empty type brackets $\langle\rangle$.

Let bv range over base values (including integers and the constants unit, true and false). The set of ground values, v, w, u, includes null and the base values. (For running processes, the set of ground values will also include object names.) Open values include ground values and variables ($V,W,U ::= v \mid x$). Let op range over base value operators (such as ==, +, &&). We assume that null has meaning at all base types (zero for integers, false for booleans). The statement language is as follows:

$$\begin{array}{rll}
M,N ::= & \texttt{return}\ V\texttt{;} & \text{(Return statement)}\\
\mid & \texttt{val}\ x\ \texttt{=}\ \texttt{new}\ D\texttt{(}\vec{V}\texttt{)}\texttt{;}\ M & \text{(Creation statement)}\\
\mid & \texttt{val}\ x\ \texttt{=}\ W\texttt{.}m\texttt{(}\vec{V}\texttt{)}\texttt{;}\ M & \text{(Method statement)}\\
\mid & \texttt{val}\ x\ \texttt{=}\ op\texttt{(}\vec{V}\texttt{)}\texttt{;}\ M & \text{(Operator statement)}\\
\mid & \texttt{val}\ x\ \texttt{=}\ \texttt{(}D\texttt{)}V\texttt{;}\ M & \text{(Cast statement)}\\
\mid & \texttt{end}\ V\texttt{;}\ M & \text{(End statement)}\\
\mid & \texttt{val}\ x\ \texttt{=}\ V\texttt{.}f\texttt{;}\ M & \text{(Field read statement)}\\
\mid & V\texttt{.}f\ \texttt{=}\ W\texttt{;}\ M & \text{(Field write statement)}\\
\mid & \texttt{if}\ \texttt{(}V\texttt{)}\ \texttt{\{}M\texttt{\}}\ \texttt{else}\ \texttt{\{}N\texttt{\}} & \text{(Conditional statement)}\\
\mid & \texttt{val}\ x\ \texttt{=}\ \texttt{sync}\ V\ \texttt{\{}N\texttt{\}}\ M & \text{(Synchronization statement)}
\end{array}$$

As discussed above, synchronization statements allow a return value. All the other constructs are standard.

We often drop the keyword val, use complex expressions and infix notation for operators and drop instances of "`return unit;`". Thus, "`y = a+b+c;`" is sugar for "`val x = +(a,b); val y = +(x, c); return unit;`", where x is fresh. We write "val x = $\cdots$; M" as "$\cdots$; M" if x does not occur free in M. We write $\uparrow V$ for "return V;" and $\uparrow(V,W)$ for "val x = new Pair(V,W); return x;", where x is fresh. We write "if (V){val x = $\cdots$;} M" for "if(V) {val x = $\cdots$; M} else {M}" if x does not occur free in M; this notation extends to field write statements, conditional statements and sequences of statements in the obvious way.

The variable x is bound with scope M in all statements of the form "val x = $\cdots$; M". For any syntax category, let *fv* return the set of free variables and let *fn* return the set of free names.

Class definitions have the following form.

$$\begin{array}{rll}
\mathscr{D} ::= & \texttt{class}\ d\langle\vec{\chi}\rangle\texttt{\{}\vec{\mathscr{F}}\,\vec{\mathscr{M}}\texttt{\}} & \text{(Class declaration)}\\
\mathscr{M} ::= & a\,T\ m\texttt{(}b\,\texttt{this}\texttt{;}\vec{a}\,\vec{T}\ \vec{x}\texttt{)}\texttt{\{}M\texttt{\}} & \text{(Method declaration)}\\
\mathscr{F} ::= & T\,f\texttt{;}\ \mid\ \texttt{final}\ a\,D\,f\texttt{;} & \text{(Field declaration)}
\end{array}$$

$$\begin{aligned} & \mid a D f \ \texttt{rdguard}\ \Phi\ \texttt{wrguard}\ \Psi\texttt{;} \\ \Phi,\Psi &::= \texttt{true} \mid F \mid \Phi\texttt{\&\&}\Psi \mid \Phi\texttt{||}\Psi && \text{(Guard)} \\ F,G &::= V \mid F.f && \text{(Field path)} \end{aligned}$$

There are field declaration forms for normal, final and lock-protected fields. Annotated fields must have object types. Annotated base types are useless since `end` requires an object type. Lock-protected fields have read and write guards, which are boolean combinations of field paths. When typing classes, all field paths will have the form `this.f1.f2.`$\cdots$`.fn`. We elide read or write guards when uninteresting and write `guardedby` Φ as shorthand for `rdguard` Φ `wrguard` Φ. We elide the $\Box$ annotation in variable and field declarations, and $\Box$`this` in method declarations.

The annotation variables $\vec{\chi}$ are bound in the declaration `class` $d\langle\vec{\chi}\rangle\texttt{\{}\vec{\mathscr{F}}\vec{\mathscr{M}}\texttt{\}}$, with scope $\vec{\mathscr{F}}$ and $\vec{\mathscr{M}}$. We identify syntax up to renaming and write $\mathscr{F}\{\!\![{}^{\vec{a}}/_{\vec{\chi}}]\!\!\}$ for the capture avoiding substitution of $\vec{a}$ for $\vec{\chi}$ in $\mathscr{F}$, and likewise for $\mathscr{M}$.

Suppose d is declared as `class` $d\langle\vec{\chi}\rangle\texttt{\{}\vec{\mathscr{F}}\vec{\mathscr{M}}\texttt{\}}$. Define the partial function *ftype* so that $\mathit{ftype}(d\langle\vec{a}\rangle.f) = \texttt{final}\ aT$ if $\vec{\mathscr{F}}\{\!\![{}^{\vec{a}}/_{\vec{\chi}}]\!\!\}$ contains the declaration "$\texttt{final}\ aT\ f\texttt{;}$", and likewise for racing and lock-protected fields. Define the partial function *mtype* so that $\mathit{mtype}(d\langle\vec{a}\rangle.m) = b;\vec{a}\vec{T} \to aT$ if $\vec{\mathscr{M}}\{\!\![{}^{\vec{a}}/_{\vec{\chi}}]\!\!\}$ contain the declaration $aT\ m(b\texttt{this}\texttt{;}\vec{a}\vec{T}\ \vec{x})\texttt{\{}M\texttt{\}}$. Define $\mathit{finals}(D) = \vec{f}$ if $\vec{f}$ are the final fields of class D, and similarly *nonfinals*.

5.2 Typing

Typing environments have the following form.

$$E ::= E,\chi \mid E,x{:}aT \mid E,\texttt{final}\ x{=}F \mid E,\texttt{lock}\ V$$

We identify environments up to reordering and repetition. Supporting judgments such as well-formed environment ($E \vdash \diamond$), type ($E \vdash T$), and annotation ($E \vdash a$) simply account for free and bound annotation variables. These definitions are elided.

Guard satisfaction is a propositional logic over field paths, with substitution of final paths.

$$\frac{E \ni \texttt{lock}\ F}{E \Vdash F} \qquad \frac{E,E'\{\!\![{}^{F}/_{x}]\!\!\} \Vdash \Phi}{E,x{:}aT,\texttt{final}\ x{=}F,E' \Vdash \Phi}$$

$$\frac{}{E \Vdash \texttt{true}} \qquad \frac{E \Vdash \Phi \quad E \Vdash \Psi}{E \Vdash \Phi\texttt{\&\&}\Psi} \qquad \frac{E \Vdash \Phi}{E \Vdash \Phi\texttt{||}\Psi} \qquad \frac{E \Vdash \Psi}{E \Vdash \Phi\texttt{||}\Psi}$$

Intuitively, two guards *must overlap* if the sets of field paths that satisfy them must always have a non-disjoint intersection. The formal definition is a safe syntactic approximation of this intuition: Let $\mathbb{F}$ and $\mathbb{G}$ range over sets of field paths. Then any guard Φ can be written in disjuntive normal as $\bigvee_i \mathbb{F}_i$, where each set $\mathbb{F}_i$ is viewed as a conjunction of field paths. We say that Φ *must overlap with* Ψ if $\Phi = \bigvee_i \mathbb{F}_i$, $\Psi = \bigvee_j \mathbb{G}_j$, and $\mathbb{F}_i \cap \mathbb{G}_j \neq \emptyset$ for all i, j.

Class ($\vdash \mathscr{D}$), method ($E \vdash \mathscr{M}$ in D) and field declarations ($E \vdash \mathscr{F}$) are as follows.

$$\frac{\forall i.\ \vec{\chi} \vdash \mathscr{F}_i \qquad \forall j.\ \vec{\chi} \vdash \mathscr{M}_j \text{ in } d\langle\vec{\chi}\rangle}{\vdash \texttt{class}\ d\langle\vec{\chi}\rangle\texttt{\{}\vec{\mathscr{F}}\vec{\mathscr{M}}\texttt{\}}}$$

$$\frac{E,\texttt{this}{:}bD,\vec{x}{:}\vec{a}\vec{T} \vdash M : aT}{E \vdash aT\ m(b\texttt{this}\texttt{;}\vec{a}\vec{T}\ \vec{x})\texttt{\{}M\texttt{\}} \text{ in } D}$$

$$\frac{E \vdash T}{E \vdash T\ f\texttt{;}} \qquad \frac{E \vdash a \quad E \vdash D}{E \vdash \texttt{final}\ aD f\texttt{;}}$$

$$\frac{E \vdash a \quad E \vdash D \quad E \vdash \Phi,\Psi \quad \Phi \textit{ must overlap with } \Psi}{E \vdash aD f\ \texttt{rdguard}\ \Phi\ \texttt{wrguard}\ \Psi\texttt{;}}$$

These are standard definitions, with annotations added. There are three forms of field declaration, and correspondingly three type rules. The read and write guards of a lock-protected field must overlap.

$$\overset{\text{(VAL-VARIABLE)}}{\frac{E \vdash \diamond \quad E \ni x{:}aT}{E \vdash x : aT}} \qquad \overset{\text{(VAL-NULL)}}{\frac{E \vdash \diamond \quad E \vdash a \quad E \vdash D}{E \vdash \texttt{null} : aD}}$$

$$\overset{\text{(VAL-SUB-TYPE)}}{\frac{E \vdash V : aD}{E \vdash V : a\texttt{Object}}} \qquad \overset{\text{(VAL-SUB-ANN1)}}{\frac{E \vdash V : aT}{E \vdash V : \Box T}} \qquad \overset{\text{(VAL-SUB-ANN2)}}{\frac{E \vdash V : \boxed{\ast} T}{E \vdash V : aT}}$$

$$\overset{\text{(STAT-NEW)}}{\frac{\begin{array}{l}\mathit{finals}(D) = \vec{f} \quad E \vdash \vec{V} : \vec{b}\vec{S} \\ \mathit{ftype}(D.\vec{f}) = \texttt{final}\ \vec{b}\vec{S} \\ E,x{:}\boxed{\ast}D \vdash M : aT\end{array}}{E \vdash \texttt{val}\ x\ \texttt{=}\ \texttt{new}\ D\texttt{(}\vec{V}\texttt{);}\ M : aT}} \qquad \overset{\text{(STAT-METHOD)}}{\frac{\begin{array}{l}E \vdash W : cD \quad E \vdash \vec{V} : \vec{b}\vec{S} \\ \mathit{mtype}(D.m) = c;\vec{b}\vec{S} \to bS \\ E,x{:}bS \vdash M : aT\end{array}}{E \vdash \texttt{val}\ x\ \texttt{=}\ W\texttt{.}m\texttt{(}\vec{V}\texttt{);}\ M : aT}}$$

$$\overset{\text{(STAT-LOCK)}}{\frac{\begin{array}{l}E \vdash V : \Box\texttt{Lock} \quad E,\texttt{lock}\ V \vdash N : bS \\ E,x{:}bS \vdash M : aT\end{array}}{E \vdash \texttt{val}\ x\ \texttt{=}\ \texttt{sync}\ V\ \texttt{\{}N\texttt{\}}\ M : aT}} \qquad \overset{\text{(STAT-RETURN)}}{\frac{E \vdash V : aT}{E \vdash {\uparrow}V : aT}}$$

$$\overset{\text{(STAT-FINAL-READ)}}{\frac{\begin{array}{l}E \vdash V : cD \quad \mathit{ftype}(D.f) = \texttt{final}\ bS \\ E,x{:}b'S,\texttt{final}\ x{=}V.f \vdash M : aT\end{array}}{E \vdash \texttt{val}\ x\ \texttt{=}\ V\texttt{.}f\texttt{;}\ M : aT}} \quad b' = \begin{cases} b, & \text{if } b = c \\ \Box, & \text{otherwise}\end{cases}$$

$$\overset{\text{(STAT-RACING-READ)}}{\frac{\begin{array}{l}E \vdash V : \Box D \quad \mathit{ftype}(D.f) = S \\ E,x{:}\Box S \vdash M : aT\end{array}}{E \vdash \texttt{val}\ x\ \texttt{=}\ V\texttt{.}f\texttt{;}\ M : aT}} \qquad \overset{\text{(STAT-RACING-WRITE)}}{\frac{\begin{array}{l}E \vdash V : \Box D \quad \mathit{ftype}(D.f) = S \\ E \vdash W : \Box S \quad E \vdash M : aT\end{array}}{E \vdash V\texttt{.}f\ \texttt{=}\ W\texttt{;}\ M : aT}}$$

$$\overset{\text{(STAT-GUARDED-READ)}}{\frac{\begin{array}{l}\mathit{ftype}(D.f) = bS\ \texttt{rdguard}\ \Phi \\ E \vdash V : \Box D \quad E \Vdash \Phi\{\!\![{}^{V}/_{\texttt{this}}]\!\!\} \\ E,x{:}bS \vdash M : aT\end{array}}{E \vdash \texttt{val}\ x\ \texttt{=}\ V\texttt{.}f\texttt{;}\ M : aT}} \qquad \overset{\text{(STAT-GUARDED-WRITE)}}{\frac{\begin{array}{l}\mathit{ftype}(D.f) = bS\ \texttt{wrguard}\ \Psi \\ E \vdash V : \Box D \quad E \Vdash \Psi\{\!\![{}^{V}/_{\texttt{this}}]\!\!\} \\ E \vdash W : bS \quad E \vdash M : aT\end{array}}{E \vdash V\texttt{.}f\ \texttt{=}\ W\texttt{;}\ M : aT}}$$

$$\overset{\text{(STAT-END)}}{\frac{\begin{array}{l}E \vdash V : \boxed{\ast}D \\ E \vdash M : aT\end{array}}{E \vdash \texttt{end}\ V\texttt{;}\ M : aT}} \qquad \overset{\text{(STAT-IF)}}{\frac{E \vdash V : \Box\texttt{boolean} \quad E \vdash M : aT \quad E \vdash N : aT}{E \vdash \texttt{if (}V\texttt{) \{}M\texttt{\} else \{}N\texttt{\}} : aT}}$$

Figure 1. Statement typing (rules for base types elided)

The judgments for well-typed values ($E \vdash V : aT$) and statements ($E \vdash M : aT$) are given in Figure 1; we have elided standard rules for base values, operators casting and conditionals.

STAT-NEW creates objects with $\boxed{\ast}$. The final fields of the class must be initialized by the constructor; non-final fields are set to null. STAT-END requires $\boxed{\ast}$ objects.

VAL-SUB-ANN1 and VAL-SUB-ANN2 define subtyping on annotations, with $\boxed{\ast}$ as bottom and $\Box$ as top.

In STAT-LOCK, statement N is typed under the assumption that lock V has been acquired, allowing guard satisfaction in STAT-GUARDED-READ and STAT-GUARDED-WRITE.

For final fields, the annotation on V must agree with that of the field to impart any benefit to the reader; otherwise, the reader sees $\Box$. The rules for racing fields impose no annotation restrictions on the writer, nor do they impart any annotation benefit to the reader. The rules for accessing a lock-protected field $V.f$ require that the field and value agree in their annotation; the annotation on V is irrelevant.

6. Soundness

To validate the soundness of our system, we use a small-step reduction relation. In this section, we explain the operational semantics, define soundness, and discuss highlights of the soundness proof.

There is insufficient space to include all definitions with sufficient motivation, therefore we elide the following (a) synchronization statements (the interesting issues are already revealed by atomics); (b) discussion of auxiliary definitions used in the operational semantics; (c) most of the additional typing rules used to prove subject reduction; and (d) all proofs. These details are included in the full version of the paper.

6.1 Operational semantics

Let p, q, s, t and ℓ range over object names; by convention, we use name metavariables s,t for thread objects and ℓ for atomic objects. Process syntax is as follows, where name p is bound with scope A in the process $(\nu p)A$.

$$\begin{array}{rcll}
v,w,u & ::= & \texttt{null} \mid bv \mid p & \text{(Ground value)}\\
M,N & ::= & \cdots \mid \texttt{val}\ x\ \texttt{=}\ \texttt{\{}N\texttt{\}}\ M & \text{(Statement)}\\
\alpha,\beta & ::= & \langle s\,\texttt{!}\,p\rangle \mid \langle s\,\texttt{!}\,p\,.f\texttt{=}v\rangle \mid \langle s\,\texttt{!}\,\ell\texttt{:}\,j\rangle & \text{(Action)}\\
A,B & ::= & p\texttt{:}D \mid \ell\texttt{:Atomic\{}v\texttt{;}j\texttt{\}} & \text{(Process)}\\
 & & \mid \texttt{free}\ p \mid \texttt{runnable}\ p & \\
 & & \mid \alpha A \mid s\texttt{[}M\texttt{]} \mid A\,|\,B \mid (\nu p)A & \\
 & & \mid \top\Rightarrow A \,[\!]\, \langle s\texttt{?}p\,.f\texttt{=}v\rangle \Rightarrow B &
\end{array}$$

The syntax of ground values and statements are extended to include the necessary runtime constructs. A partially executed method is written "`val` x `=` `{`N`}` M", where N is the (residual of the) method body, and M is the (residual of the) calling context; that is, M is the statement that executes after the method returns.

An action indicates that the creation of an object, a write, or a synchronization has occurred. As a process executes, actions accumulate in the context of a thread. The first two process forms provide denotations for object names. The denotation of an atomic includes the state of the object. The denotation of other objects includes only the class; the values of fields are determined by the action context, which varies over time and between threads. Lifecycle flags (`free` p and `runnable` p) are used to track the lifecycle of an object. Object denotations, lifecycle flags and threads (s`[`M`]`) are composed using restriction ($(\nu p)A$), action prefixing (αA), parallel composition ($A\,|\,B$) and speculation ($\top\Rightarrow A \,[\!]\, \langle s\texttt{?}p\,.f\texttt{=}v\rangle \Rightarrow B$).

In the rest of this subsection, we formalize the reduction relation ($A \to B$). The exposition is necessarily dense, as we do not have space to provide motivation or discussion, which can be found in [Jagadeesan, Pitcher, and Riely 2010]. On first reading, it is best to skim this material, coming back to it as necessary.

Actions include empirical actions α, β, speculative actions $\langle s\texttt{?}p\,.f\texttt{=}v\rangle$, ψ, and thread-labeled context holes $s[\![-]\!]$. Let $\mathit{thrd}(\sigma)$ return the unique thread associated with an action. For write and speculation actions define loc and val to return the location and value of the action as $\mathit{loc}(\langle s\texttt{!}p\,.f\texttt{=}v\rangle) = \mathit{loc}(\langle s\texttt{?}p\,.f\texttt{=}v\rangle) = p\,.f$, and $\mathit{val}(\langle s\texttt{!}p\,.f\texttt{=}v\rangle) = \mathit{val}(\langle s\texttt{?}p\,.f\texttt{=}v\rangle) = v$. Let $\mathit{act}(\mathbb{C},s)$ return the sequence of labeled actions occurring before the hole in $\mathbb{C}$.

$$\begin{array}{ll}
\mathit{act}([\![-]\!],s) = s[\![-]\!] & \\
\mathit{act}(\alpha\,\mathbb{C},s) = \alpha\,\mathit{act}(\mathbb{C},s) & \mathit{act}(\mathbb{C}\,|\,A,s) = \mathit{act}(\mathbb{C},s)\\
\mathit{act}((\nu q)\,\mathbb{C},s) = \mathit{act}(\mathbb{C},s) & \mathit{act}(A\,|\,\mathbb{C},s) = \mathit{act}(\mathbb{C},s)\\
\multicolumn{2}{l}{\mathit{act}(\top\Rightarrow A \,[\!]\, \langle s\texttt{?}p\,.f\texttt{=}v\rangle \Rightarrow \mathbb{C},s) = \langle s\texttt{?}p\,.f\texttt{=}v\rangle\,\mathit{act}(\mathbb{C},s)}\\
\multicolumn{2}{l}{\mathit{act}(\top\Rightarrow \mathbb{C} \,[\!]\, \langle s\texttt{?}p\,.f\texttt{=}v\rangle \Rightarrow A,s) = \mathit{act}(\mathbb{C},s)}
\end{array}$$

We say that context $\mathbb{C}$ *enables* $\langle s\texttt{!}\ell\texttt{:}\,j\rangle$ if $j=0$ or $\langle t\texttt{!}\ell\texttt{:}\,j{-}1\rangle \in \mathit{act}(\mathbb{C},s)$, for some t. We say that $\mathbb{C},s$ is *contiguous* if whenever $\mathbb{C} = \mathbb{C}'[\![\langle t\texttt{!}\ell\texttt{:}\,j\rangle\mathbb{C}'']\!]$ for $j>1$ then there exists t' such that $\langle t'\texttt{!}\ell\texttt{:}\,j{-}1\rangle \in \mathit{act}(\mathbb{C}',s)$. We define several notions for contiguous contexts. Suppose $\mathbb{C},s$ is contiguous.

Define *program order* ($<^{\mathbb{C},s}_{po}$) and *synchronizes-with* ($<^{\mathbb{C},s}_{sw}$) as relations on the indices of $\vec{\sigma}$, where $\vec{\sigma} = \mathit{act}(\mathbb{C},s)$.

$$\begin{array}{lll}
i <^{\mathbb{C},s}_{po} j & \text{iff} & i<j \text{ and } \mathit{thrd}(\sigma_i) = \mathit{thrd}(\sigma_j)\\
i <^{\mathbb{C},s}_{sw} j & \text{iff} & \sigma_i = \langle s\texttt{!}\ell\texttt{:}n{-}1\rangle \text{ and } \sigma_j = \langle t\texttt{!}\ell\texttt{:}n\rangle\\
 & & \text{for some } s,\,t,\,\ell \text{ and even } n
\end{array}$$

Define happens-before order ($<^{\mathbb{C},s}_{hb}$) to be the transitive closure of the union of program order and synchronizes-with.

Suppose $\mathbb{C},s$ is contiguous. Let $\vec{\sigma} = \mathit{act}(\mathbb{C},s)$. Let k be the index of $s[\![-]\!]$ in $\vec{\sigma}$.

- Define $\mathbb{C},s$ *justifies end* p if there exists i and t such that $\langle t\texttt{!}p\rangle = \sigma_i$ and $i <^{\mathbb{C},s}_{hb} k$
- Define $\mathbb{C},s$ *justifies read* $p\,.f\texttt{=}v$ if all of the following hold for some i.
  - $\sigma_i = \langle t\texttt{!}p\,.f\texttt{=}v\rangle$ for some t (possibly equal to s), or $\sigma_i = \langle t\texttt{?}p\,.f\texttt{=}v\rangle$ for some $t \neq s$; and
  - for every j such that σ_j is a write action and $i <^{\mathbb{C},s}_{hb} j <^{\mathbb{C},s}_{hb} k$, we have that $\mathit{loc}(\sigma_j) \neq \mathit{loc}(\sigma_i)$.
- Define $\mathbb{C},s$ *justifies speculation* $p\,.f\texttt{=}v$ if all of the following hold for some i.
  - $\sigma_i = \langle s\texttt{!}p\,.f\texttt{=}v\rangle$;
  - for every j such that σ_j is a write action and $i <^{\mathbb{C},s}_{hb} j <^{\mathbb{C},s}_{hb} k$, we have that $\mathit{loc}(\sigma_j) \neq \mathit{loc}(\sigma_i)$; and
  - for every j such that σ_j is a release action and $i <^{\mathbb{C},s}_{hb} j <^{\mathbb{C},s}_{hb} k$, we have that $\mathit{thrd}(\sigma_j) \neq \mathit{thrd}(\sigma_i)$.

Define the structural order $A \geqq B$ to be the least precongruence on processes that satisfies the axioms in Equation 6.1 (where $A \doteqdot B$ abbreviates the two axioms $A \geqq B$ and $B \geqq A$). Define $\equiv$ to be the kernel of $\geqq$.

As usual, the structural order allows processes to move around in the syntax. It also allows actions to move outwards from a process and to commute in certain cases. The commuting rules follow the normal intuitions [Lea 2008].

The structural order ($\geqq$) is defined using an order ($\triangleright$) on single-threaded action sequences; $\vec{\sigma} \triangleright \vec{\tau}$ is defined only if $|\mathit{thrd}(\vec{\sigma}) \cup \mathit{thrd}(\vec{\tau})| = 1$. Formally, $\triangleright$ is defined to be the least precongruence on single-threaded action sequences that satisfies the following.

(A-NONLOCK) If σ is a write or speculation, τ is write or speculation and $\mathit{loc}(\sigma) = \mathit{loc}(\tau)$ implies $\mathit{val}(\sigma) = \mathit{val}(\tau)$ then $\sigma\tau \triangleright \tau\sigma$.
(A-BEGIN) If α is a begin then $\langle s\texttt{?}p\,.f\texttt{=}v\rangle\alpha \triangleright \alpha\langle s\texttt{?}p\,.f\texttt{=}v\rangle$.
(A-ACQUIRE) If α is write and β is an acquire then $\alpha\beta \triangleright \beta\alpha$.
(A-RELEASE) If α is write and β is a release then $\beta\alpha \triangleright \alpha\beta$.
(A-ABSORPTION1) If α is a write then $\alpha \triangleright \alpha\alpha$.
(A-ABSORPTION2) If α and β are writes to the same location then $\beta\alpha \triangleright \alpha$.
(A-ABSORPTION3) If α, β and β' are writes to the same location then $\beta\beta'\alpha \triangleright \beta'\beta\alpha$.

Define the reduction relation $A \to B$ to be the least relation satisfying the rules and axioms in Equations 6.1. Define $\twoheadrightarrow$ to be the reflexive and transitive closure of $\to$. In the figure, $\mathscr{BV}$ is the set of base values. We rely on many other standard auxiliary definitions[1].

[1] In classes and methods, the parameters are bound. In statements, bound variables are preceded by `val`. In processes, bound names are preceded by ν. There are no binders in contexts. We identify syntax up to renaming of bound variables and names and write $M\{\![v/x]\!\}$ for the capture avoiding substitution of v for x in M. We assume similar notation for substitution of names for names and for substitution over other syntax categories.
Suppose d is declared as `class` $d\langle\vec{\chi}\rangle$`{`$\vec{\mathscr{F}}\,\vec{\mathscr{M}}$`}`. Define the partial function ftype so that $\mathit{ftype}(d\langle\vec{a}\rangle.f) = \texttt{final}\ a\,T$ if $\vec{\mathscr{F}}\{\![\vec{a}/\vec{\chi}]\!\}$ contains the declaration "`final` $a\,T\,f$`;`", and likewise for racing and lock-protected fields. Define the partial function mtype so that $\mathit{mtype}(d\langle\vec{a}\rangle.m) = b;\vec{a}\,\vec{T} \to a\,T$ if $\vec{\mathscr{M}}\{\![\vec{a}/\vec{\chi}]\!\}$ contain the declaration $a\,T\ m$`(`b`this;`$\vec{a}\,\vec{T}\ \vec{x}$`){`$M$`}`.
To define the dynamics, it suffices to describe classes in terms of three partial functions (*finals*, *nonfinals* and *mbody*). If $\mathit{finals}(d\langle\vec{a}\rangle) = \vec{f}$ then $\vec{f}$ are final fields of class d, similarly for *nonfinals*. If $\mathit{mbody}(d\langle\vec{a}\rangle.m) = \lambda\vec{x}.M$ then d implements method m with parameters $\vec{x}$ and body M. The abstraction $\lambda\vec{x}.M$ is written $\lambda.M$ when $\vec{x}$ is the empty sequence.

$$\frac{\text{(S-NU-FREE)}}{A \mathrel{\dot\equiv} A \,|\, (\nu p)\,(p\!:\!d\langle\vec\gamma\rangle \,|\, \mathtt{free}\ p)} \qquad \frac{\text{(S-NU-PAR)}}{(\nu p)\,(B\,|\,A) \mathrel{\dot\equiv} B\,|\,((\nu p)\,A)}\ p \notin \mathit{fn}(B) \qquad \frac{\text{(S-PAR-PAR)}}{B\,|\,(A\,|\,A') \mathrel{\dot\equiv} (B\,|\,A)\,|\,A'}$$

$$\frac{\text{(S-PAR)}}{A\,|\,A' \mathrel{\dot\equiv} A'\,|\,A} \qquad \frac{\text{(S-NU-NU)}}{(\nu p)\,(\nu p')A \mathrel{\dot\equiv} (\nu p')\,(\nu p)\,A} \qquad \frac{\text{(S-NU-PREFIX)}}{(\nu p)\,\alpha A \mathrel{\dot\equiv} \alpha(\nu p)\,A}\ p \notin \mathit{fn}(\alpha) \qquad \frac{\text{(S-PREFIX-PAR)}}{B\,|\,(\alpha A) \geqq \alpha(B\,|\,A)} \qquad \frac{\text{(S-PREFIX)}}{\vec\alpha A \geqq \vec\beta A}\ \vec\alpha \triangleright \vec\beta$$

$$\frac{\text{(S-PREFIX-SPECULATION)}}{\top\Rightarrow(\alpha A)\,[]\,\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle\Rightarrow(\alpha A') \ \geqq\ \alpha\,(\top\Rightarrow A\,[]\,\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle\Rightarrow A')} \quad \begin{array}{l}\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle\alpha \triangleright \alpha\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle \\ \text{or } \mathit{thrd}(\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle) \neq \mathit{thrd}(\alpha)\end{array}$$

$$\frac{\text{(S-NU-SPECULATION)}}{(\nu p)\,(\top\Rightarrow A\,[]\,\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle\Rightarrow A') \mathrel{\dot\equiv} \top\Rightarrow((\nu p)\,A)\,[]\,\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle\Rightarrow((\nu p)\,A')}\ p \notin \mathit{fn}(\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle)$$

$$\frac{\text{(S-PAR-SPECULATION)}}{B\,|\,(\top\Rightarrow A\,[]\,\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle\Rightarrow A') \mathrel{\dot\equiv} \top\Rightarrow(B\,|\,A)\,[]\,\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle\Rightarrow(B\,|\,A')}\ \mathit{thrd}(\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle) \notin \mathit{thrds}(B)$$

$$\text{(R-STRUCTURAL-ORDER)}\ \frac{A \geqq B \quad B \to B' \quad B' \geqq A'}{A \to A'} \qquad \text{(R-CAST)}\ \frac{\mathbb{C} \ni p\!:\!D}{\begin{array}{l}\quad\ \mathbb{C}\llbracket s\mathtt{[val}\ x\ \mathtt{=}\ \mathtt{(}D\mathtt{)}p\mathtt{;}\ M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket s\mathtt{[}M\{\!\![p/x]\!\!\}\mathtt{]}\rrbracket\end{array}} \qquad \text{(R-OPERATOR)}\ \frac{w \text{ is the result of } \mathit{op} \text{ on } \vec v}{\begin{array}{l}\quad\ \mathbb{C}\llbracket s\mathtt{[val}\ x\ \mathtt{=}\ \mathit{op}\mathtt{(}\vec v\mathtt{)\,;}\ M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket s\mathtt{[}M\{\!\![w/x]\!\!\}\mathtt{]}\rrbracket\end{array}}$$

$$\frac{\text{(R-IF-TRUE)}}{\mathbb{C}\llbracket s\mathtt{[if\ (true)\ \{}M\mathtt{\}\ else\ \{}N\mathtt{\}]}\rrbracket \to \mathbb{C}\llbracket s\mathtt{[}M\mathtt{]}\rrbracket} \qquad \frac{\text{(R-IF-FALSE)}}{\mathbb{C}\llbracket s\mathtt{[if\ (false)\ \{}M\mathtt{\}\ else\ \{}N\mathtt{\}]}\rrbracket \to \mathbb{C}\llbracket s\mathtt{[}N\mathtt{]}\rrbracket}$$

$$\text{(R-METHOD-CALL)}\ \frac{\mathbb{C} \ni p\!:\!D \quad \mathit{mbody}(D.m) = \lambda\vec y.N \quad D \text{ not reserved} \quad m \neq \mathtt{start}}{\begin{array}{l}\quad\ \mathbb{C}\llbracket s\mathtt{[val}\ x\ \mathtt{=}\ p\,\mathtt{.}m\mathtt{(}\vec v\mathtt{)\,;}\ M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket s\mathtt{[val}\ x\ \mathtt{=\ \{}N\{\!\![p/\mathtt{this}]\!\!\}\{\!\![\vec v/\vec y]\!\!\}\mathtt{\}}M\mathtt{]}\rrbracket\end{array}} \qquad \frac{\text{(R-METHOD-RETURN)}}{\begin{array}{l}\quad\ \mathbb{C}\llbracket s\mathtt{[val}\ x\ \mathtt{=\ \{}{\uparrow}v\mathtt{\}}M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket s\mathtt{[}M\{\!\![v/x]\!\!\}\mathtt{]}\rrbracket\end{array}}$$

$$\text{(R-METHOD-CONTEXT)}\ \frac{\mathbb{C}\llbracket s\mathtt{[}N\mathtt{]}\rrbracket \to \mathbb{C}'\llbracket s\mathtt{[}N'\mathtt{]}\rrbracket}{\begin{array}{l}\quad\ \mathbb{C}\ \llbracket s\mathtt{[val}\ x\ \mathtt{=\ \{}N\ \mathtt{\}}M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}'\llbracket s\mathtt{[val}\ x\ \mathtt{=\ \{}N'\mathtt{\}}M\mathtt{]}\rrbracket\end{array}}$$

$$\text{(R-START)}\ \frac{\mathbb{C} \ni p\!:\!D \quad \mathit{mbody}(D.\mathtt{run}) = \lambda\vec y.N \quad D \text{ not reserved} \quad \mathbb{C} \ni \ell\!:\!\mathtt{Lock}}{\begin{array}{l}\quad\ \mathbb{C}\llbracket \mathtt{free}\ \ell\ \mathtt{|}\ s\mathtt{[val}\ x\ \mathtt{=}\ p\mathtt{.start()\,;}\ M\mathtt{]}\ \mathtt{|}\ \mathtt{runnable}\ p\rrbracket \\ \to\ \mathbb{C}\llbracket \qquad\qquad \langle s\mathtt{!}\ell\mathtt{:1}\rangle\,(s\mathtt{[}M\{\!\![\mathtt{unit}/x]\!\!\}\mathtt{]}\ \mathtt{|}\ \langle p\mathtt{!}\ell\mathtt{:2}\rangle\,p\mathtt{[}N\{\!\![p/\mathtt{this}]\!\!\}\mathtt{]})\rrbracket\end{array}}$$

$$\text{(R-NEW)}\ \frac{\mathbb{C} \ni p\!:\!D \quad \mathit{finals}(D) = \vec f \quad \mathit{nonfinals}(D) = \vec g \quad D \text{ not reserved}}{\begin{array}{l}\quad\ \mathbb{C}\llbracket \mathtt{free}\ p \qquad\ \ \mathtt{|}\ s\mathtt{[val}\ x\ \mathtt{=\ new}\ D\langle\vec a\rangle\mathtt{(}\vec v\mathtt{)\,;}\ M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket \mathtt{runnable}\ p\ \mathtt{|}\ \langle s\mathtt{!}p\,.\vec f\mathtt{=}\vec v\rangle\,\langle s\mathtt{!}p\,.\vec g\mathtt{=}\overrightarrow{\mathtt{null}}\rangle\,\langle s\mathtt{!}p\rangle\,s\mathtt{[}M\{\!\![p/x]\!\!\}\mathtt{]}\rrbracket\end{array}}$$

$$\frac{\text{(R-FIELD-WRITE)}}{\mathbb{C}\llbracket s\mathtt{[}p\,.f\ \mathtt{=}\ v\mathtt{;}\ M\mathtt{]}\rrbracket \to \mathbb{C}\llbracket \langle s\mathtt{!}p\,.f\mathtt{=}v\rangle\ s\mathtt{[}M\mathtt{]}\rrbracket} \qquad \text{(R-FIELD-READ)}\ \frac{\mathbb{C}, s \text{ justifies read } p\,.f\mathtt{=}v}{\mathbb{C}\llbracket s\mathtt{[val}\ x\ \mathtt{=}\ p\,.f\mathtt{\,;}\ M\mathtt{]}\rrbracket \to \mathbb{C}\llbracket s\mathtt{[}M\{\!\![v/x]\!\!\}\mathtt{]}\rrbracket} \qquad \text{(R-END)}\ \frac{\mathbb{C}, s \text{ justifies end } p}{\begin{array}{l}\quad\ \mathbb{C}\llbracket s\mathtt{[end}\ p\mathtt{;}\ M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket s\mathtt{[}M\mathtt{]}\rrbracket\end{array}}$$

$$\text{(R-SPECULATION-OPEN)}\ \frac{s \in \mathit{thrds}(A) \quad p \in \mathit{objs}(A) \quad v \in \mathscr{BV} \cup \mathit{objs}(A)}{\mathbb{C}\llbracket A\rrbracket \to \mathbb{C}\llbracket \top\Rightarrow A\,[]\,\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle\Rightarrow A\rrbracket} \qquad \text{(R-SPECULATION-CLOSE)}\ \frac{\mathbb{C}, s \text{ justifies speculation } p\,.f\mathtt{=}v}{\mathbb{C}\llbracket \top\Rightarrow A\,[]\,\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle\Rightarrow B\rrbracket \to \mathbb{C}\llbracket B\rrbracket}$$

$$\frac{\text{(R-ATOMIC-NEW)}}{\begin{array}{l}\quad\ \mathbb{C}\llbracket \mathtt{free}\ \ell\ \mathtt{|}\ \ell\mathtt{:Atomic}\ \mathtt{|}\ s\mathtt{[val}\ x\ \mathtt{=\ new\ Atomic(}v\mathtt{)\,;}\ M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket \ell\mathtt{:Atomic\{}v\mathtt{;1\}} \qquad \mathtt{|}\ \langle s\mathtt{!}\ell\rangle\,\langle s\mathtt{!}\ell\mathtt{:1}\rangle\,s\mathtt{[}M\{\!\![\ell/x]\!\!\}\mathtt{]}\rrbracket\end{array}}$$

$$\frac{\text{(R-ATOMIC-GETANDSET)}}{\begin{array}{l}\quad\ \mathbb{C}\llbracket \ell\mathtt{:Atomic\{}v\mathtt{;}j\mathtt{\}} \qquad\quad \mathtt{|}\ s\mathtt{[val}\ x\ \mathtt{=}\ \ell\mathtt{.gas(}w\mathtt{)\,;}\ M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket \ell\mathtt{:Atomic\{}w\mathtt{;}j'\mathtt{+1\}}\ \mathtt{|}\ \langle s\mathtt{!}\ell\mathtt{:}\,j'\rangle\,\langle s\mathtt{!}\ell\mathtt{:}\,j'\mathtt{+1}\rangle\,s\mathtt{[}M\{\!\![v/x]\!\!\}\mathtt{]}\rrbracket\end{array}} \quad \begin{array}{l} j' = j{+}1 \text{ if } j \text{ odd} \\ j' = j \quad\ \text{ if } j \text{ even} \\ \mathbb{C} \text{ enables} \langle s\mathtt{!}\ell\mathtt{:}\,j\rangle \end{array}$$

$$\frac{\text{(R-ATOMIC-GET)}}{\begin{array}{l}\quad\ \mathbb{C}\llbracket \ell\mathtt{:Atomic\{}v\mathtt{;}j\mathtt{\}}\ \ \mathtt{|}\ s\mathtt{[val}\ x\ \mathtt{=}\ \ell\mathtt{.get()\,;}\ M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket \ell\mathtt{:Atomic\{}v\mathtt{;}j'\mathtt{\}}\ \mathtt{|}\ \langle s\mathtt{!}\ell\mathtt{:}\,j'\rangle\,s\mathtt{[}M\{\!\![v/x]\!\!\}\mathtt{]}\rrbracket\end{array}} \quad \begin{array}{l} j' = j{+}1 \text{ if } j \text{ odd} \\ j' = j \quad\ \text{ if } j \text{ even} \\ \mathbb{C} \text{ enables} \langle s\mathtt{!}\ell\mathtt{:}\,j\rangle \end{array}$$

$$\frac{\text{(R-ATOMIC-COMPAREANDSET-FALSE)}}{\begin{array}{l}\quad\ \mathbb{C}\llbracket \ell\mathtt{:Atomic\{}v\mathtt{;}j\mathtt{\}}\ \mathtt{|}\ s\mathtt{[val}\ x\ \mathtt{=}\ \ell\mathtt{.cas(}u\mathtt{,}w\mathtt{)\,;}\ M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket \ell\mathtt{:Atomic\{}v\mathtt{;}j\mathtt{\}}\ \mathtt{|}\ s\mathtt{[}M\{\!\![\mathtt{null}/x]\!\!\}\mathtt{]}\rrbracket\end{array}} \quad u \neq v$$

$$\frac{\text{(R-ATOMIC-SET)}}{\begin{array}{l}\quad\ \mathbb{C}\llbracket \ell\mathtt{:Atomic\{}v\mathtt{;}j\mathtt{\}}\ \ \mathtt{|}\ s\mathtt{[val}\ x\ \mathtt{=}\ \ell\mathtt{.set(}w\mathtt{)\,;}\ M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket \ell\mathtt{:Atomic\{}w\mathtt{;}j'\mathtt{\}}\ \mathtt{|}\ \langle s\mathtt{!}\ell\mathtt{:}\,j'\rangle\,s\mathtt{[}M\{\!\![\mathtt{unit}/x]\!\!\}\mathtt{]}\rrbracket\end{array}} \quad \begin{array}{l} j' = j \quad\ \text{ if } j \text{ odd} \\ j' = j{+}1 \text{ if } j \text{ even} \\ \mathbb{C} \text{ enables} \langle s\mathtt{!}\ell\mathtt{:}\,j\rangle \end{array}$$

$$\frac{\text{(R-ATOMIC-COMPAREANDSET-TRUE)}}{\begin{array}{l}\quad\ \mathbb{C}\llbracket \ell\mathtt{:Atomic\{}v\mathtt{;}j\mathtt{\}} \qquad\quad \mathtt{|}\ s\mathtt{[val}\ x\ \mathtt{=}\ \ell\mathtt{.cas(}v\mathtt{,}w\mathtt{)\,;}\ M\mathtt{]}\rrbracket \\ \to\ \mathbb{C}\llbracket \ell\mathtt{:Atomic\{}w\mathtt{;}j'\mathtt{+1\}}\ \mathtt{|}\ \langle s\mathtt{!}\ell\mathtt{:}\,j'\rangle\,\langle s\mathtt{!}\ell\mathtt{:}\,j'\mathtt{+1}\rangle\,s\mathtt{[}M\{\!\![v/x]\!\!\}\mathtt{]}\rrbracket\end{array}} \quad \begin{array}{l} j' = j{+}1 \text{ if } j \text{ odd} \\ j' = j \quad\ \text{ if } j \text{ even} \\ \mathbb{C} \text{ enables} \langle s\mathtt{!}\ell\mathtt{:}\,j\rangle \end{array}$$

Figure 2. Structural order ($A \geqq B$) and reduction ($A \to B$)

Many of the reduction rules are standard. We very briefly discuss some of the exceptions.

`Atomic` objects maintain a shared global state, enforcing sequential consistency. The state includes an integer, which is odd if the last operation was a set. These objects generate synchronization actions, which define the synchronizes-with relation. Even actions may be seen as an get/acquire and odd actions a set/release. In the formal development, we take it that `cas` returns `null` if the comparison fails, and the prior object otherwise.

As in Java, the reserved method `start` starts method `run` under the thread identity of the receiving object; this is a synchronization event enforced using a fresh "dummy" lock.

R-FIELD-WRITE describes field writes in a relaxed memory model, so the field writes become actions that float into the evaluation context. Field reads, R-FIELD-READ, may take any value that is justified by the evaluation context. R-END removes an end if the creation of the object reference is in a happens-before relationship; this is used in the following sections on typing.

Speculation can occur at any point, using R-SPECULATION-OPEN. The initial branch has guard $\top$, indicating that this branch may make no additional assumptions. The final branch has a speculative action as its guard that can be used to justify reads. The context rules permit each branch of speculation evolve independently. Results from an active speculation can only leak to the outside world via S-PREFIX-SPECULATION if all branches produce the same action. This is significant, since only actions that manage to make it outside of a speculation may be used to finalize via R-SPECULATION-CLOSE.

6.2 Defining soundness

Soundness is defined in terms of *stuck* threads, which are defined in terms of *evaluation contexts*, defined as follows.

$$\begin{array}{rll} \mathbb{E} & ::= \llbracket - \rrbracket \ \mid\ \mathtt{val}\ x\ \mathtt{=\ \{}\mathbb{E}\mathtt{\}}\ M & \text{(Statement ctxt)} \\ \mathbb{C} & ::= \llbracket - \rrbracket \ \mid\ \mathbb{C}\,|\,A \ \mid\ A\,|\,\mathbb{C} \ \mid\ (\nu p)\,\mathbb{C} \ \mid\ \alpha\,\mathbb{C} & \text{(Process ctxt)} \\ & \quad \mid \top\Rightarrow\mathbb{C}\,[]\,\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle\Rightarrow A \ \mid\ \top\Rightarrow A\,[]\,\langle s\mathtt{?}p\,.f\mathtt{=}v\rangle\Rightarrow\mathbb{C} & \end{array}$$

Roughly, a stuck thread is a non-terminated thread that cannot reduce. As discussed by Igarashi, Pierce, and Wadler [2001], Java allows runtime type exceptions which are not considered a failure of the static type system. Here we also have null pointer exceptions[2].

Definition 1 (Stuck thread). Thread s is *stuck* in A if there exists $\mathbb{C}\llbracket s\,\texttt{[}\mathbb{E}\llbracket M\rrbracket\texttt{]}\rrbracket = A$ such that *none* of the following hold.

(1) s has terminated: M is a return statement and $\mathbb{E} = \llbracket - \rrbracket$;
(2) s can reduce: $\mathbb{C}\llbracket s\,\texttt{[}\mathbb{E}\llbracket M\rrbracket\texttt{]}\rrbracket \to \mathbb{C}'\llbracket s\,\texttt{[}M'\texttt{]}\rrbracket$ for some $\mathbb{C}'$, M';
(3) s has had a class cast exception: $M = \texttt{val}\; x \;\texttt{=}\; (D)v\texttt{;}\; M'$ and $\mathbb{C}$ does not contain subterm $v\,{:}\,D$; or
(4) s has had a null pointer exception: M is a method, end, read, write, conditional or synchronization statement with target `null`. □

The stuck processes of most interest come about because of the premise of the reduction rule for end statements. In order for `end` p to reduce, it must be the case the action $\langle t\,\texttt{!}\,p\rangle$ happens-before, for some t. This is a global property of the system, tracking data flow.

Theorem 2 (Soundness). *Suppose the class table is well typed and method* `Main.main ()` *is defined. Suppose further that A is a* speculation-free *process that is reachable from the bootstrap process, that is*

$$(\nu\texttt{m})\;\texttt{m:Main}\;\texttt{|}\;\texttt{m[m.main();]} \to \cdots \to A.$$

Then A does not contain a stuck thread. □

We restrict attention to speculation-free processes, because threads within the final branch of a "bad" speculation may get stuck. Typing ensures that such speculations can never be finalized.

7. Typing processes

The proof of soundness follows standard lines: We provide typing rules for processes and prove subject reduction and progress. The details of the proof are too long to include here. Indeed, even the invariant maintained by running processes is too long to fully describe here. In this section, we describe some of the most interesting typing rules for processes, which embody intermediate states of computation. We first discuss the rules for "standard" processes, then for speculation.

In order to type processes, we introduce *effects* and extend environments as follows.

$$\begin{aligned} \zeta, \xi \;&::=\; \wr p \int \\ E \;&::=\; \cdots \mid E, p{:}D \mid E, s{:}\texttt{lock}\;\ell \mid E, s\;\texttt{returns}\;aD \\ &\quad\;\; \mid E, \zeta \mid E, s{:}\zeta \mid E, \ell{:}\zeta \mid E, \texttt{onacq}\;\Phi\;\zeta \end{aligned}$$

The effect $\wr p \int$ indicates that $\langle t\,\texttt{!}\,p\rangle$ has occurred, for some t. Effects are associated with threads ($s{:}\zeta$) and synchronization objects, or *locks* ($\ell{:}\zeta$). Naked effects (ζ), used in statement typing, are associated with the "local" thread. The environment $s{:}\texttt{lock}\;\ell$ indicates that s holds lock ℓ. The `onacq` environment is used to type bad speculations, as discussed in Section 7.1.

Define $locks(s, E) = \{\texttt{lock}\;V \mid s{:}\texttt{lock}\;V \in E\}$ and $effects(s, E) = \{\zeta \mid s{:}\zeta \in E\}$. Define $E \Vdash_s \Phi$ to mean $E, effects(s, E), locks(s, E) \Vdash \Phi$, and similarly for $E \vdash_s V : aT$ and $E \vdash_s M : aT$. Let Z range over sets of effects.

The new rules for values and the key rules for processes ($E \vdash A$) and actions ($E \vdash \sigma \triangleright E'$) are given in Figure 3. The proof of soundness also uses *well-formedness*, which is defined in the full version of this paper. Well-formedness is trivially true of the bootstrap process and is preserved by structural order and reduction. Well-formedness eliminates nonsense processes, such as $s\,\texttt{[}M\texttt{]}\,\texttt{|}\,s\,\texttt{[}N\texttt{]}$ and $\langle s\,\texttt{!}\,p\,.f\texttt{=}v\rangle\,\langle t\,\texttt{!}\,p\,.f\texttt{=}w\rangle\,A$, where f is a final field.

The action sequence records the entire history of the running process. The type rules can compute the happens-before relation

(VAL-OBJECT-□)
$$\frac{E \vdash \diamond \quad E \ni p{:}D}{E \vdash p : \Box D}$$

(VAL-OBJECT-⊛)
$$\frac{E \vdash \diamond \quad E \ni p{:}D \quad E \Vdash \wr p \int}{E \vdash p : \boxast D}$$

(PROC-ACTION)
$$\frac{E \vdash \alpha \triangleright E' \quad E' \vdash A}{E \vdash \alpha A}$$

(PROC-THREAD)
$$\frac{E \ni s\;\texttt{returns}\;aT \quad E \vdash_s M : aT}{E \vdash s\,\texttt{[}M\texttt{]}}$$

(ACT-BEGIN)
$$\frac{s \in dom(E) \quad p \in dom(E)}{E \vdash \langle s\,\texttt{!}\,p\rangle \triangleright E, s{:}\,\wr p \int}$$

(ACT-ACQUIRE)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni \ell{:}\texttt{Lock} \\ E' = \left\{ s{:}\zeta \;\middle|\; E \ni \texttt{onacq}\;\Phi\;\zeta \text{ and } E \Vdash_s \Phi \right\} \end{array}}{E \vdash \langle s\,\texttt{!}\,\ell{:}\,j\rangle \triangleright E, E', s{:}\texttt{lock}\;\ell, s{:}effects(\ell, E)} \; j \text{ even}$$

(ACT-RELEASE)
$$\frac{s \in dom(E) \quad E \ni \ell{:}\texttt{Lock}}{E, s{:}\texttt{lock}\;\ell \vdash \langle s\,\texttt{!}\,\ell{:}\,j\rangle \triangleright E, \ell{:}effects(s, E)} \; j \text{ odd}$$

(ACT-⊛-GUARDED-WRITE)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \quad ftype(D.f) = \boxast S\;\texttt{wrguard}\;\Psi \\ E \vdash_s v : \boxast S \quad E \ni t{:}\,\wr p \int \text{ implies } E \Vdash_s \Psi\{\!\![{}^{p}\!/\texttt{this}]\!\!\} \end{array}}{E \vdash \langle s\,\texttt{!}\,p\,.f\texttt{=}v\rangle \triangleright E}$$

Figure 3. Selected type rules for processes (actions and threads)

(ACT-⊛-GUARDED-SPECULATION)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \\ E \vdash_s v : \Box S \quad ftype(D.f) = \boxast S\;\texttt{rdguard}\;\Phi \\ E' = \left\{ t{:}\,\wr v \int \;\middle|\; E \Vdash_t \Phi\{\!\![{}^{p}\!/\texttt{this}]\!\!\} \right\} \end{array}}{E \vdash \langle s\,\texttt{?}\,p\,.f\texttt{=}v\rangle \triangleright E, E', \texttt{onacq}\;(\Phi\{\!\![{}^{p}\!/\texttt{this}]\!\!\})\;(\wr v \int)}$$

(ACT-⊛-FINAL-SPECULATION)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \vdash_s v : \Box S \\ E \ni p{:}D \quad ftype(D.f) = \texttt{final}\;\boxast\;S \end{array}}{E \vdash \langle s\,\texttt{?}\,p\,.f\texttt{=}v\rangle \triangleright E, \left\{ t{:}\,\wr v \int \;\middle|\; E \Vdash_t \wr p \int \right\}}$$

(PROC-SPECULATION)
$$\frac{E \vdash A \quad E \vdash \phi \triangleright E' \quad E' \vdash B}{E \vdash \top \Rightarrow A \,[\!]\, \phi \Rightarrow B}$$

(PROC-SPECULATION-SHAPE-ERROR)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \quad E \vdash A \\ ftype(D.f) = \ldots D' \ldots \text{ and } E \ni v{:}D'' \text{ and } D'' \neq D' \end{array}}{E \vdash \top \Rightarrow A \,[\!]\, \langle s\,\texttt{?}\,p\,.f\texttt{=}v\rangle \Rightarrow B}$$

Figure 4. Selected type rules for processes (speculation)

as it moves through the action context. The same is true for the locks held at any point. Subject reduction uses the fact that such sequences are stable up to single-threaded action reordering.

ACT-ACQUIRE adds a lock to the environment and copies effects from locks to threads. ACT-RELEASE removes a lock from the environment and copies effects from threads to locks. (We discuss `onacq` in in Section 7.1.)

Using PROC-ACTION and ACT-BEGIN, the effect $\wr p \int$ is available to A in $\langle t\,\texttt{!}\,p\rangle A$. The VAL-OBJECT rules subsequently require that $\wr p \int$ be present in order to type p at $\boxast$.

ACT-⊛-GUARDED-WRITE directly parallels STAT-GUARDED-WRITE, ensuring that appropriate locks and effects are in scope. The locks are only necessary after the fields of p are initialized and thus $t{:}\,\wr p \int$ is in scope, for some t. This rule is preserved by action reordering: Although write actions can commute with some write actions, the rules require that locks can only expand their scope.

7.1 Speculation

Several of the typing rules for speculation are given in Figure 4.

[2] When formalizing synchronization statements, one must add a case for threads that are waiting to acquire a lock.

The speculation ⊤⇒*A* [] ⟨s?p.f=q⟩⇒*B* allows the justifying branch *A* and final branch *B* to proceed independently, and also allows the final branch to speculate on a future write ⟨s!p.f=q⟩. The operational semantics does not constrain the speculated write, so the speculated write may not be well-typed. Consequently, a read of the speculated write by the final branch process *B* can cause the resulting final branch to fail to type. Moreover, the speculation can reduce to the final branch if the speculated write is justified by the speculation's context. Thus it is natural to ask whether a process with ill-typing confined to the final branch of a speculation can reduce to a process with ill-typing outside speculation.

There are two classes of typing errors that may arise. First, shape errors may arise when a speculated write refers to an object with an incompatible class. For example, the speculated write has a shape error in the following process, where p:C7, q:Object.

```
class C7 {C7 f;}
⊤⇒ ... [] ⟨s?p.f=q⟩⇒ ...
```

Such shape errors are permitted in final branches of speculations. However, such speculations are *unfinalizable* since a well-typed justifying branch, which does not see the speculation, cannot produce the necessary write to justify the speculation. The use of the justifying branch of a speculation as a filter is critical to prevent ill-typed actions leaking from the final branch of a speculation.

Type errors may also arise in the final branch of a speculation due to a misuse of effects, but such speculations are also unfinalizable. Unlike shape errors, it is not immediately evident that an effect error has occurred. Our approach with effects, therefore, is to saturate the final branch with sufficient effects to ensure that no type errors arise, while ensuring that all of the imparted effects can be justified when the speculation is finalized.

A read of the speculated write may convey information about happens-before relationships via ⧆-annotated types that is not justified by the process context. For example, a speculated write ⟨s?p.f=q⟩ may be read in the final branch of a speculation in the absence of a happens-before relationship with ≀q∫ (thus preventing a write action ⟨s!p.f=q⟩ being typeable at the point of the speculation).

The type system accommodates potential memory-effect errors by associating *speculated effects* ≀q∫ with speculated writes ⟨s?p.f=q⟩, whenever the corresponding write ⟨s!p.f=q⟩ would convey a memory effect. This allows the final branches of speculations to be typed even if reduction reads speculated writes that do not yet convey happens-before relationships. Speculated effects can be discharged when their associated speculations are finalized because the justifying writes for the speculation appear outside the speculation and thus justify the speculated memory effects.

Typing for speculation on a final field does not require locks or latent effects. Instead, as with reads of final fields, it is significant whether a thread has the effect ≀p∫ given a speculated write ⟨s?p.f=q⟩. ACT-⧆-FINAL-SPECULATION generates an environment extension with an effect ≀q∫ for all threads that prove ≀p∫. If the speculation on a final field's write is finalized, the use of ACT-⧆-FINAL-SPECULATION can be discharged immediately in the resulting final branch.

It is worth emphasizing that the additional effects provided by speculative fields are only necessary for speculations that cannot be finalized. Our semantics has been designed to allow as many speculations as possible.

Alternatively, one could attempt to disallow "useless" speculations, which either (1) cannot be finalized or (2) add nothing to the execution. For example, one can show that speculation with a value of the wrong shape is useless, for the first reason; speculating on a lock-protected field is useless, for the second reason. It seems difficult, however to prohibit useless speculation on final fields, which communicate effects. In addition, our approach can be adapted to richer effect languages.

7.2 Typing examples

The environments used to type processes have some shared information, common to all threads, such as the type of an object reference p. In addition, environments record local information including effects held on a per thread basis. Consequently, each thread may have a different view of the effects associated with an object.

For example, consider the following process (we leave names free, and omit uninteresting local variable bindings and continuation terms). Let p:C5.

```
class C5 {final ⧆Lock l; ⧆Object f guard this.l;}
s[val x=new Object(); sync p.l{p.f=x;}] |
t[sync p.l{val y=p.f; end y;}]
```

After the constructor, thread s will be typed in an environment including x:⧆Object. In the presence of suitable auxiliary processes such as "free q", the above process may reduce to the following, where p:C5, q:Object.

```
⟨s!q⟩(s[sync p.l{p.f=q;}] |
      t[sync p.l{val y=p.f; end y;}])
```

The environment for typing the threads includes p:C5, q:Object, s:≀q∫. The localized effect s:≀q∫ represents the fact that code running at thread s has a happens-before relationship with q's constructor. This effect is introduced into the environment by PROC-ACTION's use of ACT-BEGIN when typing the action ⟨s!p⟩. The same effect is required to justify the typing of q as ⧆Object by VAL-OBJECT-⧆. Without the effect s:≀q∫, uses of q could only be typed as □Object via VAL-OBJECT-□.

The effect s:≀q∫ cannot be used directly within thread t, but is communicated in the statics (as in the dynamics) via memory synchronization actions such as lock acquisition and release. To see this, consider a further three reductions by thread s above. We assume that a write of k to p.l by some thread is visible. (We apply structural reordering for readability and again omit auxiliary processes. We assume that lock k has not yet been used before). Let p:C5, q:Object, k:Lock.

```
⟨s!q⟩⟨s!k:0⟩⟨s!p.f=q⟩⟨s!k:42⟩
         (s[] | t[sync p.l{val y=p.f; end y;}])
```

The environment records that lock k is held by thread s between the actions ⟨s!k:0⟩ and ⟨s!k:42⟩. In particular, this justifies the write ⟨s!p.f=q⟩ to a ⧆ field guarded by k. Moreover, the assignment of q to a ⧆ field in ⟨s!p.f=q⟩ requires the s:≀q∫ effect in the environment at the point of the write, mirroring the assignment typing in terms described above.

Further reductions lead to the following process where the lock k has been acquired by t, and q has been read from p.f then substituted for z. Let p:C5, q:Object, k:Lock.

```
⟨s!q⟩⟨s!k:0⟩⟨s!p.f=q⟩⟨s!k:42⟩⟨t!k:27⟩
                 (s[] | t[{end q;} k.release();])
```

Here the typing of q via STAT-END requires that q be typed as ⧆Object within thread t, and, as before, this requires that the environment provides ≀q∫ for thread t. The lock release ⟨s!k:42⟩ and lock acquire ⟨t!k:27⟩ achieve this by transferring the effects known to thread s (s:≀q∫) to lock k (k:≀q∫) at the release of k by thread s, and transferring the effects for lock k to thread t (t:≀q∫) at the acquisition of k by thread t. Thus we type the threads in an environment including t:≀q∫. In this way, the type system calculates communication of effects between threads following the memory synchronization actions.

The typing of final fields necessarily differs from the typing of fields guarded by locks because there are no memory synchronization actions used when reading a final field. The key observation in reading a final field is that the object reference through which the final field is read must be ⧆ annotated. For example, in typing the following process, (`end x`) is only permitted if `x:⧆Object`, and this in turns requires `s:≀p∫` in the typing environment for the process. Let `p:C6`.

```
class C6 {final ⧆Object f;}
s[val x=p.f; end x;]
```

This is expressed in the rule STAT-FINAL-READ.

Although writes to final fields may migrate via the structural order, this invariant is sufficient because writes to final fields always happen-before the begin representing construction of the object. The ⧆ nature of a value `q` read from a final field `p.f` then follows from transitivity of the happens-before relation between: (1) the construction of `q` and the write to `p.f`; (2) the write to `p.f` and the begin for `p`; (3) the begin for `p` and its receipt at ⧆ type; and (4) receipt of `p` and the read `p.f`. This property demonstrates how happens-before relationships can be established for lockless traversal of immutable portions of data structures whenever a happens-before relationship is established with the construction of the initial object reference.

8. Related work and conclusion

We have already cited several related pieces of work in context in the paper. We refer the reader to [Steinke and Nutt 2004] and [Adve and Gharachorloo 1996] for surveys of memory models for hardware architectures. Saraswat [2004] provides a framework for operational semantics with relaxed memory models for typed languages. Our prior work [Jagadeesan, Pitcher, and Riely 2010] uses the techniques of Cenciarelli, Knapp, and Sibilio [2007] and Boudol and Petri [2009] to accommodate the full expressiveness of the JMM.

The ⧆ annotation uses objects to witness side-effects [Terauchi and Aiken 2008]. However, in contrast to the large research on data races, we only enforce the absence of read-write data races on our witness objects and allow write-write data races. Happens-before plays a key role in the static analysis of programs, e.g., vector-clock algorithms for race detection [Flanagan and Freund 2009] or the characterization of execution paths [Burckhardt, Kothari, Musuvathi, and Nagarakatte 2010].

There has also been recent work on static analysis for weak memory models [Ferrara 2008; Miné 2011; Alglave, Kroening, Lugton, Nimal, and Tautschnig 2011] and provably correct compilers targeting weak memory Sevcík, Vafeiadis, Nardelli, Jagannathan, and Sewell [2011].

The study of relaxed memory models has hitherto focused on the dynamic semantics of programs. This paper has focused on static semantics. We have argued that correspondence assertions are the correct formalism for reasoning about happens-before relations, and that these correspondences can be specified and validated by enriching interfaces with notation for memory effects. Our techniques apply to common data structures from `java.util.concurrent`, including both lock-based and non-blocking implementations.

Our analysis intentionally includes only a small fragment of the type system of Abadi, Flanagan, and Freund [2006] for data race detection. Fully integrating other techniques used there (such as existential types in the form of ghost variables) and other existing techniques, including analyses based on locality, confinement, linearity, monotonicity, and effects is a topic of future research.

References

M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 28(2):207–255, 2006.

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

J. Alglave, D. Kroening, J. Lugton, V. Nimal, and M. Tautschnig. Soundness of data flow analyses for weak memory models. In *APLAS*, 2011.

H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI '08*, pages 68–78, 2008.

G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *POPL*, pages 392–403, 2009.

S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, pages 167–178, 2010.

P. Cenciarelli, A. Knapp, and E. Sibilio. The Java memory model: Operationally, denotationally, axiomatically. In *ESOP*, pages 331–346, 2007.

P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In *TAP*, pages 116–133, 2008.

C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theor. Comput. Sci.*, 300:379–409, 2003.

C. Hawblitzel. Linear types for aliased resources. Technical Report MSR-TR-2005-141, Microsoft Research, 2005.

A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

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

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

D. Lea. The JSR-133 cookbook for compiler writers. http://gee.cs.oswego.edu/dl/jmm/cookbook.html, 2008.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, pages 47–57, 1988.

J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL '05*, pages 378–391, 2005.

A. Miné. Static analysis of run-time errors in embedded critical parallel c programs. In *ESOP*, pages 398–418, 2011.

V. A. Saraswat. Concurrent constraint-based memory machines: A framework for Java memory models. In *ASIAN*, 2004.

J. Sevcík. *Program Transformations in Weak Memory Models*. PhD, Univ. of Edinburgh, 2008.

J. Sevcík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, pages 27–51, 2008.

J. Sevcík, V. Vafeiadis, F. Z. Nardelli, S. Jagannathan, and P. Sewell. Relaxed-memory concurrency and verified compilation. In *POPL*, 2011.

P. Sewell. Global/local subtyping and capability inference for a distributed π-calculus. In *ICALP '98, LNCS 1443*, pages 695–706, 1998.

R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5):800–849, 2004.

T. Terauchi and A. Aiken. Witnessing side effects. *TOPLAS*, 30(3), 2008.

T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. *IEEE Computer*, 25:39–52, January 1992.

T. Zhao, J. Palsberg, and J. Vitek. Type-based confinement. *J. Funct. Program.*, 16:83–128, January 2006.

A. Syntax

Figure 5 collects the syntactic definitions from the text. The syntax differs from that of the body text only in that we have introduced speculative actions and used these to define speculative processes. We have also introduced a category for actions. For bookkeeping purposes, we also annotate partially executed method bodies and synchronization blocks with the identity of the thread in which they occur.

Figure 6 includes the additional reduction rules for locks and conditions.

B. Well-formed processes

Define *procs*, mapping contexts to processes, as follows.

$$\begin{aligned} &procs(\llbracket - \rrbracket) = \emptyset \\ &procs(\alpha\mathbb{C}) = procs(\mathbb{C}) && procs((\nu q)\mathbb{C}) = procs(\mathbb{C}) \\ &procs(\top \Rightarrow A \,[]\, \phi \Rightarrow \mathbb{C}) = procs(\mathbb{C}) && procs(A \,|\, \mathbb{C}) = A \uplus procs(\mathbb{C}) \\ &procs(\top \Rightarrow \mathbb{C} \,[]\, \phi \Rightarrow A) = procs(\mathbb{C}) && procs(\mathbb{C} \,|\, A) = procs(\mathbb{C}) \uplus A \end{aligned}$$

A process A is *well-formed* if it satisfies the following.

(1) If A is $\mathbb{C}\llbracket\texttt{free}\ s\rrbracket$, $\mathbb{C}\llbracket\texttt{runnable}\ s\rrbracket$ or $\mathbb{C}\llbracket s\,\texttt{[}M\texttt{]}\rrbracket$ then the following hold.

   (a) $\mathbb{C}$ is contiguous.
   (b) There is no $\texttt{free}\ s$, $\texttt{runnable}\ s$ or $s\,\texttt{[}N\texttt{]}$ in $procs(\mathbb{C})$.
   (c) At most one of $\texttt{free}\ t$, $\texttt{runnable}\ t$ or $t\,\texttt{[}N\texttt{]}$ occurs in $procs(\mathbb{C})$, for any t.
   (d) At most one $p\texttt{:}D$ occurs in $procs(\mathbb{C})$, for any p.
   (e) If there is an action in $act(\mathbb{C}, s)$ with source $t \neq s$, then $t\,\texttt{[}N\texttt{]}$ occurs in $procs(\mathbb{C})$.
   (f) If p occurs in $\mathbb{C}$ or M, then $p\texttt{:}D$ occurs in $procs(\mathbb{C})$.
   (g) If $p\texttt{:}D$ occurs in $procs(\mathbb{C})$ and $f \in finals(D)$, then there exists exactly one write action to $p\texttt{.}f$ in $act(\mathbb{C}, s)$.

(2) Let *thrds* and *objs* return the names of the potential threads and the objects of a process, respectively. For example $thrds(\texttt{free}\ s) = thrds(\texttt{runnable}\ s) = thrds(s\,\texttt{[}M\texttt{]}) = \{s\}$ and $objs(p\texttt{:}D) = objs(p\texttt{:Atomic\{}v\texttt{;}j\texttt{\}}) = objs(p\texttt{:Lock\{}\vec{s}\texttt{;}\,j\texttt{\}}) = \{p\}$.

   (a) In any subprocess $B \,|\, B'$, we have that $objs(B) \cap objs(B') = \emptyset$, $thrds(B) \cap thrds(B') = \emptyset$.
   (b) In any subprocess αB, we have that $thrd(\alpha) \in thrds(B)$.
   (c) In any subprocess $\top \Rightarrow B \,[]\, \phi \Rightarrow B'$, we have that $thrd(\phi) \in thrds(B) \cap thrds(B')$.

Well-formedness is trivially true of the bootstrap process and is preserved by structural order and reduction.

Lemma 3. *The bootstrap process is well-formed.*

PROOF. Trivial. □

Lemma 4. *(a) Suppose A is well-formed and $A \geqq A'$ then A' is well-formed. (b) Suppose A is well-formed and $A \to A'$ then A' is well-formed.*

PROOF. (a) follows by induction on the proof that $A \geqq A'$. (b) follows by induction on the proof that $A \to A'$. □

C. Typing details

The reserved methods `start` and `run` have type "$\circledast;\, \emptyset \to \square\texttt{Unit}$."

The reserved classes have the following signatures. Recall that we elide "$\square$`this;`" from method interfaces. In the formal development, we take it that `cas` returns `null` if the comparison fails, and the prior object otherwise.

```
class Object { }
class Atomic⟨χ⟩ {
  χ Object get();
  void set(χ Object v);
  χ Object gas(χ Object v);
  χ Object cas(Object oldv, χ Object newv);
}
class Lock { }
class Cond⟨χ⟩ {
  χ Object await();
  void signal(χ Object);
}
```

We now present the judgments, divided into five groups:

Judgements for satisfaction.

$E \Vdash \Phi$	(Guard satisfaction)
$E \Vdash \zeta$	(Effect satisfaction)

Judgements for statics.

$\vdash \mathscr{D}$	(Ok class declaration)
$E \vdash \mathscr{M}$ in D	(Ok method declaration)
$E \vdash \mathscr{F}$	(Ok field declaration)
$E \vdash V : aT$	(Ok value)
$E \vdash M : aT \triangleright Z$	(Ok statement)

Judgements for dynamics.

$E \vdash A$	(Ok process)
$E \vdash \sigma \triangleright E'$	(Ok action)

Supporting judgements.

$E \vdash \diamond$	(Ok environment)
$E \vdash T$	(Ok type)
$E \vdash a$	(Ok annotation)
$E \vdash F : T$	(Ok path)
$E \vdash \Phi$	(Ok guard)
$E \vdash \zeta$	(Ok effect)

Judgements for contexts.

$E \vdash \mathbb{C} \triangleright E'$	(Ok context)

We identify environments up to reordering.

C.1 Judgements for satisfaction

Judgment $E \Vdash \Phi$.

$$\frac{E \ni \texttt{lock}\ F}{E \Vdash F} \qquad \frac{}{E \Vdash \texttt{true}} \qquad \frac{E \Vdash \Phi \quad E \Vdash \Psi}{E \Vdash \Phi\,\texttt{\&\&}\,\Psi} \qquad \frac{E \Vdash \Phi}{E \Vdash \Phi\,\texttt{||}\,\Psi} \qquad \frac{E \Vdash \Psi}{E \Vdash \Phi\,\texttt{||}\,\Psi}$$

$$\frac{E, E'\{\!\![{}^{F}\!/x]\!\!\} \Vdash \Phi}{E, x{:}aT, \texttt{final}\ x{=}F, E' \Vdash \Phi}$$

Judgment $E \Vdash \zeta$.

$$\frac{E \ni \zeta}{E \Vdash \zeta} \qquad \frac{E, E'\{\!\![{}^{F}\!/x]\!\!\} \Vdash \zeta}{E, x{:}aT, \texttt{final}\ x{=}F, E' \Vdash \zeta}$$

C.2 Judgements for statics

Let $\mathbb{F}$ and $\mathbb{G}$ range over sets of field paths. Then any guard Φ can be written in disjuntive normal as $\bigvee_i \mathbb{F}_i$, where each set $\mathbb{F}_i$ is viewed as a conjunction of field paths. We say that Φ *must overlap with* Ψ if $\Phi = \bigvee_i \mathbb{F}_i$, $\Psi = \bigvee_j \mathbb{G}_j$, and $\mathbb{F}_i \cap \mathbb{G}_j \neq \emptyset$ for all i, j.

Judgments $\vdash \mathscr{D}$ and $E \vdash \mathscr{M}$ in D.

$$\frac{\begin{array}{l}\forall i.\ \vec{\chi} \vdash \mathscr{F}_i \\ \forall j.\ \vec{\chi} \vdash \mathscr{M}_j \text{ in } d\langle\vec{\chi}\rangle\end{array}}{\vdash \texttt{class}\ d\langle\vec{\chi}\rangle\texttt{\{}\vec{\mathscr{F}}\,\vec{\mathscr{M}}\texttt{\}}} \qquad \frac{E, \texttt{this}{:}bD, \vec{x}{:}\vec{a}\vec{T} \vdash M : aT}{E \vdash aT\ m\texttt{(}b\,\texttt{this;}\vec{a}\vec{T}\ \vec{x}\texttt{)\{}M\texttt{\}}\text{ in } D}$$

Judgment $E \vdash \mathscr{F}$.

$$\frac{E \vdash T}{E \vdash T\,f\texttt{;}} \qquad \frac{E \vdash a \quad E \vdash D}{E \vdash \texttt{final}\ aD\,f\texttt{;}} \qquad \frac{\begin{array}{l}E \vdash a \quad E \vdash D \quad E \vdash \Phi, \Psi \\ \Phi \textit{ must overlap with } \Psi\end{array}}{E \vdash aD\,f\ \texttt{rdguard}\ \Phi\ \texttt{wrguard}\ \Psi\texttt{;}}$$

Judgment $E \vdash V : aT$.

$$\overset{\text{(VAL-SUB-TYPE)}}{\frac{E \vdash V : aD}{E \vdash V : a\texttt{Object}}} \qquad \overset{\text{(VAL-SUB-ANN1)}}{\frac{E \vdash V : aT}{E \vdash V : \square T}} \qquad \overset{\text{(VAL-SUB-ANN2)}}{\frac{E \vdash V : \circledast T}{E \vdash V : aT}}$$

$$\overset{\text{(VAL-BASE)}}{\frac{E \vdash \diamond \quad bv \text{ inhabits } bt}{E \vdash bv : \square bt}}$$

$$
\begin{array}{rcll}
bt & \in & \mathscr{BT} \supseteq \{\texttt{Unit},\texttt{Bool},\texttt{Int}\} & \text{(Base type)}\\
f,g & \in & \mathscr{FN} & \text{(Field name)}\\
m & \in & \mathscr{MN} \supseteq \{\texttt{start},\texttt{run}\} & \text{(Method name)}\\
d & \in & \mathscr{CN} \supseteq \{\texttt{Object},\texttt{Atomic},\texttt{Lock},\texttt{Cond}\} & \text{(Class name)}\\
\chi & \in & \mathscr{AV} & \text{(Annotation variable)}\\
\gamma & ::= & \boxast \mid \Box & \text{(Ground annotation)}\\
a,b,c & ::= & \chi \mid \gamma & \text{(Annotation)}\\
D & ::= & d\langle\vec{a}\rangle & \text{(Class type)}\\
T,S & ::= & D \mid bt & \text{(Type)}\\
\mathscr{D} & ::= & \texttt{class}\,d\langle\vec{\chi}\rangle\texttt{\{}\vec{\mathscr{F}}\,\vec{\mathscr{M}}\texttt{\}} & \text{(Class declaration)}\\
\mathscr{M} & ::= & a T\; m\texttt{(}b\,\texttt{this;}\vec{a}\vec{T}\;\vec{x}\texttt{)\{}M\texttt{\}} & \text{(Method declaration)}\\
\mathscr{F} & ::= & T\,f\texttt{;} \mid \texttt{final}\,aD\,f\texttt{;} & \text{(Field declaration)}\\
 & & \mid aD\,f\;\texttt{rdguard}\;\Phi\;\texttt{wrguard}\;\Psi\texttt{;} & \\
\Phi,\Psi & ::= & \texttt{true} \mid F \mid \Phi\,\texttt{\&\&}\,\Psi \mid \Phi\,\texttt{||}\,\Psi & \text{(Guard)}\\
F,G & ::= & V \mid F\,.f & \text{(Field path)}\\
bv & \in & \mathscr{BV} \supseteq \{\texttt{unit},\texttt{true},\texttt{false},\texttt{0},\texttt{42}\} & \text{(Base value)}\\
op & \in & \mathscr{OP} \supseteq \{\texttt{==},\texttt{+}\} & \text{(Base value operators)}\\
x,y & \in & \mathscr{VN} \supseteq \{\texttt{this}\} & \text{(Variable name)}\\
p,q,s,t,\ell & \in & \mathscr{ON} & \text{(Object name)}\\
v,w,u & ::= & \texttt{null} \mid bv \mid p & \text{(Ground value)}\\
V,W,U & ::= & v \mid x & \text{(Open value)}\\
M,N & ::= & \texttt{return}\;V\texttt{;} \mid \texttt{val}\;x\;\texttt{=}\;\texttt{new}\;D\texttt{(}\vec{V}\texttt{);}\;M & \text{(Statement)}\\
 & & \mid \texttt{val}\;x\;\texttt{=}\;W\,.m\texttt{(}\vec{V}\texttt{);}\;M \mid \texttt{val}\;x\;\texttt{=}\;op\texttt{(}\vec{V}\texttt{);}\;M & \\
 & & \mid \texttt{val}\;x\;\texttt{=}\;\texttt{(}D\texttt{)}V\texttt{;}\;M \mid \texttt{end}\;V\texttt{;}\;M & \\
 & & \mid \texttt{val}\;x\;\texttt{=}\;V\,.f\texttt{;}\;M \mid V\,.f\;\texttt{=}\;W\texttt{;}\;M & \\
 & & \mid \texttt{if (}V\texttt{) \{}M\texttt{\} else \{}N\texttt{\}} \mid \texttt{val}\;x\;\texttt{=}\;\texttt{sync}\;V\;\texttt{\{}N\texttt{\}}\;M & \\
 & & \mid \texttt{val}\;x\;\texttt{=}\;{}_{s}\texttt{\{}N\texttt{\}}\;M \mid \texttt{val}\;x\;\texttt{=}\;{}_{s}\texttt{\{}N\texttt{\}}\;\ell\texttt{.release();}\;M & \\
\mathbb{E} & ::= & [\![-]\!] \mid \texttt{val}\;x\;\texttt{=}\;{}_{s}\texttt{\{}\mathbb{E}\texttt{\}}\;M & \text{(Statement ctxt)}\\
 & & \mid \texttt{val}\;x\;\texttt{=}\;{}_{s}\texttt{\{}\mathbb{E}\texttt{\}}\;p\texttt{.release();}\;M & \\
\alpha,\beta & ::= & \langle s\texttt{!}p\,.f\texttt{=}v\rangle \mid \langle s\texttt{!}\ell\texttt{:}\,j\rangle \mid \langle s\texttt{!}p\rangle & \text{(Empirical Action)}\\
\phi & ::= & \langle s\texttt{?}p\,.f\texttt{=}v\rangle & \text{(Speculative action)}\\
\sigma,\tau & ::= & \alpha \mid \phi \mid s[\![-]\!] & \text{(Action)}\\
A,B & ::= & p\texttt{:}D \mid \ell\texttt{:Atomic\{}v\texttt{;}\,j\texttt{\}} \mid \ell\texttt{:Lock\{}\vec{s}\texttt{;}\,j\texttt{\}} \mid p\texttt{:Cond\{}\ell\texttt{;}\vec{s}\texttt{;}\vec{t}\texttt{\}} & \text{(Process)}\\
 & & \mid \alpha A \mid s\texttt{[}M\texttt{]} \mid A\,\texttt{|}\,B \mid (\nu p)A \mid \texttt{free}\;p \mid \texttt{runnable}\;p & \\
 & & \mid \top\Rightarrow A\,[\!]\,\phi\Rightarrow B & \\
\mathbb{C} & ::= & [\![-]\!] \mid \mathbb{C}\,\texttt{|}\,A \mid A\,\texttt{|}\,\mathbb{C} \mid (\nu p)\mathbb{C} \mid \alpha\,\mathbb{C} & \text{(Process ctxt)}\\
 & & \mid \top\Rightarrow\mathbb{C}\,[\!]\,\phi\Rightarrow A \mid \top\Rightarrow A\,[\!]\,\phi\Rightarrow\mathbb{C} & \\
\zeta,\xi & ::= & \wr\, p \int & \text{(Effect)}\\
Z & ::= & \{\zeta_1,\ldots,\zeta_n\} & \text{(Set of effects)}\\
E & ::= & \emptyset \mid E,\chi \mid E,x\texttt{:}aT \mid E,p\texttt{:}D \mid E,s\;\texttt{returns}\;aD & \text{(Environment)}\\
 & & \mid E,\texttt{final}\;x{=}F \mid E,\texttt{lock}\;V \mid E,s\texttt{:lock}\;\ell & \\
 & & \mid E,\zeta \mid E,s\texttt{:}\zeta \mid E,\ell\texttt{:}\zeta \mid E,\texttt{onacq}\;\Phi\;\zeta &
\end{array}
$$

Figure 5. Syntactic categories

(VAL-VARIABLE)
$$\frac{E\vdash\diamond \quad E\ni x{:}aT}{E\vdash x : aT}$$

(VAL-NULL)
$$\frac{E\vdash\diamond \quad E\vdash a \quad E\vdash D}{E\vdash\texttt{null} : aD}$$

(VAL-OBJECT-$\Box$)
$$\frac{E\vdash\diamond \quad E\ni p{:}D}{E\vdash p : \Box D}$$

(VAL-OBJECT-$\boxast$)
$$\frac{E\vdash\diamond \quad E\ni p{:}D \quad E\Vdash \wr p\int}{E\vdash p : \boxast D}$$

Judgment $E\vdash M : aT$.

(STAT-NEW)
$$\frac{\begin{array}{l} \mathit{finals}(D)=\vec{f} \quad \mathit{ftype}(D.\vec{f})=\texttt{final}\,\vec{b}\vec{S} \quad E\vdash\vec{V} : \vec{b}\vec{S}\\ E,x{:}\,\boxast\,D\vdash M : aT\end{array}}{E\vdash\texttt{val}\;x\;\texttt{=}\;\texttt{new}\;D\texttt{(}\vec{V}\texttt{);}\;M : aT}$$

(STAT-METHOD)
$$\frac{\begin{array}{l} E\vdash W : cD \quad \mathit{mtype}(D.m)=c;\vec{b}\vec{S}\to bS \quad E\vdash\vec{V} : \vec{b}\vec{S}\\ E,x{:}bS\vdash M : aT\end{array}}{E\vdash\texttt{val}\;x\;\texttt{=}\;W\,.m\texttt{(}\vec{V}\texttt{);}\;M : aT}$$

(STAT-RETURN)
$$\frac{E\vdash V : aT}{E\vdash\texttt{return}\;V\texttt{;} : aT}$$

(STAT-OPERATOR)
$$\frac{\begin{array}{l} op\text{ has type }\vec{bt}\to bt \quad E\vdash\vec{V} : \vec{\Box}\,\vec{bt}\\ E,x{:}\,\Box\,bt\vdash M : aT\end{array}}{E\vdash\texttt{val}\;x\;\texttt{=}\;op\texttt{(}\vec{V}\texttt{);}\;M : aT}$$

(STAT-CAST)
$$\frac{\begin{array}{l} E\vdash V : b\texttt{Object}\\ E,x{:}bD\vdash M : aT\end{array}}{E\vdash\texttt{val}\;x\;\texttt{=}\;\texttt{(}D\texttt{)}V\texttt{;}\;M : aT}$$

(STAT-END)
$$\frac{\begin{array}{l} E\vdash V : \boxast D\\ E\vdash M : aT\end{array}}{E\vdash\texttt{end}\;V\texttt{;}\;M : aT}$$

(STAT-RACING-READ)
$$\frac{\begin{array}{l} E\vdash V : \Box D \quad \mathit{ftype}(D.f)=S\\ E,x{:}\,\Box\,S\vdash M : aT\end{array}}{E\vdash\texttt{val}\;x\;\texttt{=}\;V\,.f\texttt{;}\;M : aT}$$

(STAT-RACING-WRITE)
$$\frac{\begin{array}{l} E\vdash V : \Box D \quad \mathit{ftype}(D.f)=S\\ E\vdash W : \Box S \quad E\vdash M : aT\end{array}}{E\vdash V\,.f\;\texttt{=}\;W\texttt{;}\;M : aT}$$

(STAT-FINAL-READ)
$$\frac{\begin{array}{l} E\vdash V : cD \quad \mathit{ftype}(D.f)=\texttt{final}\,bS\\ E,x{:}b'S,\texttt{final}\;x{=}V\,.f\vdash M : aT\end{array}}{E\vdash\texttt{val}\;x\;\texttt{=}\;V\,.f\texttt{;}\;M : aT} \quad b'=\begin{cases} b, & \text{if } b=c\\ \Box, & \text{otherwise}\end{cases}$$

(R-LOCK-NEW)

$$\frac{}{\begin{array}{l} \quad\mathbb{C}\llbracket \texttt{free}\ \ell \mid \ell\texttt{:Lock} \mid s\texttt{[val}\ x \texttt{ = new Lock(); } M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket \ell\texttt{:Lock\{}\emptyset\texttt{;}2\texttt{\}} \mid \langle s\texttt{!}\ell\rangle\,\langle s\texttt{!}\ell\texttt{:}1\rangle\, s\texttt{[}M\{\!\!\{{}^{\ell}\!/\!_{x}\}\!\!\}\texttt{]}\rrbracket \end{array}}$$

(R-LOCK-CONTEXT)

$$\frac{\mathbb{C}\llbracket s\texttt{[}N\texttt{]}\rrbracket \to \mathbb{C}'\llbracket s\texttt{[}N'\texttt{]}\rrbracket}{\begin{array}{l} \quad\mathbb{C}\,\llbracket s\texttt{[val}\ x \texttt{ = }{}_{s}\texttt{\{}N\texttt{\}}\ \ell\texttt{.release();}M\texttt{]}\rrbracket \\ \to \mathbb{C}'\llbracket s\texttt{[val}\ x \texttt{ = }{}_{s}\texttt{\{}N'\texttt{\}}\ \ell\texttt{.release();}M\texttt{]}\rrbracket \end{array}}$$

(R-LOCK-ACQUIRE)

$$\frac{}{\begin{array}{l} \quad\mathbb{C}\llbracket \ell\texttt{:Lock\{}\emptyset\texttt{;}j\texttt{\}} \mid s\texttt{[val}\ x \texttt{ = sync}\ \ell\ \texttt{\{}N\texttt{\}}\ M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket \ell\texttt{:Lock\{}s\texttt{;}j\texttt{\}} \mid \langle s\texttt{!}\ell\texttt{:}j\rangle\, s\texttt{[val}\ x \texttt{ = }{}_{s}\texttt{\{}N\texttt{\}}\ \ell\texttt{.release(); }M\texttt{]}\rrbracket \end{array}}\ \mathbb{C}\ \text{enables}\langle s\texttt{!}\ell\texttt{:}j\rangle$$

(R-LOCK-RELEASE)

$$\frac{}{\begin{array}{l} \quad\mathbb{C}\llbracket \ell\texttt{:Lock\{}s\texttt{;}j\texttt{\}} \mid s\texttt{[val}\ x \texttt{ = }{}_{s}\texttt{\{}\uparrow v\texttt{\}}\ \ell\texttt{.release(); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket \ell\texttt{:Lock\{}\emptyset\texttt{;}j{+}2\texttt{\}} \mid \langle s\texttt{!}\ell\texttt{:}j{+}1\rangle\, s\texttt{[}M\{\!\!\{{}^{v}\!/\!_{x}\}\!\!\}\texttt{]}\rrbracket \end{array}}$$

(R-LOCK-ACQUIRE-REENTRANT)

$$\frac{}{\begin{array}{l} \quad\mathbb{C}\llbracket \ell\texttt{:Lock\{}\vec{t}\texttt{;}j\texttt{\}} \mid s\texttt{[val}\ x \texttt{ = sync}\ \ell\ \texttt{\{}N\texttt{\}}\ M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket \ell\texttt{:Lock\{}\vec{t}\uplus s\texttt{;}j\texttt{\}} \mid s\texttt{[val}\ x \texttt{ = }{}_{s}\texttt{\{}N\texttt{\}}\ \ell\texttt{.release(); }M\texttt{]}\rrbracket \end{array}}\ s\in\vec{t}$$

(R-LOCK-RELEASE-REENTRANT)

$$\frac{}{\begin{array}{l} \quad\mathbb{C}\llbracket \ell\texttt{:Lock\{}\vec{t}\uplus s\texttt{;}j\texttt{\}} \mid s\texttt{[val}\ x \texttt{ = }{}_{s}\texttt{\{}\uparrow v\texttt{\}}\ \ell\texttt{.release(); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket \ell\texttt{:Lock\{}\vec{t}\texttt{;}j\texttt{\}} \mid s\texttt{[}M\{\!\!\{{}^{v}\!/\!_{x}\}\!\!\}\texttt{]}\rrbracket \end{array}}$$

(R-CONDITION-NEW)

$$\frac{}{\begin{array}{l} \quad\mathbb{C}\llbracket \texttt{free}\ p \mid p\texttt{:Cond} \mid s\texttt{[val}\ x \texttt{ = new Cond(}\ell\texttt{); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket p\texttt{:Cond\{}\ell\texttt{;}\emptyset\texttt{;}\emptyset\texttt{\}} \mid \langle s\texttt{!}p\rangle\, s\texttt{[}M\{\!\!\{{}^{p}\!/\!_{x}\}\!\!\}\texttt{]}\rrbracket \end{array}}\ \mathbb{C}\ni\ell\texttt{:Lock\{}\ldots\texttt{\}}$$

(R-CONDITION-RELEASE)

$$\frac{}{\begin{array}{l} \quad\mathbb{C}\llbracket p\texttt{:Cond\{}\ell\texttt{;}\vec{q}\texttt{;}Q\texttt{\}} \mid \ell\texttt{:Lock\{}\vec{t}\texttt{;}j\texttt{\}} \mid s\texttt{[val}\ x \texttt{ = }p\texttt{.await(); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket p\texttt{:Cond\{}\ell\texttt{;}\vec{q}\uplus\vec{t}\texttt{;}Q\texttt{\}} \mid \ell\texttt{:Lock\{}\emptyset\texttt{;}j{+}2\texttt{\}} \mid \langle\vec{t}\texttt{!}\ell\texttt{:}j{+}1\rangle\, s\texttt{[val}\ x \texttt{ = }p\texttt{.await(); }M\texttt{]}\rrbracket \end{array}}\ s\in\vec{t}$$

(R-CONDITION-SIGNAL)

$$\frac{}{\begin{array}{l} \quad\mathbb{C}\llbracket p\texttt{:Cond\{}\ell\texttt{;}\vec{q}\uplus\vec{s}\texttt{;}Q\texttt{\}} \mid s\texttt{[val}\ x \texttt{ = }p\texttt{.signal(}v\texttt{); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket p\texttt{:Cond\{}\ell\texttt{;}\vec{q}\texttt{;}Q\uplus(\vec{s},v)\texttt{\}} \mid s\texttt{[}M\{\!\!\{{}^{\texttt{unit}}\!/\!_{x}\}\!\!\}\texttt{]}\rrbracket \end{array}}\ \begin{array}{l}\mathbb{C}\ni\ell\texttt{:Lock\{}\vec{t}\texttt{;}j\texttt{\}}\\ s\in\vec{t}\end{array}$$

(R-CONDITION-ACQUIRE)

$$\frac{}{\begin{array}{l} \quad\mathbb{C}\llbracket p\texttt{:Cond\{}\ell\texttt{;}\vec{q}\texttt{;}Q\uplus(\vec{t},v)\texttt{\}} \mid \ell\texttt{:Lock\{}\emptyset\texttt{;}j\texttt{\}} \mid s\texttt{[val}\ x \texttt{ = }p\texttt{.await(); }M\texttt{]}\rrbracket \\ \to \mathbb{C}\llbracket p\texttt{:Cond\{}\ell\texttt{;}\vec{q}\texttt{;}Q\texttt{\}} \mid \ell\texttt{:Lock\{}\vec{t}\texttt{;}j\texttt{\}} \mid \langle s\texttt{!}\ell\texttt{:}j\rangle\, s\texttt{[}M\{\!\!\{{}^{v}\!/\!_{x}\}\!\!\}\texttt{]}\rrbracket \end{array}}\ \begin{array}{l}\mathbb{C}\ \text{enables}\langle s\texttt{!}\ell\texttt{:}j\rangle\\ s\in\vec{t}\end{array}$$

Figure 6. Reduction rules for locks and conditions ($A \to B$)

(STAT-GUARDED-READ)

$$\frac{\begin{array}{l} E\vdash V : \Box D \quad \mathit{ftype}(D.f) = bS\ \texttt{rdguard}\ \Phi \quad E \Vdash \Phi\{\!\!\{{}^{V}\!/\!_{\texttt{this}}\}\!\!\} \\ E,x{:}bS\vdash M : aT \end{array}}{E\vdash \texttt{val}\ x \texttt{ = } V.f\texttt{; }M : aT}$$

(STAT-GUARDED-WRITE)

$$\frac{\begin{array}{l} E\vdash V : \Box D \quad \mathit{ftype}(D.f) = bS\ \texttt{wrguard}\ \Psi \quad E \Vdash \Psi\{\!\!\{{}^{V}\!/\!_{\texttt{this}}\}\!\!\} \\ E\vdash W : bS \\ E\vdash M : aT \end{array}}{E\vdash V.f \texttt{ = } W\texttt{; }M : aT}$$

(STAT-IF)

$$\frac{\begin{array}{l} E\vdash V : \Box\texttt{boolean} \\ E\vdash M : aT \quad E\vdash N : aT \end{array}}{E\vdash \texttt{if (}V\texttt{) \{}M\texttt{\} else \{}N\texttt{\}} : aT}$$

(STAT-LOCK)

$$\frac{\begin{array}{l} E\vdash V : \Box\texttt{Lock} \quad E,\texttt{lock}\ V\vdash N : bS \\ E,x{:}bS\vdash M : aT \end{array}}{E\vdash \texttt{val}\ x \texttt{ = sync}\ V\ \texttt{\{}N\texttt{\}}\ M : aT}$$

(STAT-LOCK-CONTEXT)

$$\frac{\begin{array}{l} E,s\ \texttt{returns}\ bS\vdash N : bS \quad E\vdash p : \Box\texttt{Lock} \\ E,s\ \texttt{returns}\ aT,x{:}bS\vdash M : aT \end{array}}{E,s\ \texttt{returns}\ aT\vdash \texttt{val}\ x \texttt{ = }{}_{s}\texttt{\{}N\texttt{\}}\ p\texttt{.release(); }M : aT}$$

(STAT-METHOD-CONTEXT)

$$\frac{\begin{array}{l} E,s\ \texttt{returns}\ bS\vdash N : bS \\ E,s\ \texttt{returns}\ aT,x{:}bS\vdash M : aT \end{array}}{E,s\ \texttt{returns}\ aT\vdash \texttt{val}\ x \texttt{ = }{}_{s}\texttt{\{}N\texttt{\}}\ M : aT}$$

C.3 Judgements for dynamics

Define $\mathit{locks}(s,E) = \{\texttt{lock}\ V \mid s{:}\texttt{lock}\ V \in E\}$ and $\mathit{effects}(s,E) = \{\zeta \mid s:\zeta \in E\}$. Define $E \Vdash_s \Phi$ to mean $E,\mathit{effects}(s,E),\mathit{locks}(s,E) \Vdash \Phi$, and similarly for $E\vdash_s V : aT$ and $E\vdash_s M : aT$.
Judgment $E\vdash A$.

(PROC-FREE)

$$\frac{E\vdash\diamond \quad E\ni p{:}D}{E\vdash \texttt{free}\ p}$$

(PROC-ALLOCATED)

$$\frac{E\vdash\diamond \quad E\ni p{:}D}{E\vdash p{:}D}$$

(PROC-RUNNABLE)

$$\frac{E\vdash\diamond \quad p\in \mathit{dom}(E)}{E\vdash \texttt{runnable}\ p}$$

(PROC-ATOMIC)

$$\frac{\begin{array}{l} E\vdash\diamond \quad E\ni \ell\texttt{:Atomic} \\ v\in \mathscr{BV}\cup \mathit{dom}(E) \end{array}}{E\vdash \ell\texttt{:Atomic\{}v\texttt{;}j\texttt{\}}}$$

(PROC-LOCK)

$$\frac{\begin{array}{l} E\vdash\diamond \quad E\ni \ell\texttt{:Lock} \\ s\in \mathit{dom}(E) \end{array}}{E\vdash \ell\texttt{:Lock\{}j\texttt{;}s\texttt{\}}}$$

(PROC-CONDITION)

$$\frac{\begin{array}{l} E\vdash\diamond \quad E\ni p\texttt{:Cond} \\ E\ni \ell\texttt{:Lock} \quad \vec{s}\cup\vec{t}\subseteq \mathit{dom}(E) \end{array}}{E\vdash p\texttt{:Cond\{}\ell\texttt{;}\vec{s}\texttt{;}\vec{t}\texttt{\}}}$$

(PROC-THREAD)

$$\frac{E\ni s\ \texttt{returns}\ aT \quad E\vdash_s M : aT}{E\vdash s\texttt{[}M\texttt{]}}$$

(PROC-PAR)

$$\frac{E\vdash A \quad E\vdash B}{E\vdash A\,|\,B}$$

(PROC-NU)

$$\frac{E,p{:}D,p\ \texttt{returns}\ \Box\ \texttt{Unit}\vdash A}{E\vdash (\nu p)A}$$

(PROC-ACTION)

$$\frac{E\vdash \alpha \triangleright E' \quad E'\vdash A}{E\vdash \alpha A}$$

(PROC-SPECULATION)

$$\frac{E\vdash A \quad E\vdash \phi\triangleright E' \quad E'\vdash B}{E\vdash \top\Rightarrow A \mathbin{[\,]} \phi\Rightarrow B}$$

(PROC-SPECULATION-SHAPE-ERROR)

$$\frac{\begin{array}{l} s\in \mathit{dom}(E) \quad E\ni p{:}D \quad E\vdash A \\ \text{Either } \mathit{ftype}(D.f) = \ldots bt\ldots \text{ and } \neg(v \text{ inhabits } bt) \\ \text{or } \quad \mathit{ftype}(D.f) = \ldots D'\ldots \text{ and } E\ni v{:}D'' \text{ and } D''\neq D' \end{array}}{E\vdash \top\Rightarrow A \mathbin{[\,]} \langle s\texttt{?}p.f\texttt{=}v\rangle \Rightarrow B}$$

Judgment $E \vdash \sigma \triangleright E'$.

(ACT-BEGIN)
$$\frac{s \in dom(E) \quad p \in dom(E)}{E \vdash \langle s\,\mathtt{!}\,p\rangle \triangleright E, s{:}\wr p \int}$$

(ACT-ACQUIRE)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni \ell{:}\mathtt{Lock} \\ E' = \left\{ s{:}\zeta \;\middle|\; E \ni \mathtt{onacq}\,\Phi\,\zeta \text{ and } E \Vdash_s \Phi \right\} \end{array}}{E \vdash \langle s\,\mathtt{!}\,\ell{:}\,j\rangle \triangleright E, E', s{:}\mathtt{lock}\,\ell, s{:}\mathit{effects}(\ell, E)} \; j \text{ even}$$

(ACT-RELEASE)
$$\frac{s \in dom(E) \quad E \ni \ell{:}\mathtt{Lock}}{E, s{:}\mathtt{lock}\,\ell \vdash \langle s\,\mathtt{!}\,\ell{:}\,j\rangle \triangleright E, \ell{:}\mathit{effects}(s, E)} \; j \text{ odd}$$

(ACT-$\Box$-RACING-FINAL-WRITE)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \quad \mathit{ftype}(D.f) \in \{S, \mathtt{final}\ \Box\, S\} \\ E \vdash_s v : \Box S \end{array}}{E \vdash \langle s\,\mathtt{!}\,p\,.f\mathtt{=}v\rangle \triangleright E}$$

(ACT-$\boxast$-FINAL-WRITE)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \quad \mathit{ftype}(D.f) = \mathtt{final}\ \boxast\, S \\ E \vdash_s v : \boxast S \end{array}}{E \vdash \langle s\,\mathtt{!}\,p\,.f\mathtt{=}v\rangle \triangleright E}$$

(ACT-$\Box$-GUARDED-WRITE)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \quad \mathit{ftype}(D.f) = \Box S\,\mathtt{wrguard}\,\Psi \\ E \vdash_s v : \Box S \quad E \ni t{:}\wr p \int \text{ implies } E \Vdash_s \Psi\{\![{}^{p}/_{\mathtt{this}}]\!\} \end{array}}{E \vdash \langle s\,\mathtt{!}\,p\,.f\mathtt{=}v\rangle \triangleright E}$$

(ACT-$\boxast$-GUARDED-WRITE)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \quad \mathit{ftype}(D.f) = \boxast S\,\mathtt{wrguard}\,\Psi \\ E \vdash_s v : \boxast S \quad E \ni t{:}\wr p \int \text{ implies } E \Vdash_s \Psi\{\![{}^{p}/_{\mathtt{this}}]\!\} \end{array}}{E \vdash \langle s\,\mathtt{!}\,p\,.f\mathtt{=}v\rangle \triangleright E}$$

(ACT-$\Box$-RACING-FINAL-GUARDED-SPECULATION)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \quad \mathit{ftype}(D.f) \in \{S, \ldots \Box\, S \ldots\} \\ E \vdash_s v : \Box S \end{array}}{E \vdash \langle s\mathtt{?}p\,.f\mathtt{=}v\rangle \triangleright E}$$

(ACT-$\boxast$-FINAL-SPECULATION)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \quad \mathit{ftype}(D.f) = \mathtt{final}\ \boxast\, S \\ E \vdash_s v : \Box S \\ E' = \left\{ t{:}\wr v \int \;\middle|\; E \Vdash_t \wr p \int \right\} \end{array}}{E \vdash \langle s\mathtt{?}p\,.f\mathtt{=}v\rangle \triangleright E, E'}$$

(ACT-$\boxast$-GUARDED-SPECULATION)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \quad \mathit{ftype}(D.f) = \boxast S\,\mathtt{rdguard}\,\Phi \\ E \vdash_s v : \Box S \\ E' = \left\{ t{:}\wr v \int \;\middle|\; E \Vdash_t \Phi\{\![{}^{p}/_{\mathtt{this}}]\!\} \right\} \end{array}}{E \vdash \langle s\mathtt{?}p\,.f\mathtt{=}v\rangle \triangleright E, E', \mathtt{onacq}\,(\Phi\{\![{}^{p}/_{\mathtt{this}}]\!\})\,(\wr v \int)}$$

C.4 Supporting judgments

Define $\mathit{envlocks}(E)$ to return the set of values V such that $\mathtt{lock}\,V$ occurs in E.

Judgment $E \vdash \diamond$.

$$\frac{}{\emptyset \vdash \diamond} \qquad \frac{E \vdash \diamond \quad \chi \notin dom(E)}{E, \chi \vdash \diamond} \qquad \frac{E \vdash \diamond \quad E \vdash D \quad p \notin dom(E)}{E, p{:}D \vdash \diamond}$$

$$\frac{E \vdash \diamond \quad E \vdash \zeta}{E, \zeta \vdash \diamond}$$

$$\frac{E \vdash \diamond \quad E \vdash a \quad E \vdash T \quad x \notin dom(E)}{E, x{:}a\,T \vdash \diamond}$$

$$\frac{E \vdash \diamond \quad E \vdash x : \Box T \quad E \vdash F : T}{E, \mathtt{final}\,x{=}F \vdash \diamond}$$

$$\frac{E \vdash \diamond \quad V \in dom(E) \quad V \notin \mathit{envlocks}(E)}{E, \mathtt{lock}\,V \vdash \diamond}$$

$$\frac{E \vdash \diamond \quad \{s, \ell\} \subseteq dom(E) \quad \ell \notin \mathit{envlocks}(E)}{E, s{:}\mathtt{lock}\,\ell}$$

$$\frac{E \vdash \diamond \quad p \in dom(E) \quad E \vdash \zeta}{E, p{:}\zeta} \qquad \frac{E \vdash \diamond \quad E \vdash \Phi \quad E \vdash \zeta}{E, \mathtt{onacq}\,\Phi\,\zeta}$$

Judgment $E \vdash T$.

$$\frac{}{E \vdash bt} \qquad \frac{d\langle \vec{\chi} \rangle \text{ is defined} \quad E \vdash \vec{a} \quad |\vec{a}| = |\vec{\chi}|}{E \vdash d\langle \vec{a} \rangle}$$

Judgment $E \vdash a$.

$$\frac{}{E \vdash \boxast} \qquad \frac{}{E \vdash \Box} \qquad \frac{\chi \in dom(E)}{E \vdash \chi}$$

Judgment $E \vdash F : T$.

$$\frac{E \vdash V : \Box D}{E \vdash V : D} \qquad \frac{E \vdash F : D \quad \mathit{ftype}(D.f) = \mathtt{final}\,a\,T}{E \vdash F\,.f : T}$$

Judgment $E \vdash \Phi$.

$$\frac{}{E \vdash \mathtt{true}} \qquad \frac{E \vdash F : D}{E \vdash F} \qquad \frac{E \vdash \Phi \quad E \vdash \Psi}{E \vdash \Phi\,\mathtt{\&\&}\,\Psi} \qquad \frac{E \vdash \Phi \quad E \vdash \Psi}{E \vdash \Phi\,\mathtt{||}\,\Psi}$$

Judgment $E \vdash \zeta$.

$$\frac{p \in dom(E)}{E \vdash \wr p \int}$$

C.5 Judgments for contexts

The following judgment is used in proofs to connect contexts and environments. See Appendix D.3.

Judgment $E \vdash \mathbb{C} \triangleright E'$.

(CONTEXT-HOLE) $\qquad$ (CONTEXT-PAR-1) $\qquad$ (CONTEXT-PAR-2)
$$\frac{}{E \vdash [\![-]\!] \triangleright E} \qquad \frac{E \vdash A \quad E \vdash \mathbb{C} \triangleright E'}{E \vdash \mathbb{C}\,|\,A \triangleright E'} \qquad \frac{E \vdash A \quad E \vdash \mathbb{C} \triangleright E'}{E \vdash A\,|\,\mathbb{C} \triangleright E'}$$

(CONTEXT-NU)
$$\frac{E, p{:}D, p\,\mathtt{returns}\ \Box\ \mathtt{Unit} \vdash \mathbb{C} \triangleright E'}{E \vdash (\nu p)\mathbb{C} \triangleright E', p{:}D, p\,\mathtt{returns}\ \Box\ \mathtt{Unit}}$$

(CONTEXT-ACTION)
$$\frac{E \vdash \alpha \triangleright E' \quad E' \vdash \mathbb{C} \triangleright E''}{E \vdash \alpha\mathbb{C} \triangleright E''}$$

(CONTEXT-SPECULATION-INITIAL)
$$\frac{E \vdash \mathbb{C} \triangleright E'' \quad E \vdash \phi \triangleright E' \quad E' \vdash B}{E \vdash \top \Rightarrow \mathbb{C} \,[\!]\, \phi \Rightarrow B \triangleright E''}$$

(CONTEXT-SPECULATION-FINAL)
$$\frac{E \vdash A \quad E \vdash \phi \triangleright E' \quad E' \vdash \mathbb{C} \triangleright E''}{E \vdash \top \Rightarrow A \,[\!]\, \phi \Rightarrow \mathbb{C} \triangleright E''}$$

(CONTEXT-SPECULATION-INITIAL-SHAPE-ERROR)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \quad E \vdash \mathbb{C} \triangleright E' \\ \text{Either } \mathit{ftype}(D.f) = \ldots bt \ldots \text{ and } \neg(v \text{ inhabits } bt) \\ \text{or } \quad \mathit{ftype}(D.f) = \ldots D' \ldots \text{ and } E \ni v{:}D'' \text{ and } D'' \neq D' \end{array}}{E \vdash \top \Rightarrow \mathbb{C} \,[\!]\, \langle s\mathtt{?}p\,.f\mathtt{=}v\rangle \Rightarrow B \triangleright E'}$$

(CONTEXT-SPECULATION-FINAL-SHAPE-ERROR)
$$\frac{\begin{array}{l} s \in dom(E) \quad E \ni p{:}D \quad E \vdash A \\ \text{Either } \mathit{ftype}(D.f) = \ldots bt \ldots \text{ and } \neg(v \text{ inhabits } bt) \\ \text{or } \quad \mathit{ftype}(D.f) = \ldots D' \ldots \text{ and } E \ni v{:}D'' \text{ and } D'' \neq D' \end{array}}{E \vdash \top \Rightarrow A \,[\!]\, \langle s\mathtt{?}p\,.f\mathtt{=}v\rangle \Rightarrow \mathbb{C} \triangleright E}$$

D. Proofs

Soundness follows from progress and preservation. Recall the soundness theorem.

Definition (1) (Stuck thread). Thread s is *stuck* in A if there exists $\mathbb{C}[\![s\,\mathtt{[}\mathbb{E}[\![M]\!]\mathtt{]}]\!] = A$ such that *none* of the following hold.

(1) s has terminated: M is a return statement and $\mathbb{E} = [\![-]\!]$;
(2) s can reduce: $\mathbb{C}[\![s\,\mathtt{[}\mathbb{E}[\![M]\!]\mathtt{]}]\!] \rightarrow \mathbb{C}'[\![s\,\mathtt{[}M'\mathtt{]}]\!]$ for some $\mathbb{C}'$, M';
(3) s has had a class cast exception: $M = \mathtt{val}\ x\ \mathtt{=}\ (D)v\mathtt{;}\ M'$ and $\mathbb{C}$ does not contain subterm $v{:}D$; or
(4) s has had a null pointer exception: M is a method, end, read, write, conditional or synchronization statement with target `null`.
(5) s is waiting for a lock, i.e., M is a synchronization statement or call to `Cond.await`; □

Theorem (2) (Soundness). *Suppose the class table is well typed and method* `Main.main ()` *is defined. Suppose further that A is a* speculation-free *process that is reachable from the bootstrap process, that is*

$$(\nu \mathtt{m})\ \mathtt{m{:}Main} \mid \mathtt{m[m.main();]} \to \cdots \to A.$$

Then A does not contain a stuck thread.

PROOF. By inspection, the bootstrap process is well-formed and well-typed under the empty environment.

As shown in Appendix B, well-formedness is preserved by reduction. By Proposition 30, proved in Appendix D.7 below, well-typedness is preserved by reduction.

By Proposition 21, proved in Appendix D.5 below, no well-formed and well-typed processes may contain a stuck thread. $\Box$

To prove progress and preservation, we first introduce the notion of a *trap*, i.e., a context that contains a bad speculation. In Appendix D.3, we provide composition/decomposition results enabling us to go back and forth between contexts and process. We relate the dynamic and static definitions of happens-before in Appendix D.4. We state progress in Appendix D.5 and give a brief outline of the proof. We prove preservation in two parts, treating structural order in Appendix D.6 and reduction in Appendix D.7.

D.1 Traps

Define $\mathit{lockenv}(E,\mathbb{C})$ as follows.

$$\begin{aligned}
\mathit{lockenv}(E,[\![-]\!]) &= E\\
\mathit{lockenv}(E,\mathbb{C} \mid A) &= \mathit{lockenv}(E,\mathbb{C})\\
\mathit{lockenv}(E,A \mid \mathbb{C}) &= \mathit{lockenv}(E,\mathbb{C})\\
\mathit{lockenv}(E,(\nu p)\mathbb{C}) &= \mathit{lockenv}(E,\mathbb{C})\\
\mathit{lockenv}(E,\top \Rightarrow \mathbb{C} \mathbin{[\,]} \phi \Rightarrow A) &= \mathit{lockenv}(E,\mathbb{C})\\
\mathit{lockenv}(E,\top \Rightarrow A \mathbin{[\,]} \phi \Rightarrow \mathbb{C}) &= \mathit{lockenv}(E,\mathbb{C})\\
\mathit{lockenv}(E,\alpha\mathbb{C}) &= \mathit{lockenv}(E,\mathbb{C}) \quad \text{if } \alpha \text{ nonlock}\\
\mathit{lockenv}(E,\langle s\,\mathtt{!}\,\ell : j\rangle\mathbb{C}) &= \mathit{lockenv}(E,s{:}\mathtt{lock}\ \ell,\mathbb{C}) \quad \text{if } j \text{ even}\\
\mathit{lockenv}(E,s{:}\mathtt{lock}\ \ell,\langle s\,\mathtt{!}\,\ell : j\rangle\mathbb{C}) &= \mathit{lockenv}(E,\mathbb{C}) \quad \text{if } j \text{ odd}
\end{aligned}$$

Define $\mathit{lockenv}(\mathbb{C}) = \mathit{lockenv}(\emptyset,\mathbb{C})$. Define $\mathbb{C} \Vdash_s \Phi$ as shorthand for $\mathit{lockenv}(\mathbb{C}) \Vdash_s \Phi$.

Definition 5 (Trap). A context $\mathbb{C}$ is a *shape trap* if $\mathbb{C} \ni p{:}D$,

$$\mathbb{C} = \mathbb{C}'[\![\top \Rightarrow A \mathbin{[\,]} \langle s?p.f{=}v\rangle \Rightarrow \mathbb{C}'']\!]$$

and either

- $\mathit{ftype}(D.f) = \ldots bt \ldots$ and $\neg(v$ inhabits $bt)$, or
- $\mathit{ftype}(D.f) = \ldots D' \ldots$ and $\neg(E \ni v{:}D')$.

A context $\mathbb{C} = \mathbb{C}'[\![\top \Rightarrow A \mathbin{[\,]} \langle s?p.f{=}v\rangle \Rightarrow \mathbb{C}'']\!]$ is a *trap* if $\mathbb{C} \ni p{:}D$ and either it is a shape trap or

- $\mathit{ftype}(D.f) = \boxast T\ \mathtt{wrguard}\ \Psi$ and there exists $\mathbb{D}[\![\mathbb{C}''']\!] = \mathbb{C}''$ such that (a) $\neg(\mathbb{C}'[\![\mathbb{D}]\!] \Vdash_s \Psi)$, and (b) for every $\mathbb{D}'[\![\mathbb{D}'']\!] = \mathbb{D}$, $\neg(\mathbb{C}'[\![\mathbb{D}']\!], s$ justifies speculation $p.f{=}v)$. $\Box$

The last possibility indicates that s has given up the locks required to justify the speculation and has not done the necessary write. A shape trap can always be typed using PROC-SPECULATION-SHAPE-ERROR.

Lemma 6 (Trap). *Suppose $\mathbb{C}$ is a trap, $\emptyset \vdash \mathbb{C}[\![s\,\mathtt{[}M\mathtt{]}]\!]$, and $\mathbb{C}[\![s\,\mathtt{[}M\mathtt{]}]\!] \to \mathbb{C}'[\![s\,\mathtt{[}M'\mathtt{]}]\!]$. Then, $\mathbb{C}'$ is a trap.*

PROOF. A trap can only be removed by R-SPECULATION-CLOSE. Assume for the sake of contradiction that there exists a trap that can be removed by R-SPECULATION-CLOSE. Thus, we have a speculation of the form $\top \Rightarrow A \mathbin{[\,]} \langle s?p.f{=}v\rangle \Rightarrow \mathbb{C}$ such that $\mathbb{C}'[\![\top \Rightarrow A \mathbin{[\,]} \langle s?p.f{=}v\rangle \Rightarrow \mathbb{C}]\!] \to \mathbb{C}'[\![\mathbb{C}]\!]$ and that (1) $\mathbb{C}', s$ justifies speculation $p.f{=}v$. As (1) holds, we know that there exists some i such that $\sigma_i = \langle s\,\mathtt{!}\,p.f{=}v\rangle$. However, this would require (2) ($\mathit{ftype}(D.f) = \ldots bt \ldots$ and v inhabits bt) or ($\mathit{ftype}(D.f) = \ldots D' \ldots$ and $E \ni v{:}D'$), which contradict the first two premises of the definition of a trap.

For the third possibility, note that by typing rules for actions, s must hold the necessary locks to produce a justifying write action. s cannot acquire the locks to do so after the speculation, since in this case the write cannot leave the speculation (synchronization actions do not commute with speculation). So s must have acquired the necessary locks before the speculation, and must perform the write before releasing the locks, as s cannot acquire them again and subsequently justify the speculation. But this case is exactly that prohibited by the third clause of the definition.

Therefore, the original assumption is false, and the lemma holds. $\Box$

D.2 Weakening/Strengthening

The proofs of the following lemmas follow standard lines.

Lemma 7 (Weakening). *If $E \vdash A$, $E,E' \vdash \diamond$, and E' contains no locks, then $E,E' \vdash A$. Similarly for other judgments.*

Corollary 8. *(a) Assume $p \notin \mathit{fn}(\sigma)$ and $E \vdash \sigma \triangleright E'$. Then, $E,p{:}D \vdash \sigma \triangleright E',p{:}D$.*

(b) Assume $E \vdash A$ and $E \vdash \alpha \triangleright E'$ such that A doesn't have any synchronization actions that overlap with α. Then, $E' \vdash A$.

(c) Assume $E \vdash A$ and $E \vdash \phi \triangleright E'$. Then, $E' \vdash A$. $\Box$

Lemma 9 (Strengthening). *If $E,x{:}bS \vdash M : aT$, and $x \notin M$, then $E \vdash M : aT$.*

If $E,p{:}D,p$ $\mathtt{returns}$ $\Box\,\mathtt{Unit} \vdash A$, and $p \notin A$, then $E \vdash A$.

Corollary 10. *Assume $p \notin \mathit{fn}(\sigma)$ and $E,p{:}D \vdash \sigma \triangleright E',p{:}D,E''$. Then, $E \vdash \sigma \triangleright E',E''$.* $\Box$

Definition 11 (Sensible extension). Suppose environment E is well formed. We define E' *to be a sensible extension of E* if E' is well formed and there exists s_i, ℓ_i, F, and F' such that: $E = F,s_1{:}\mathtt{lock}\ \ell_1,\ldots,s_n{:}\mathtt{lock}\ \ell_n$ and $E' = FF'$. $\Box$

Lemma 12 (Actions preserve wellformed environments). *Suppose $E \vdash \sigma \triangleright E'$ and $E \vdash \diamond$. Then of $E' \vdash \diamond$.*

PROOF. By induction on $E \vdash \sigma \triangleright E'$. $\Box$

Lemma 13 (Actions sensibly extend environments). *Suppose $E \vdash \sigma \triangleright E'$. E' is a sensible extension of E.*

PROOF. We prove this by analysis of the $E \vdash \sigma \triangleright E'$ judgments. We first note that for ACT-$\Box$-RACING-FINAL-WRITE, ACT-$\Box$-GUARDED-WRITE, ACT-$\boxast$-FINAL-WRITE, ACT-$\boxast$-GUARDED-WRITE, and ACT-$\Box$-RACING-FINAL-GUARDED-SPECULATION the result is immediate since $E' = E$. ACT-BEGIN,

ACT-$\boxast$-GUARDED-SPECULATION are almost equally trivial since they simply add to E, which clearly is a sensible extension. Letting $E = F,s{:}\mathtt{lock}\ \ell$ and

$$E' = F,\ell{:}\mathit{effects}(s,E')$$

gives the desired result for ACT-RELEASE. $\Box$

Lemma 14 (Transitivity of sensibly extended environments). *Suppose E' sensibly extends E and E'' sensibly extends E'. Then, E'' sensibly extends E.*

PROOF. Trivial from the definition of sensibly extended environments. $\Box$

Lemma 15 (Sensible environment extensions preserve typability). *If E' sensibly extends E and $E \vdash M$, then $E' \vdash M$.*

PROOF. This follows from the definition of sensible extensions and from Lemma 7. $\Box$

Note, the previous lemma doesn't hold for processes since removal of locks can invalidate it for processes.

D.3 Decomposition/Composition

Lemma 16 (Context decomposition). *Suppose $E \vdash \mathbb{C}[\![A]\!]$ and $\mathbb{C}$ is not a shape trap. Then $\exists E'$ such that $E \vdash \mathbb{C} \triangleright E'$ and $E' \vdash A$. Further, E' is a sensible extension of E.*

PROOF. We prove this by induction on the context, $\mathbb{C}$. First, let us note that as $\mathbb{C}$ is not a shape trap, we know by the definition of shape trap that the context $\mathbb{C}'$ derived from $\mathbb{C}$ in the proof below is also not a shape trap.

Case $\mathbb{C} = \llbracket - \rrbracket$**:** Immediate.

Case $\mathbb{C} = \mathbb{C}' \mid B$**:** By assumption and inversion, $\mathbb{C}'\llbracket A \rrbracket \mid B$ must be typeable by PROC-PAR, and therefore (1) $E \vdash B$ and $E \vdash \mathbb{C}'\llbracket A \rrbracket$. By induction, there exists E' such that (2) $E \vdash \mathbb{C}' \triangleright E'$, (3) $E' \vdash A$, and (4) E' is a sensible extension of E. The result holds by (3), (4), and applying CONTEXT-PAR-1 to (1) and (2).

Case $\mathbb{C} = B \mid \mathbb{C}'$**:** Similar.

Case $\mathbb{C} = (\nu p)\,\mathbb{C}'$**:** By assumption and inversion, $(\nu p)\,\mathbb{C}'\llbracket A \rrbracket$ must be typeable by PROC-NU, giving (1) $E, p\!:\!D, p$ `returns` $\square$ `Unit` $\vdash \mathbb{C}'\llbracket A \rrbracket$. By induction, therefore, there exists E' such that (2) $E, p\!:\!D, p$ `returns` $\square$ `Unit` $\vdash \mathbb{C}' \triangleright E'$, (3) $E' \vdash A$, and (4) E' is a sensible extension of E. The result holds by (3), (4), and applying CONTEXT-NU to (2).

Case $\mathbb{C} = \alpha\mathbb{C}'$**:** By assumption and inversion, $\alpha\mathbb{C}'\llbracket A \rrbracket$ must be typeable by PROC-ACTION, giving (1) $E \vdash \alpha \triangleright E'$ and (2) $E' \vdash \mathbb{C}'\llbracket A \rrbracket$. Further, as Lemma 13 holds, E' sensibly extends E. By induction on (2), there exists (3) E'' that sensibly extends E' such that (4) $E' \vdash \mathbb{C}' \triangleright E''$, and (5) $E'' \vdash A$. By the transitivity of sensibly extended environments, (6) E'' sensibly extends E. By (6), (5), and applying CONTEXT-ACTION to (1) and (4), we get the desired result.

Case $\mathbb{C} = \top \Rightarrow \mathbb{C}' \mathbin{[]} \phi \Rightarrow B$**:** By assumption and inversion, $\top \Rightarrow \mathbb{C}'\llbracket A \rrbracket \mathbin{[]} \phi \Rightarrow B$ must be typeable by PROC-SPECULATION, giving (1) $E \vdash \mathbb{C}'\llbracket A \rrbracket$, (2) $E \vdash \phi \triangleright E'$, and (3) $E' \vdash B$. By induction on (1), there exists (4) E'' that sensibly extends E such that (5) $E \vdash \mathbb{C}' \triangleright E''$ and (6) $E'' \vdash A$. By (4), (6), and applying CONTEXT-SPECULATION-INITIAL to (2), (3), and (5), we get the desired result. It should be noted that PROC-SPECULATION-SHAPE-ERROR and CONTEXT-SPECULATION-INITIAL-SHAPE-ERROR cannot be applied in this case since we assumed that $\mathbb{C}$ was not a shape trap.

Case $\mathbb{C} = \top \Rightarrow B \mathbin{[]} \phi \Rightarrow \mathbb{C}'$**:** By assumption and inversion, $\top \Rightarrow B \mathbin{[]} \phi \Rightarrow \mathbb{C}'\llbracket A \rrbracket$ must be typeable by PROC-SPECULATION, giving (1) $E \vdash B$, (2) $E \vdash \phi \triangleright E'$, and (3) $E' \vdash \mathbb{C}'\llbracket A \rrbracket$. Further, as Lemma 13 holds, E' sensibly extends E. By induction on (3), there exists (4) E'' that sensibly extends E' such that (5) $E' \vdash \mathbb{C}' \triangleright E''$ and (6) $E'' \vdash A$. By the transitivity of sensibly extended environments, (7) E'' sensibly extends E. By (7), (6), and applying CONTEXT-SPECULATION-FINAL to (1), (2), and (5), we get the desired result. It should be noted that PROC-SPECULATION-SHAPE-ERROR and CONTEXT-SPECULATION-FINAL-SHAPE-ERROR cannot be applied in this case since we assumed that $\mathbb{C}$ was not a shape trap.

This completes the possible cases for the context resulting in the lemma being true. $\square$

Lemma 17 (Context composition). *Suppose (a) $E \vdash \mathbb{C} \triangleright E'$ and either (b) $\mathbb{C}$ is a shape trap or (c) $E' \vdash A$. Then $E \vdash \mathbb{C}\llbracket A \rrbracket$.*

PROOF. We prove this by induction on the context, $\mathbb{C}$.

Case $\mathbb{C} = \llbracket - \rrbracket$**:** By assumption, $\mathbb{C}$ is clearly not a shape trap and so it must be the case that (c) holds. The result is immediate from this.

In the remainder of the cases, it is the case that $\mathbb{C}$ could be a shape trap, i.e. that (b) holds. If this is the case, then by definition of a shape trap, any context, $\mathbb{C}'$, derived from $\mathbb{C}$ would also be a shape trap. So for the remainder of the proof we will assume that (b*) if $\mathbb{C}$ is a shape trap, then $\mathbb{C}'$ is a shape trap holds.

Case $\mathbb{C} = \mathbb{C}' \mid B$**:** By applying inversion and CONTEXT-PAR-1 to $E \vdash \mathbb{C} \triangleright E'$, we get (1) $E \vdash B$ and (2) $E \vdash \mathbb{C}' \triangleright E'$. We know by assumption that either (b*) or (c) holds. In either case, we can apply the inductive hypothesis to (2) since either $\mathbb{C}'$ is a shape trap or $E' \vdash A$. This gives (3) $E \vdash \mathbb{C}'\llbracket A \rrbracket$. Applying PROC-PAR to (1) and (3) gives $E \vdash \mathbb{C}'\llbracket A \rrbracket \mid B$ and thus, we have the desired result.

Case $\mathbb{C} = B \mid \mathbb{C}'$**:** Similar except using CONTEXT-PAR-2 instead of CONTEXT-PAR-1.

Case $\mathbb{C} = (\nu p)\,\mathbb{C}'$**:** Since $E \vdash \mathbb{C} \triangleright E'$, and $\mathbb{C} = (\nu p)\,\mathbb{C}'$, (a*) $E \vdash \mathbb{C} \triangleright E'', p\!:\!D, p$ `returns` $\square$ `Unit` holds, since $p\!:\!D, p$ `returns` $\square$ `Unit` must be part of E' by CONTEXT-NU. Applying inversion and CONTEXT-NU to (a*), we get (1) $E, p\!:\!D, p$ `returns` $\square$ `Unit` $\vdash \mathbb{C}' \triangleright E''$. We know by assumption that either (b*) or (c) holds. In either case, we can apply the inductive hypothesis to (1) since either $\mathbb{C}'$ is a shape trap or $E' \vdash A$. This gives (2) $E, p\!:\!D, p$ `returns` $\square$ `Unit` $\vdash \mathbb{C}'\llbracket A \rrbracket$. Applying PROC-NU, we get (3) $E \vdash (\nu p)\,\mathbb{C}'\llbracket A \rrbracket$, the desired result.

Case $\mathbb{C} = \alpha\mathbb{C}'$**:** By applying inversion and CONTEXT-ACTION to (a), we get (1) $E \vdash \alpha \triangleright E''$ and (2) $E'' \vdash \mathbb{C}' \triangleright E'$. We know by assumption that either (b*) or (c) holds. In either case, we can apply the inductive hypothesis to (2) since either $\mathbb{C}'$ is a shape trap or $E' \vdash A$. This gives (3) $E'' \vdash \mathbb{C}'\llbracket A \rrbracket$. Applying PROC-ACTION to (1), and (3), we get (4) $E \vdash \alpha\mathbb{C}'\llbracket A \rrbracket$, the desired result.

Case $\mathbb{C} = \top \Rightarrow \mathbb{C}' \mathbin{[]} \phi \Rightarrow B$**:** There are two cases for which typing rule will type (a).

Case CONTEXT-SPECULATION-INITIAL: By applying inversion and CONTEXT-SPECULATION-INITIAL to (a), we get (1) $E \vdash \mathbb{C}' \triangleright E'$, (2) $E \vdash \phi \triangleright E''$, (3) $E'' \vdash B$. We know by assumption that either (b*) or (c) holds. In either case, we can apply the inductive hypothesis to (1) since either $\mathbb{C}'$ is a shape trap or $E' \vdash A$. This gives (4) $E \vdash \mathbb{C}'\llbracket A \rrbracket$. Applying PROC-SPECULATION to (4), (2), and (3) gives $E \vdash \top \Rightarrow \mathbb{C}'\llbracket A \rrbracket \mathbin{[]} \phi \Rightarrow B$, as desired.

Case CONTEXT-SPECULATION-INITIAL-SHAPE-ERROR: By applying inversion and CONTEXT-SPECULATION-INITIAL-SHAPE-ERROR to (a), we get (1) $s \in \mathit{dom}(E)$, (2) $E \ni p\!:\!D$, (3) $E \vdash \mathbb{C}' \triangleright E'$, and (4) either $\mathit{ftype}(D.f) = \ldots \mathit{bt} \ldots$ and $\neg(v$ inhabits $\mathit{bt})$ or $\mathit{ftype}(D.f) = \ldots D' \ldots$ and $E \ni v\!:\!D''$ and $D'' \neq D'$. We know by assumption that either (b*) or (c) holds. In either case, we can apply the inductive hypothesis to (3) since either $\mathbb{C}'$ is a shape trap or $E' \vdash A$. This gives (5) $E \vdash \mathbb{C}'\llbracket A \rrbracket$. Applying PROC-SPECULATION-SHAPE-ERROR to (1), (2), (5), and (4) gives $E \vdash \top \Rightarrow \mathbb{C}'\llbracket A \rrbracket \mathbin{[]} \phi \Rightarrow B$, as desired.

Since CONTEXT-SPECULATION-INITIAL and CONTEXT-SPECULATION-INITIAL-SHAPE-ERROR are the only two judgments which could type (a) under the current assumption of what shape $\mathbb{C}$ has and the lemma holds for both of these judgments, then the lemma holds for this case.

Case $\mathbb{C} = \top \Rightarrow B \mathbin{[]} \phi \Rightarrow \mathbb{C}'$**:** There are two cases for which typing rule will type (a).

Case CONTEXT-SPECULATION-FINAL: By applying inversion and CONTEXT-SPECULATION-FINAL to (a), we get (1) $E \vdash B$, (2) $E \vdash \phi \triangleright E''$, and (3) $E'' \vdash \mathbb{C}' \triangleright E'$. We know by assumption that either (b*) or (c) holds. In either case, we can apply the inductive hypothesis to (3) since either $\mathbb{C}'$ is a shape trap or $E' \vdash A$. This gives (4) $E'' \vdash \mathbb{C}'\llbracket A \rrbracket$. Applying PROC-SPECULATION to (1), (2), and (4) gives $E \vdash \top \Rightarrow B \mathbin{[]} \phi \Rightarrow \mathbb{C}'\llbracket A \rrbracket$, as desired.

Case CONTEXT-SPECULATION-FINAL-SHAPE-ERROR: By applying inversion and CONTEXT-SPECULATION-FINAL-SHAPE-ERROR to (a), we get (1) $s \in \mathit{dom}(E)$, (2) $E \ni p\!:\!D$, (3) $E \vdash B$, (4) either $\mathit{ftype}(D.f) = \ldots \mathit{bt} \ldots$ and $\neg(v$ inhabits $\mathit{bt})$ or $\mathit{ftype}(D.f) = \ldots D' \ldots$ and $E \ni v\!:\!D''$ and $D'' \neq D'$. Applying PROC-SPECULATION-SHAPE-ERROR to (1), (2), (3), and (4) gives $E \vdash \top \Rightarrow B \mathbin{[]} \phi \Rightarrow \mathbb{C}'\llbracket A \rrbracket$, as desired.

Since CONTEXT-SPECULATION-FINAL and CONTEXT-SPECULATION-FINAL-SHAPE-ERROR are the only two judgments which could type (a) under the current assumption of what shape $\mathbb{C}$ has and the lemma holds for both of these judgments, then the lemma holds for this case.

As we have proved all cases, we have proved the lemma. $\square$

D.4 Happens-before

In non-trap processes, the type system exactly captures the happens-before relation.

Lemma 18 (Happens-before). *Suppose $\emptyset \vdash \mathbb{C} \triangleright E$.*

(a) If $\langle t \,\mathbf{!}\, p\rangle <^{\mathbb{C},s}_{hb} s\llbracket - \rrbracket$ then $E \ni (s\!: \wr p \int)$.
(b) If $\mathbb{C}$ is not a trap and $E \ni (s\!: \wr p \int)$ then $\langle t \,\mathbf{!}\, p\rangle <^{\mathbb{C},s}_{hb} s\llbracket - \rrbracket$.

PROOF. By induction on $\mathbb{C}$. The interesting cases are for speculation, and these follow from the definitions of trap and of justified speculation. $\square$

Lemma 19 (Justified read). *Suppose (a) $\emptyset \vdash \mathbb{C} \triangleright E$, (b) $\mathbb{C}, s$ justifies read $p\,.f$=v and (c) $E \ni p\!:\!D$.*

- *If (d1) $\mathit{ftype}(D.f) = \ldots \square\, T \ldots$ then either (f1) $\mathbb{C}$ is a shape trap or (g1) $E \vdash_s v : \square\, T$.*
- *If (d2) $\mathit{ftype}(D.f) =$ `final` $\boxast\, T$ then either (f2) $\mathbb{C}$ is a shape trap or (g2) $E \vdash_s v : \boxast\, T$.*

- *If (d3) $ftype(D.f) = \boxast T$ `rdguard` Φ and (e3) $E \Vdash_s \Phi$, then either (f3) $\mathbb{C}$ is a shape trap or (g3) $E \vdash_s v : \boxast T$.*

PROOF. The first case is immediate from the definitions.

The other cases are immediate unless the read is justified by a speculation. Then we must show that $E \ni (s : \wr v \int)$.

In the case of a final field, the result is immediate from the typing rule for final speculations.

In the case of a lock-protected field, the result follows from the fact that s holds the locks necessary to read the field. By the typing rule for lock-protected speculations (and lock acquisition) the effect is guaranteed to be there. □

It is worth noting that the environment $E' = \{t : \wr v \int \mid E \Vdash_t \Phi\{\![{}^{p}/_{\texttt{this}}]\!\}\}$ in ACT-$\boxast$-GUARDED-SPECULATION could be restricted to $t \neq s$, since s cannot read the speculated value. But this is not necessary.

For a speculation about a write by s on a lock-protected field, it is only possible for the speculation to finalize if it is justified before the speculation or if s holds the necessary locks at the point of speculation. If s acquires a lock inside a speculation, then the commuting rules do not allow any subsequent write to justify the speculation.

Lemma 20 (Justified speculation). *Suppose $\emptyset \vdash \mathbb{C} \triangleright E$ and $E \vdash \phi \triangleright E'$ and $E' \vdash A$. If $\mathbb{C}, s$ justifies speculation ϕ then $E \vdash A$.*

PROOF. In general, E' extends E by adding immediate and latent effects. Otherwise the environments are the same.

For final fields, only one write can possibly justify the speculation, and this write happens-before $\wr p \int$. If thread writing $p.f$ had $\wr v \int$, then so will any thread that has $\wr p \int$.

For lock-protected fields, the justifying write must occur while holding $\wr v \int$ and the relevant write locks. Any reader must have the corresponding read locks. Since these must overlap, one can reason, via the definition of $\emptyset \vdash \mathbb{C} \triangleright E$, that any occurrence of v has the required effect. □

D.5 Progress

Recall the definition of a stuck thread

Definition (1) (Stuck thread). Thread s is *stuck* in A if there exists $\mathbb{C}\llbracket s\,\texttt{[}\mathbb{E}\llbracket M\rrbracket\texttt{]}\rrbracket = A$ such that none of the following hold.

(1) s has terminated, i.e, M is a return statement and $\mathbb{E} = \llbracket - \rrbracket$;
(2) s can reduce, i.e.; $\mathbb{C}\llbracket s\,\texttt{[}\mathbb{E}\llbracket M\rrbracket\texttt{]}\rrbracket \to \mathbb{C}'\llbracket s\,\texttt{[}M'\texttt{]}\rrbracket$ for some $\mathbb{C}'$ and M';
(3) s is waiting for a lock, i.e., M is a synchronization statement or call to `Cond.await`;
(4) s has had a class cast exception, i.e., $M = \texttt{val}\ x \texttt{=} (D)v\texttt{;}\ M'$ and $\mathbb{C}$ does not contain subterm $v : D$; or
(5) s has had a null pointer exception, i.e., M is a method, end, read, write, conditional or synchronization statement with target `null`. □

Proposition 21 (Progress). *If A contains no speculation and $\emptyset \vdash A$ then A contains no stuck thread.*

PROOF. For a contradiction, suppose that s is stuck in A and $A = \mathbb{C}\llbracket s\,\texttt{[}M\texttt{]}\rrbracket$.

Note that $\mathbb{C}$ cannot be a trap, since it contains no speculation.

Applying decomposition and inversion to the supposition, we have that there exists E, a and T such that (1) $\emptyset \vdash \mathbb{C} \triangleright E$ and (2) $E \vdash_s M : aT$. By induction on judgment (2), one can show a contradiction for any M.

The only unusual case is R-END, and this follows from Lemma 18. □

D.6 Structural order preserves typing

We begin with some lemmas.

Lemma 22 (Action permutation). *Let $\vec{\sigma} \triangleright \vec{\tau}$ where $s\llbracket - \rrbracket \notin \vec{\sigma}$ and $s\llbracket - \rrbracket \notin \vec{\tau}$ and $E \vdash \vec{\sigma} \triangleright E'$. Then, $E \vdash \vec{\tau} A$.*

PROOF. By induction on the definition of the $\triangleright$ precongruence. □

Lemma 23 (Action-speculation permutation). *If $\phi\alpha \triangleright \alpha\phi$ or $thrd(\phi) \neq thrd(\alpha)$, and $E \vdash \phi \triangleright E'$, $E' \vdash \alpha \triangleright E'''$, and $E \vdash \alpha \triangleright E''$, then $E'' \vdash \phi \triangleright E'''$.*

PROOF. This can be proven by induction on the $\triangleright$ precongruence and a case analysis of α and ϕ. □

Lemma 24 (Substitution). *1. If $E, x{:}D \vdash M : C$ and $E \vdash v : D$, then $E \vdash M\{\![{}^{v}/_{x}]\!\} : C$.*

2. Assume $E \vdash p : cD$, $mtype(D.m) = c;\vec{b}\vec{S} \to bS$, $E \vdash \vec{V} : \vec{b}\vec{S}$, $E, x{:}bS \vdash M : aT$, $E \vdash W : bS$, and W is the result of $p.m(\vec{V})$. Then $E \vdash M\{\![{}^{W}/_{x}]\!\} : aT$.

3. Assume $E \vdash p : cD$, $mtype(D.m) = c;\vec{b}\vec{S} \to bS$, $mbody(D.m) = \lambda\vec{y}.M$, and $E \vdash \vec{V} : \vec{b}\vec{S}$. Then $E \vdash M\{\![{}^{p}/_{\texttt{this}}]\!\}\{\![{}^{\vec{V}}/_{\vec{y}}]\!\} : aT$.

PROOF. Each of these can be proved using induction on $E \vdash M : C$ □

We now explicitly state the rules for defining $A \geqq B$ as the least precongruence on processes that satisfy the axioms in Equation 6.1. They are as follows:

$$\text{(S-REFLEXIVITY)}\ \frac{}{A \geqq A} \qquad \text{(S-TRANSITIVITY)}\ \frac{A \geqq A' \quad A' \geqq B}{A \geqq B} \qquad \text{(S-CTXT-ACTION)}\ \frac{A \geqq A'}{\alpha A \geqq \alpha A'}$$

$$\text{(S-CTXT-PAR1)}\ \frac{A \geqq A'}{A \,|\, B \geqq A' \,|\, B} \qquad \text{(S-CTXT-PAR2)}\ \frac{B \geqq B'}{A \,|\, B \geqq A \,|\, B'} \qquad \text{(S-CTXT-NU)}\ \frac{A \geqq A'}{(\nu p)A \geqq (\nu p)A'}$$

$$\text{(S-CTXT-SPECULATION1)}\ \frac{A \geqq A'}{\top \Rightarrow A \,[\!]\, \phi \Rightarrow B \geqq \top \Rightarrow A' \,[\!]\, \phi \Rightarrow B} \qquad \text{(S-CTXT-SPECULATION2)}\ \frac{B \geqq B'}{\top \Rightarrow A \,[\!]\, \phi \Rightarrow B \geqq \top \Rightarrow A \,[\!]\, \phi \Rightarrow B'}$$

Proposition 25 (Structural order preserves typing). *If $E \vdash A$, and $A \geqq B$, then $E \vdash B$.*

PROOF. Assume that $E \vdash A$, and $A \geqq B$. We prove this by induction on the structural order rules found in Equation 6.1 and on the ones explicitly stated above. For each case, we then use induction on the typing rules which are valid for that structural order.

Case S-REFLEXIVITY:
Assume that $A = B$. Therefore, the results holds trivially.

Case S-TRANSITIVITY:
By inversion on S-TRANSITIVITY, we get (1) $A \geqq A'$ and (2) $A' \geqq B$ for all A'. By the induction hypothesis on (1) and since $E \vdash A$, (3) $E \vdash A'$. Now this, together with the induction hypothesis on (2), means that (4) $E \vdash B$, as desired.

Case S-CTXT-. . . :
In each case, the result follows using inversion and induction.

Case S-NU-FREE ($\to$):
Assume that $B = A \,|\, (\nu p)\,(p{:}d\langle\vec{\gamma}\rangle \,|\, \texttt{free}\ p)$. We can further safely assume that (a) $d\langle\vec{\chi}\rangle$ is defined, (b) $p \notin dom(E)$, and that (c) $|\vec{\gamma}| = |\vec{\chi}|$. Otherwise, we can simply pick a different p and d which do satisfy these assumptions. We know by assumption that $E \vdash A$, so (1) $E \vdash \diamond$. By the rules for typing $E \vdash \boxast$ and $E \vdash \square$, we know that (2) $E \vdash \vec{\gamma}$. By (a), (2), and (c), we know that (3) $E \vdash d\langle\vec{\gamma}\rangle$ holds. As we have (1), (3), and (b), we know that (4) $E, p{:}d\langle\vec{\gamma}\rangle, p\ \texttt{returns}\ \square\ \texttt{Unit} \vdash \diamond$ holds. Further, by applying PROC-FREE to (1) and (4), we have (5) $E, p{:}d\langle\vec{\gamma}\rangle, p\ \texttt{returns}\ \square\ \texttt{Unit} \vdash \texttt{free}\ p$. We can also apply PROC-ALLOCATED to (1) and (4), giving us (6) $E, p{:}d\langle\vec{\gamma}\rangle, p\ \texttt{returns}\ \square\ \texttt{Unit} \vdash p{:}d\langle\vec{\gamma}\rangle$. By applying PROC-PAR to (6) and (5), we get (7) $E, p{:}d\langle\vec{\gamma}\rangle, p\ \texttt{returns}\ \square\ \texttt{Unit} \vdash (p{:}d\langle\vec{\gamma}\rangle \,|\, \texttt{free}\ p)$. By applying PROC-NU to (7), we get (8) $E \vdash (\nu p)\,(p{:}d\langle\vec{\gamma}\rangle \,|\, \texttt{free}\ p)$. Finally, by applying PROC-PAR to $E \vdash A$ and (8), we get (9) $E \vdash A \,|\, (\nu p)\,(p{:}d\langle\vec{\gamma}\rangle \,|\, \texttt{free}\ p)$, as desired.

Case S-NU-FREE ($\leftarrow$):
Assume that $A = A' \,|\, (\nu p)\,(p{:}d\langle\vec{\gamma}\rangle \,|\, \texttt{free}\ p)$ and $B = A'$. We know by assumption that $E \vdash A$. By applying inversion and PROC-PAR on $E \vdash A$, we get (1) $E \vdash A'$ and (2) $E \vdash (\nu p)\,(p{:}d\langle\vec{\gamma}\rangle \,|\, \texttt{free}\ p)$. The result holds from (1).

Case S-NU-PAR ($\to$):
Assume $A = (\nu p)\,(B' \,|\, A')$ and $B = B' \,|\, ((\nu p)A')$. By inversion and PROC-NU on $E \vdash A$, we have (1) $E, p{:}D, p\ \texttt{returns}\ \square\ \texttt{Unit} \vdash (B' \,|\, A')$. By inversion and PROC-PAR on (1), we have (2) $E, p{:}D, p\ \texttt{returns}\ \square\ \texttt{Unit} \vdash B'$ and (3) $E, p{:}D, p\ \texttt{returns}\ \square\ \texttt{Unit} \vdash A'$. Now, by assumption of S-NU-PAR, we know that $p \notin fn(B')$. Thus, we can apply Corollary 10 to (2), giving (4) $E \vdash B'$. By applying PROC-NU to (3), we get (5) $E \vdash ((\nu p)A')$. Applying PROC-PAR to (4) and (5) gives $E \vdash B' \,|\, ((\nu p)A')$, as desired.

Case S-NU-PAR ($\leftarrow$):
Similar except use Corollary 8 instead of Corollary 10.

Case S-PAR-PAR ($\rightarrow$):
Case S-PAR-PAR ($\leftarrow$):
Case S-PAR ($\rightarrow$):
Case S-PAR ($\leftarrow$):
Trivial through application of inversion and PROC-PAR.
Case S-NU-NU ($\rightarrow$):
Case S-NU-NU ($\leftarrow$):
Trivial through application of inversion and PROC-NU, Lemma 7, Corollary 8, Lemma 9, and Corollary 10.

Case S-NU-PREFIX ($\rightarrow$):
Let $A = (\nu p)\,\alpha A'$ and $B = \alpha(\nu p)A'$ such that $p \notin fn(\alpha)$. Applying inversion and PROC-NU to $E \vdash A$ gives (1) $E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit} \vdash \alpha A'$. Applying inversion and PROC-ACTION to (1) gives (2) $E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit} \vdash \alpha \triangleright E'$, where $E' = E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit}, E''$, and (3) $E' \vdash A'$. Applying Corollary 8 followed by Corollary 10 to (2) gives (4) $E \vdash \alpha \triangleright E, E''$, which in turn means that (5) $E' = E, E'', p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit}$. Thus, we have (6) $E, E'', p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit} \vdash A'$, by substituting in for E' in (3). Applying PROC-NU to (6) gives (7) $E, E'' \vdash (\nu p)A'$. Applying PROC-ACTION on (4) and (7) gives $E \vdash \alpha(\nu p)A'$, the desired result.

Case S-NU-PREFIX ($\leftarrow$):
Similar.

Case S-PREFIX-PAR:
Let $A = A' \,|\, (\alpha A'')$ and $B = \alpha(A' \,|\, A'')$. We know by applying inversion and PROC-PAR to A that (1) $E \vdash A'$ and (2) $E \vdash \alpha A''$ hold. Applying inversion and PROC-ACTION to (2) gives (3) $E \vdash \alpha \triangleright E'$ and (4) $E' \vdash A''$. We know by the well-formedness of A that $\mathbb{C}$ is contiguous. Thus, we know that if α is a synchronization action, then it can't overlap with A. Thus, by Corollary 8, we have (5) $E' \vdash A$. Applying inversion and PROC-PAR to (5) gives (6) $E' \vdash A'$ and (7) $E' \vdash \alpha A''$. Applying PROC-PAR to (6) and (4) gives (8) $E' \vdash A' \,|\, A''$. Applying PROC-ACTION to (3) and (8) gives (9) $E \vdash \alpha(A' \,|\, A'')$, as desired.

Case S-PREFIX:
Let $A = \vec{\alpha} A'$ and $B = \vec{\beta} A'$ such that $\vec{\alpha} \triangleright \vec{\beta}$. We know by assumption that $E \vdash A$. By successive applications of inversion and PROC-ACTION, we have (1) $E \vdash \vec{\alpha} \triangleright E'$ and (2) $E' \vdash A'$. By applying Lemma 22, we have (3) $E \vdash \vec{\beta} A'$, as desired.

Case S-PREFIX-SPECULATION:
Let $A = \top \Rightarrow (\alpha A') \mathbin{[\!]} \phi \Rightarrow (\alpha A'')$ and $B = \alpha(\top \Rightarrow A' \mathbin{[\!]} \phi \Rightarrow A'')$ such that (1) $\phi\alpha \triangleright \alpha\phi$ or (2) $thrd(\phi) \neq thrd(\alpha)$. As $E \vdash A$, there are two typing rules which can be applied here, PROC-SPECULATION and PROC-SPECULATION-SHAPE-ERROR. We consider both cases to complete the proof of this case.

Case PROC-SPECULATION:
Assume that PROC-SPECULATION types $E \vdash A$. By inversion and PROC-SPECULATION, we have (3a) $E \vdash \alpha A'$, (4a) $E \vdash \phi \triangleright E'$, and (5a) $E' \vdash \alpha A''$. Applying inversion and PROC-ACTION to (3a) gives (6a) $E \vdash \alpha \triangleright E''$ and (7a) $E'' \vdash A'$. Applying inversion and PROC-ACTION to (5a) gives (8a) $E' \vdash \alpha \triangleright E'''$ and (9a) $E''' \vdash A''$. Applying Lemma 23 to (1), (2), (4a), (8a), and (6a) gives (11a) $E'' \vdash \phi \triangleright E'''$. Applying PROC-SPECULATION to (7a), (11a), and (7a) gives (12a) $E'' \vdash \top \Rightarrow A' \mathbin{[\!]} \phi \Rightarrow A''$. Applying PROC-ACTION to (6a) and (12a) gives $E \vdash \alpha(\top \Rightarrow A' \mathbin{[\!]} \phi \Rightarrow A'')$, as desired.

Case PROC-SPECULATION-SHAPE-ERROR:
Assume that PROC-SPECULATION-SHAPE-ERROR types $E \vdash A$. Thus, $\phi = \langle s?p\,.f{=}v \rangle$. By inversion and PROC-SPECULATION-SHAPE-ERROR, we have (3b) $s \in dom(E)$, (4b) $E \ni p{:}D$, (5b) either ($ftype(D.f) = \ldots bt \ldots$ and $\neg$(v inhabits bt)) or ($ftype(D.f) = \ldots D' \ldots$ and $E \ni v{:}D''$ and $D'' \neq D'$) and (6b) $E \vdash \alpha A'$. Applying inversion and PROC-ACTION to (6b) gives (7b) $E \vdash \alpha \triangleright E'$ and (8b) $E' \vdash A'$. It is clear that if (3b), (4b), and (5b) hold, then (9b) $s \in dom(E')$, (10b) $E' \ni p{:}D$, and (11b) either ($ftype(D.f) = \ldots bt \ldots$ and $\neg$(v inhabits bt)) or ($ftype(D.f) = \ldots D' \ldots$ and $E' \ni v{:}D''$ and $D'' \neq D'$) also must hold. Therefore, we can apply PROC-SPECULATION-SHAPE-ERROR to (9b), (10b), (8b), and (11b) to get (12b) $E' \vdash \top \Rightarrow A' \mathbin{[\!]} \phi \Rightarrow A''$. Applying PROC-ACTION to (7b) and (12b) gives (13b) $E \vdash \alpha(\top \Rightarrow A' \mathbin{[\!]} \phi \Rightarrow A'')$, as desired.

Since we showed that for both cases of typing $E \vdash A$ results in $E \vdash B$ being typed, we have shown that this case is satisfied as well.

Case S-NU-SPECULATION ($\rightarrow$):
Let $A = (\nu p)\,(\top \Rightarrow A' \mathbin{[\!]} \phi \Rightarrow A'')$ and $B = \top \Rightarrow ((\nu p)A') \mathbin{[\!]} \phi \Rightarrow ((\nu p)A'')$ such that $p \notin fn(\phi)$. Applying inversion and PROC-NU to $E \vdash A$ gives (1) $E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit} \vdash \top \Rightarrow A' \mathbin{[\!]} \phi \Rightarrow A''$. There are two typing rules which can be applied here, PROC-SPECULATION and PROC-SPECULATION-SHAPE-ERROR. We consider both cases to complete the proof of this case.

Case PROC-SPECULATION:
Applying inversion and PROC-SPECULATION to (1) gives (2a) $E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit} \vdash A'$, (3a) $E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit} \vdash \phi \triangleright E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit}, E'$, and (4a) $E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit}, E' \vdash A''$. Applying Corollary 8 followed by Corollary 10 to (3a) gives (5a) $E \vdash \phi \triangleright E, E'$ and (6a) $E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit} \vdash \phi \triangleright E, E', p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit}$, which means that (7a) $E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit}, E' = E, E', p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit}$ holds as well. Substituting in (4a) gives (8a) $E, E', p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit} \vdash A''$. Applying PROC-NU to both (2a) and (8a) gives (9a) $E \vdash (\nu p)A'$ and (10a) $E, E' \vdash (\nu p)A''$, respectively. Applying PROC-SPECULATION to (9a), (5a), and (10a) gives (11a) $E \vdash \top \Rightarrow ((\nu p)A') \mathbin{[\!]} \phi \Rightarrow ((\nu p)A'')$, as desired.

Case PROC-SPECULATION-SHAPE-ERROR:
Applying inversion and PROC-SPECULATION-SHAPE-ERROR to (1) where $\phi = \langle s?q\,.f{=}v \rangle$ gives (2a) $s \in dom(E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit})$, (3a) $E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit} \ni q{:}D$ with $p \neq q$, (4a) either ($ftype(D.f) = \ldots bt \ldots$ and $\neg$(v inhabits bt)) or ($ftype(D.f) = \ldots D' \ldots$ and $E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit} \ni v{:}D''$ and $D'' \neq D'$) and (5a) $E, p{:}D, p\ \texttt{returns}\ \Box\ \texttt{Unit} \vdash A'$. Applying PROC-NU to (5a) gives (6a) $E \vdash (\nu p)A'$. Applying PROC-SPECULATION-SHAPE-ERROR to (2a), (2a), (6a), and (4a) gives (7a) $E \vdash \top \Rightarrow ((\nu p)A') \mathbin{[\!]} \phi \Rightarrow ((\nu p)A'')$, as desired. Note, we can just put in the (νp) before A'' in the speculation since the speculation side is not typed by PROC-SPECULATION-SHAPE-ERROR.

Case S-NU-SPECULATION ($\leftarrow$):
Similar, but reversing the steps.
Case S-PAR-SPECULATION ($\rightarrow$):
Similar to the S-NU-SPECULATION ($\rightarrow$) case with the exception of using PROC-PAR instead of PROC-NU.
Case S-PAR-SPECULATION ($\leftarrow$):
Similar to the S-NU-SPECULATION ($\leftarrow$) case with the exception of using PROC-PAR instead of PROC-NU.

As we have proved all cases, we have proved the proposition. $\Box$

D.7 Reduction preserves typing

Lemma 26 (Shape traps preserve typability). *If (1) $\emptyset \vdash \mathbb{C}[\![s\,\texttt{[}M\texttt{]}]\!]$, (2) $\mathbb{C}$ is a shape trap, and (3) $\mathbb{C}[\![s\,\texttt{[}M\texttt{]}]\!] \rightarrow \mathbb{C}'[\![s\,\texttt{[}M'\texttt{]}]\!]$, then $\emptyset \vdash \mathbb{C}'[\![s\,\texttt{[}M'\texttt{]}]\!]$.*

PROOF. Let us assume (1), (2), and (3). Then we know that $\mathbb{C}'$ is also a shape trap since traps can never be removed. So, by the definition of a shape trap, $\emptyset \vdash \mathbb{D} \triangleright E$ and $\mathbb{C}' = \mathbb{D}[\![\top \Rightarrow A' \mathbin{[\!]} \langle s?p\,.f{=}v \rangle \Rightarrow \mathbb{D}']\!]$ where (4) $\top \Rightarrow A' \mathbin{[\!]} \langle s?p\,.f{=}v \rangle \Rightarrow \mathbb{D}'$ is a shape trap with respect to E. By the definition of a shape trap on (4), (5) $E \vdash \top \Rightarrow A' \mathbin{[\!]} \langle s?p\,.f{=}v \rangle \Rightarrow \mathbb{D}'$. By applying CONTEXT-SPECULATION-FINAL-SHAPE-ERROR to the premises gotten by applying inversion and PROC-SPECULATION-SHAPE-ERROR to (5), we have (6) $E \vdash \top \Rightarrow A' \mathbin{[\!]} \langle s?p\,.f{=}v \rangle \Rightarrow \mathbb{D}' \triangleright E$. Combining (6) with $\emptyset \vdash \mathbb{D} \triangleright E$, gives (7) $\emptyset \vdash \mathbb{C}' \triangleright E$. By applying Lemma 17 on (7) and the fact that $\mathbb{C}'$ is a shape trap, we get (8) $\emptyset \vdash \mathbb{C}'[\![s\,\texttt{[}M'\texttt{]}]\!]$, as desired. $\Box$

Lemma 27. (Thread reduction introduces a shape trap or sensibly extends the environment). *If $\emptyset \vdash \mathbb{C}[\![s\,\texttt{[}M\texttt{]}]\!]$, $\mathbb{C}[\![s\,\texttt{[}M\texttt{]}]\!] \rightarrow \mathbb{C}'[\![s\,\texttt{[}M'\texttt{]}]\!]$, and $\emptyset \vdash \mathbb{C} \triangleright E$, then either ($\emptyset \vdash \mathbb{C}' \triangleright E'$ and E' sensibly extends E) or $\mathbb{C}'$ is a shape trap.*

PROOF. If $\mathbb{C}$ is a shape trap, we are done since $\mathbb{C}'$ must also be a shape trap since traps can never be removed. If a shape trap is introduced to $\mathbb{C}'$ during the reduction step, we are again done. Therefore the interesting part of the proof is when $\mathbb{C}$ and $\mathbb{C}'$ are not shape traps. This can be proven by induction on reduction rules found in Equation 6.1, Equation 6.1, and Figure 6. $\Box$

Lemma 28 (Thread return type existence). *If $E \vdash s\,\texttt{[}M\texttt{]}$, then $E = E', s\ \texttt{returns}\,a\,T$, for some E'.*

PROOF. Assume $E \vdash s[M]$. Applying inversion and PROC-THREAD gives (1) $E \ni s\ \mathtt{returns}\ aT$ and (2) $E \vdash_s M : aT$. Since we have (1) we can reorder E so that $E = E', s\ \mathtt{returns}\ aT$, as desired. □

Lemma 29 (Thread return types). *If $(\nu s) \in \mathbb{C}$, then if $E \vdash \mathbb{C} \rhd E', s\ \mathtt{returns}\ aT$, then $E \vdash \mathbb{C} \rhd E', s\ \mathtt{returns}\ bS$.*

If $(\nu s) \notin \mathbb{C}$, then if $E, s\ \mathtt{returns}\ aT \vdash \mathbb{C} \rhd E', s\ \mathtt{returns}\ aT$, then $E, s\ \mathtt{returns}\ bS \vdash \mathbb{C} \rhd E', s\ \mathtt{returns}\ bS$.

PROOF. The first case follows from CONTEXT-NU. The second case follows from the fact that $\mathbb{C}$ doesn't modify the return type of a thread. □

We now state and prove preservation, also known as subject reduction.

Proposition 30 (Preservation). *If $\emptyset \vdash A$, and $A \to B$, then $\emptyset \vdash B$.*

PROOF. Assume that (1) $\emptyset \vdash A$ and (2) $A \to B$. We prove this by induction on the reduction rules found in Equation 6.1, Equation 6.1, and Figure 6. That is, we prove this by induction on (2). We start with the case of R-STRUCTURAL-ORDER.

Case R-STRUCTURAL-ORDER: Assume that R-STRUCTURAL-ORDER satisfies (2). Then by assumption, we know that (3) $A \geqq A'$, (4) $A' \to B'$, and (5) $B' \geqq B$. Applying Proposition 25 on (1) and (3) gives (6) $\emptyset \vdash A'$. Applying the inductive hypothesis on (6) and (4) gives (7) $\emptyset \vdash B'$. Applying Proposition 25 on (7) and (5) gives (8) $\emptyset \vdash B$, as desired.

We partition the remaining cases into two sets that we need to consider. One set of cases is comprised of the reduction rules which modify the context while reducing, namely R-METHOD-CONTEXT, and R-LOCK-CONTEXT. The other set consists of all remaining cases and do not modify the context. Let us first consider the cases for the rules which modify the context, as they are more interesting.

Case R-METHOD-CONTEXT: Assume that R-METHOD-CONTEXT satisfies (2). So, (3) $A = \mathbb{C}\llbracket s[\mathtt{val}\ x =_s \{N\} M] \rrbracket$, (4) $B = \mathbb{C}'\llbracket s[\mathtt{val}\ x =_s \{N'\} M] \rrbracket$. By the equalities of (3) and (4) combined with (2), we have (5) $\mathbb{C}\llbracket s[\mathtt{val}\ x =_s \{N\} M] \rrbracket \to \mathbb{C}'\llbracket s[\mathtt{val}\ x =_s \{N'\} M] \rrbracket$, and (6) $\mathbb{C}\llbracket s[N] \rrbracket \to \mathbb{C}'\llbracket s[N'] \rrbracket$. Now there are two cases to consider. Either $\mathbb{C}$ is a shape trap or it isn't. If it is, then by Lemma 26, $\emptyset \vdash \mathbb{C}'\llbracket s[N'] \rrbracket$. Now since $B = \mathbb{C}'\llbracket s[N'] \rrbracket$, we have $\emptyset \vdash B$, as desired. Thus, if $\mathbb{C}$ is a shape trap, we are done. We proceed assuming that $\mathbb{C}$ is not a shape trap.
By applying Lemma 16 on (1) and (3), we have (7) $\emptyset \vdash \mathbb{C} \rhd E$, (8) $E \vdash s[\mathtt{val}\ x =_s \{N\} M]$, and (9) E is a sensible extension of $\emptyset$. By applying inversion and PROC-THREAD on (8), we get (10) $E \ni s\ \mathtt{returns}\ aT$ and (11) $E \vdash_s \mathtt{val}\ x =_s \{N\} M : aT$. Let (12) $E', s\ \mathtt{returns}\ aT$ be a reordering of E. Further, by using inversion and STAT-METHOD-CONTEXT on (11), we know that (13) $E', s\ \mathtt{returns}\ bS \vdash_s N : bS$ and (14) $E', s\ \mathtt{returns}\ aT, x{:}bS \vdash_s M : aT$. Using PROC-THREAD on (13) gives (15) $E', s\ \mathtt{returns}\ bS \vdash s[N]$. Substituting (12) into (3), gives (16) $\emptyset \vdash \mathbb{C} \rhd E', s\ \mathtt{returns}\ aT$. Now it must be the case that $(\nu s) \in \mathbb{C}$ since it clearly can't be in $\emptyset$.
Applying Lemma 29 to (16) gives (17a) $\emptyset \vdash \mathbb{C} \rhd E', s\ \mathtt{returns}\ bS$. Applying Lemma 17 on (17a) and (15) gives (18a) $\emptyset \vdash \mathbb{C}\llbracket s[N] \rrbracket$. By applying the inductive hypothesis on (18a), (6), we know that (19a) $\emptyset \vdash \mathbb{C}'\llbracket s[N'] \rrbracket$. Applying Lemma 16 to (19a) gives (20a) $\emptyset \vdash \mathbb{C}' \rhd F$, (21a) $F \vdash s[N']$, and (22a) F sensibly extends $\emptyset$. Now as $(\nu s) \in \mathbb{C}$, $(\nu s) \in \mathbb{C}'$, allowing us to apply Lemma 29 to (20a) giving (23a) $\emptyset \vdash \mathbb{C}' \rhd F', s\ \mathtt{returns}\ bS$. Combining (21a) and (23a), we get (24a) $F', s\ \mathtt{returns}\ bS \vdash s[N']$. Applying inversion and PROC-THREAD to (24a) gives (25a) $F', s\ \mathtt{returns}\ bS \ni s\ \mathtt{returns}\ bS$ and (26a) $F', s\ \mathtt{returns}\ bS \vdash_s N' : bS$. Applying Lemma 27 to (1), (5), and (7) gives either ((23a) and (28a) $F', s\ \mathtt{returns}\ bS$ sensibly extends E) or $\mathbb{C}'$ is a shape trap. Now we have already shown that if $\mathbb{C}'$ is a shape trap, then we are done. So let us consider (23a) and (28a) to hold. Applying Lemma 15 and Lemma 29 to (23a), (28a), and (14) give (29a) $\emptyset \vdash \mathbb{C}' \rhd F', s\ \mathtt{returns}\ aT$, and (30a) $F', s\ \mathtt{returns}\ aT, x{:}bS \vdash_s M : aT$. Applying STAT-METHOD-CONTEXT to (26a) and (30a) gives (31a) $F', s\ \mathtt{returns}\ aT \vdash_s \mathtt{val}\ x =_s \{N'\} M : aT$. Applying PROC-THREAD to $F', s\ \mathtt{returns}\ aT \ni s\ \mathtt{returns}\ aT$ and (31a) gives (32a) $F', s\ \mathtt{returns}\ aT \vdash s[\mathtt{val}\ x =_s \{N'\} M]$. Applying Lemma 17 to (29a) and (32a) gives (33a) $\emptyset \vdash \mathbb{C}'\llbracket s[\mathtt{val}\ x =_s \{N'\} M] \rrbracket$. Using the equality of (4) gives $\emptyset \vdash B$, as desired.

Case R-LOCK-CONTEXT: Similar. The only differences are the use of STAT-LOCK-CONTEXT instead of STAT-METHOD-CONTEXT and the handling of the additional assumption from the judgment, namely that $E'', s\ \mathtt{returns}\ aT \vdash p : \Box\mathtt{Lock}$. Of course, this can be handled in the same way that (14) $E', s\ \mathtt{returns}\ aT, x{:}bS \vdash_s M : aT$ from the previous case is handled.

We now have to prove the proposition is true for the remaining cases. Assume that one of the remaining reduction rules satisfies (2). Let us assume for this case that (3) $A = \mathbb{C}\llbracket A' \rrbracket$ and (4) $B = \mathbb{C}'\llbracket B' \rrbracket$. By inspecting the rule for this case, we know that $\mathbb{C} = \mathbb{C}'$. Thus, we have (5) $B = \mathbb{C}\llbracket B' \rrbracket$. Now there are two cases to consider. Either $\mathbb{C}$ is a shape trap or it isn't. If it is, then by the fact that Lemma 26 holds, $\emptyset \vdash \mathbb{C}\llbracket B' \rrbracket$. Now since (5) $B = \mathbb{C}\llbracket B' \rrbracket$ holds, we have $\emptyset \vdash B$, as desired. Thus, if $\mathbb{C}$ is a shape trap, we are done. Since we arbitrarily chose the reduction rule from the remaining ones, we know that this is the case for all of them. So for each of the remaining cases, we proceed assuming that $\mathbb{C}$ is not a shape trap. We consider each case in turn beginning with the most interesting ones.

Case R-SPECULATION-CLOSE: Assume that R-SPECULATION-CLOSE satisfies (2). Let us assume (3) $A = \mathbb{C}\llbracket \top \Rightarrow A' \,[]\, \langle s?p.f{=}v \rangle \Rightarrow B' \rrbracket$ and (4) $B = \mathbb{C}\llbracket B' \rrbracket$. By the equalities of (3) and (4) combined with (2), we have (5) $\mathbb{C}\llbracket \top \Rightarrow A' \,[]\, \langle s?p.f{=}v \rangle \Rightarrow B' \rrbracket \to \mathbb{C}\llbracket B' \rrbracket$, and (6) $\mathbb{C}, s$ justifies speculation $p.f = v$. By applying Lemma 16 to (1), we know that (7) $\emptyset \vdash \mathbb{C} \rhd E$ and (8) $E \vdash \top \Rightarrow A' \,[]\, \langle s?p.f{=}v \rangle \Rightarrow B'$. By inversion and PROC-SPECULATION on (8), we have (9) $E \vdash A'$, (10) $E \vdash \langle s?p.f{=}v \rangle \rhd E'$, and (11) $E' \vdash B'$. We note that PROC-SPECULATION-SHAPE-ERROR cannot be applied here since (6) holds. Applying Lemma 20 to (6), (10), and (11) gives (12) $E \vdash B'$. Applying Lemma 17 to (12) and (11) gives (13) $\emptyset \vdash \mathbb{C}\llbracket B' \rrbracket$. By the equality of (4), we know (14) $\emptyset \vdash B$, as desired.

Case R-SPECULATION-OPEN: Assume that R-SPECULATION-OPEN satisfies (2). Let us assume (3) $A = \mathbb{C}\llbracket A' \rrbracket$ and (4) $B = \mathbb{C}\llbracket \top \Rightarrow A' \,[]\, \langle s?p.f{=}v \rangle \Rightarrow A' \rrbracket$. By the equalities of (3) and (4) combined with (2), we have (5) $\mathbb{C}\llbracket A' \rrbracket \to \mathbb{C}\llbracket \top \Rightarrow A' \,[]\, \langle s?p.f{=}v \rangle \Rightarrow A' \rrbracket$, (6) $s \in \mathit{thrds}(A')$, (7) $p \in \mathit{objs}(A')$, and (8) $v \in \mathscr{BV} \cup \mathit{objs}(A')$. By applying Lemma 16 to (1), we know that (8) $\emptyset \vdash \mathbb{C} \rhd E$ and (9) $E \vdash A'$. As (6) and (9) hold, we know (10) $s \in \mathit{dom}(E)$. As (7) and (9) hold, it is the case that (11) $E \ni p{:}D$. There are two cases to consider.

Case $E \vdash \langle s?p.f{=}v \rangle \rhd E'$: Assume (12a) $E \vdash \langle s?p.f{=}v \rangle \rhd E'$ for some environment E'. By applying Corollary 8 to (9) with (12a), we get (13a) $E' \vdash A'$. Applying PROC-SPECULATION to (9), (12a), and (13a), we get (14a) $E \vdash \top \Rightarrow A' \,[]\, \langle s?p.f{=}v \rangle \Rightarrow A'$. Applying Lemma 17 to (8) and (14a) gives (15a) $\emptyset \vdash \mathbb{C}\llbracket \top \Rightarrow A' \,[]\, \langle s?p.f{=}v \rangle \Rightarrow A' \rrbracket$. Due the equivalence of (4), (16a) $\emptyset \vdash B$ holds, as desired.

Case $E \nvdash \langle s?p.f{=}v \rangle$: Assume $E \nvdash \langle s?p.f{=}v \rangle$. Then it must be the case that (12b) either ($\mathit{ftype}(D.f) = \ldots bt \ldots$ and $\neg(v$ inhabits $bt)$) or ($\mathit{ftype}(D.f) = \ldots D' \ldots$ and $E \ni v{:}D''$ and $D'' \neq D'$). Applying PROC-SPECULATION-SHAPE-ERROR to (10), (11), (9), and (12b), we have (13b) $E \vdash \top \Rightarrow A' \,[]\, \langle s?p.f{=}v \rangle \Rightarrow A'$. Applying Lemma 17 to (8) and (13b) gives (14b) $\emptyset \vdash \mathbb{C}\llbracket \top \Rightarrow A' \,[]\, \langle s?p.f{=}v \rangle \Rightarrow A' \rrbracket$. Due the equivalence of (4), (15b) $\emptyset \vdash B$ holds, as desired.

As we showed that each case satisfies the proposition, we know that R-SPECULATION-OPEN satisfies the proposition.

Case R-START: Assume that R-START satisfies (2). Let us assume (3) $A = \mathbb{C}\llbracket A' \rrbracket$, (4) $B = \mathbb{C}\llbracket B' \rrbracket$, where $A' = \mathtt{free}\ \ell \mid s[\mathtt{val}\ x = p.\mathit{start}();\ M] \mid \mathtt{runnable}\ p$ and $B' = \langle s!\ell{:}0 \rangle \langle s!\ell{:}1 \rangle (s[M\{\!\![^{\mathtt{unit}}/_x]\!\!\}] \mid \langle p!\ell{:}2 \rangle\, p[N\{\!\![^{p}/_{\mathtt{this}}]\!\!\}])$. By the equalities of (3) and (4) combined with (2), we have (5) $\mathbb{C}\llbracket \mathtt{free}\ \ell \mid s[\mathtt{val}\ x = p.\mathit{start}();\ M] \mid \mathtt{runnable}\ p \rrbracket \to \mathbb{C}\llbracket \langle s!\ell{:}0 \rangle \langle s!\ell{:}1 \rangle (s[M\{\!\![^{\mathtt{unit}}/_x]\!\!\}$
Further, by the assumptions of the rule, we have (6) $\mathbb{C} \ni p{:}D$, (7) $\mathit{mbody}(D.\mathtt{run}) = \lambda\vec{y}.N$, (8) D not reserved, and (9) $\mathbb{C} \ni \ell{:}\mathtt{Lock}$. By applying Lemma 16, inversion and PROC-PAR two times, and inversion and PROC-THREAD on (1) and (3), we have (10) $\emptyset \vdash \mathbb{C} \rhd E$, (11) E is a sensible extension of $\emptyset$, (12) $E \vdash \mathtt{free}\ \ell$, (13) $E \vdash \mathtt{runnable}\ p$, (14) $E \ni s\ \mathtt{returns}\ aT$ and (15) $E \vdash_s \mathtt{val}\ x = p.\mathit{start}();\ M : aT$. By (14), we know (16) $s \in \mathit{dom}(E)$.
We now type the synchronization actions. By applying inversion on PROC-RUNNABLE and PROC-FREE to (13) and (14), we have (17) $E \vdash \diamond$, (18) $p \in \mathit{dom}(E)$, (19) $E \ni \ell{:}\mathtt{Lock}$. Applying ACT-ACQUIRE, ACT-RELEASE, and ACT-ACQUIRE again on (14), (18), and (19), we get (20) $E \vdash \langle s!\ell{:}0 \rangle \rhd F, s{:}\mathtt{lock}\ \ell$, (21) $F, s{:}\mathtt{lock}\ \ell \vdash \langle s!\ell{:}1 \rangle \rhd F'$, and (22) $F' \vdash \langle p!\ell{:}2 \rangle \rhd F''$, where

- $F = E, s\!:\!Z, s\!:\!\mathit{effects}(\ell, E)$,
- $Z = \{\zeta \mid E \ni \mathtt{onacq}\,\Phi\,\zeta \text{ and } E \Vdash_s \Phi\}$,
- $F' = F, \ell\!:\!\mathit{effects}(s, F)$,
- $F'' = F', p\!:\!Z', p\!:\!\mathtt{lock}\,\ell, p\!:\!\mathit{effects}(\ell, F')$,
- $Z' = \{\zeta \mid F' \ni \mathtt{onacq}\,\Phi\,\zeta \text{ and } F' \Vdash_p \Phi\}$.

We now type $p\,\mathtt{[}N\{\!\!\{{}^{p}/_{\mathtt{this}}\}\!\!\}\mathtt{]}$ and $s\,\mathtt{[}M\{\!\!\{{}^{\mathtt{unit}}/_{x}\}\!\!\}\mathtt{]}$. By the definitions of F, F', F'', (6), and (18), $F''' \vdash p : \boxast D$. Noting that the type of $\mathit{mtype}(D.\mathit{run}()) = \boxast; \emptyset \to \square\,\mathtt{Unit}$, we can apply Lemma 24 on (7) followed by PROC-THREAD to get (23) $F'' \vdash p\,\mathtt{[}N\{\!\!\{{}^{p}/_{\mathtt{this}}\}\!\!\}\mathtt{]}$. Starting from (15), applying inversion and STAT-METHOD, Lemma 24, and PROC-THREAD gives (24) $E \vdash s\,\mathtt{[}M\{\!\!\{{}^{\mathtt{unit}}/_{x}\}\!\!\}\mathtt{]}$.
We can combine the processes and actions as follows. Starting with (23), we can apply PROC-ACTION on (22), (21), and (20), followed by PROC-PAR on (24). This gives (25)
$E \vdash s\,\mathtt{[}M\{\!\!\{{}^{\mathtt{unit}}/_{x}\}\!\!\}\mathtt{]} \mid \langle s\,\mathtt{!}\,\ell\!:\!0\rangle\langle s\,\mathtt{!}\,\ell\!:\!1\rangle\langle p\,\mathtt{!}\,\ell\!:\!2\rangle p\,\mathtt{[}N\{\!\!\{{}^{p}/_{\mathtt{this}}\}\!\!\}\mathtt{]}$. Applying Proposition 25 on (25) and the result of applying S-PREFIX-PAR twice on the process in (25) gives (26)
$E \vdash \langle s\,\mathtt{!}\,\ell\!:\!0\rangle\langle s\,\mathtt{!}\,\ell\!:\!1\rangle(s\,\mathtt{[}M\{\!\!\{{}^{\mathtt{unit}}/_{x}\}\!\!\}\mathtt{]} \mid \langle p\,\mathtt{!}\,\ell\!:\!2\rangle p\,\mathtt{[}N\{\!\!\{{}^{p}/_{\mathtt{this}}\}\!\!\}\mathtt{]})$.
Applying Lemma 17 on (10) and (26) gives (27)
$\emptyset \vdash \mathbb{C}\llbracket\langle s\,\mathtt{!}\,\ell\!:\!0\rangle\langle s\,\mathtt{!}\,\ell\!:\!1\rangle(s\,\mathtt{[}M\{\!\!\{{}^{\mathtt{unit}}/_{x}\}\!\!\}\mathtt{]} \mid \langle p\,\mathtt{!}\,\ell\!:\!2\rangle p\,\mathtt{[}N\{\!\!\{{}^{p}/_{\mathtt{this}}\}\!\!\}\mathtt{]})\rrbracket$. By applying the equality of (4) to (27), we have (28) $\emptyset \vdash B$, as desired.

Case R-NEW: Assume that R-NEW satisfies (2). Let us assume (3) $A = \mathbb{C}\llbracket A'\rrbracket$, (4) $B = \mathbb{C}\llbracket B'\rrbracket$, where $A' = \mathtt{free}\,p \mid s\,\mathtt{[val}\,x\,\mathtt{=}\,\mathtt{new}\,D\langle\vec{a}\rangle(\vec{v});\,M\mathtt{]}$ and $B' =$
$\mathtt{runnable}\,p \mid \langle s\,\mathtt{!}\,p\,\mathtt{.}\vec{f}\mathtt{=}\vec{v}\rangle\,\langle s\,\mathtt{!}\,p\,\mathtt{.}\vec{g}\mathtt{=}\vec{\mathtt{null}}\rangle\,\langle s\,\mathtt{!}\,p\rangle\,s\,\mathtt{[}M\{\!\!\{{}^{p}/_{x}\}\!\!\}\mathtt{]}$. By the equalities of (3) and (4) combined with (2), we have (5) $\mathbb{C}\llbracket\mathtt{free}\,p \mid s\,\mathtt{[val}\,x\,\mathtt{=}\,\mathtt{new}\,D\langle\vec{a}\rangle(\vec{v});\,M\mathtt{]}\rrbracket \to$
$\mathbb{C}\llbracket\mathtt{runnable}\ p \mid \langle s\,\mathtt{!}\,p\,\mathtt{.}\vec{f}\mathtt{=}\vec{v}\rangle\,\langle s\,\mathtt{!}\,p\,\mathtt{.}\vec{g}\mathtt{=}\vec{\mathtt{null}}\rangle\,\langle s\,\mathtt{!}\,p\rangle\,s\,\mathtt{[}M\{\!\!\{{}^{p}/_{x}\}\!\!\}\mathtt{]}\rrbracket$. Further, by the assumptions of the rule, we have (6) $\mathbb{C} \ni p\!:\!D$, (7) $\mathit{finals}(D) = \vec{f}$, (8) $\mathit{nonfinals}(D) = \vec{g}$, and (9) D not reserved. By applying Lemma 16, inversion and PROC-PAR, and inversion and PROC-THREAD on (1) and (3), we have (7) $\emptyset \vdash \mathbb{C} \triangleright E$, (8) E is a sensible extension of $\emptyset$, (9) $E \vdash \mathtt{free}\ p$, (10) $E \ni s\ \mathtt{returns}\ aT$ and (11) $E \vdash_s \mathtt{val}\ x\ \mathtt{=}\ \mathtt{new}\ D\langle\vec{a}\rangle(\vec{v});\ M : aT$. Applying STAT-NEW to (11) gives (12) $\mathit{ftype}(D.\vec{f}) = \mathtt{final}\ \vec{b}\vec{S}$, (13) $E \vdash_s \vec{v} : \vec{b}\vec{S}$, and (14) $E, x\!:\boxast D \vdash_s M : aT$. Applying inversion and PROC-FREE on (9) and by (10), we have $E \vdash \diamond$, (15) $p \in \mathit{dom}(E)$ and (16) $s \in \mathit{dom}(E)$. We can get the write actions on the final fields by using either ACT-$\square$-RACING-FINAL-WRITE or ACT-$\boxast$-FINAL-WRITE on (16), (15) with (6), (12), and (13), giving (17) $E \vdash \langle s\,\mathtt{!}\,p\,\mathtt{.}\vec{f}\mathtt{=}\vec{v}\rangle \triangleright E$. In a similar way, we can get the non-final write actions by using one of ACT-$\square$-GUARDED-WRITE or ACT-$\boxast$-GUARDED-WRITE or ACT-$\square$-RACING-FINAL-WRITE, giving (18) $E \vdash \langle s\,\mathtt{!}\,p\,\mathtt{.}\vec{g}\mathtt{=}\vec{\mathtt{null}}\rangle \triangleright E$. Applying ACT-BEGIN to (15) and (16) gives (19) $E \vdash \langle s\,\mathtt{!}\,p\rangle \triangleright E, s\!:\wr p \int$. Given (19), with VAL-OBJECT-$\boxast$, we can show that (20) $E \vdash p : \boxast D$, which in turn gives (21) $E, s\!:\wr p\int \vdash p : \boxast D$ and (22) $E, s\!:\wr p\int \vdash_s p : \boxast D$. Applying Lemma 24 followed by PROC-THREAD gives (23) $E, s\!:\wr p\int \vdash s\,\mathtt{[}M\{\!\!\{{}^{p}/_{x}\}\!\!\}\mathtt{]}$. Applying PROC-ACTION to (23), (19), (18), and (17) followed by PROC-PAR on the result of PROC-RUNNABLE applied to (15) gives (24)
$E \vdash \mathtt{runnable}\,p \mid \langle s\,\mathtt{!}\,p\,\mathtt{.}\vec{f}\mathtt{=}\vec{v}\rangle\langle s\,\mathtt{!}\,p\,\mathtt{.}\vec{g}\mathtt{=}\vec{\mathtt{null}}\rangle\langle s\,\mathtt{!}\,p\rangle s\,\mathtt{[}M\{\!\!\{{}^{p}/_{x}\}\!\!\}\mathtt{]}$. Applying Lemma 17 on (7) and (24) gives (25)
$\emptyset \vdash \mathbb{C}\llbracket\mathtt{runnable}\ p \mid \langle s\,\mathtt{!}\,p\,\mathtt{.}\vec{f}\mathtt{=}\vec{v}\rangle\langle s\,\mathtt{!}\,p\,\mathtt{.}\vec{g}\mathtt{=}\vec{\mathtt{null}}\rangle\langle s\,\mathtt{!}\,p\rangle s\,\mathtt{[}M\{\!\!\{{}^{p}/_{x}\}\!\!\}\mathtt{]}\rrbracket$. By applying the equality of (4) to (25), we have (26) $\emptyset \vdash B$, as desired.

Case R-FIELD-WRITE: Assume that R-FIELD-WRITE satisfies (2). This gives us (3) $A = \mathbb{C}\llbracket s\,\mathtt{[}p.f\ \mathtt{=}\ v;\ M\mathtt{]}\rrbracket$, (4) $B = \mathbb{C}\llbracket\langle s\,\mathtt{!}\,p\,\mathtt{.}f\mathtt{=}v\rangle\,s\,\mathtt{[}M\mathtt{]}\rrbracket$. By the equalities of (3) and (4) combined with (2), we have (5) $\mathbb{C}\llbracket s\,\mathtt{[}p.f\ \mathtt{=}\ v;\ M\mathtt{]}\rrbracket \to \mathbb{C}\llbracket\langle s\,\mathtt{!}\,p\,\mathtt{.}f\mathtt{=}v\rangle\,s\,\mathtt{[}M\mathtt{]}\rrbracket$. By applying Lemma 16, and inversion and PROC-THREAD on (1) and (3), we have (6) $\emptyset \vdash \mathbb{C} \triangleright E$, E is a sensible extension of $\emptyset$, (7) $E \ni s\,\mathtt{returns}\,aT$ and (8) $E \vdash_s s\,\mathtt{[}p.f\ \mathtt{=}\ v;\ M\mathtt{]} : aT$. There are two possibilities of judgments that can type (8).

Case STAT-GUARDED-WRITE: Assume that STAT-GUARDED-WRITE types (8). By inversion and STAT-GUARDED-WRITE, we know (9a) $E \vdash_s p : \square D$, (10a) $\mathit{ftype}(D.f) = bS\ \mathtt{wrguard}\ \Psi$, (11a) $E \Vdash_s \Psi\{\!\!\{{}^{v}/_{\mathtt{this}}\}\!\!\}$, (12a) $E \vdash_s v : bS$ and (13a) $E \vdash_s M : aT$. If $b = \square$, use ACT-$\square$-RACING-GUARDED-WRITE to type (14a) $E \vdash \langle s\,\mathtt{!}\,p\,\mathtt{.}f\mathtt{=}v\rangle \triangleright E$. Otherwise, use ACT-$\boxast$-GUARDED-WRITE to type (14a) $E \vdash \langle s\,\mathtt{!}\,p\,\mathtt{.}f\mathtt{=}v\rangle \triangleright E$. Applying PROC-THREAD to (7) and (13a) followed by PROC-ACT on (14a) gives (15a) $E \vdash \langle s\,\mathtt{!}\,p\,\mathtt{.}f\mathtt{=}v\rangle\,s\,\mathtt{[}M\mathtt{]}$. By applying Lemma 17 on (6) and (15a) and using the equality of (4), we have (16a) $\emptyset \vdash B$, as desired.

Case STAT-RACING-WRITE: Assume that STAT-RACING-WRITE types (8). By inversion and STAT-RACING-WRITE, we know (9a) $E \vdash_s p : \square D$, (10a) $\mathit{ftype}(D.f) = S$, (11a) $E \vdash_s v : \square S$, and (12a) $E \vdash_s M : aT$. Use ACT-$\square$-RACING-FINAL-WRITE to type (13a) $E \vdash \langle s\,\mathtt{!}\,p\,\mathtt{.}f\mathtt{=}v\rangle \triangleright E$. Applying PROC-THREAD to (7) and (12a) followed by PROC-ACT on (13a) gives (14a) $E \vdash \langle s\,\mathtt{!}\,p\,\mathtt{.}f\mathtt{=}v\rangle\,s\,\mathtt{[}M\mathtt{]}$. By applying Lemma 17 on (6) and (15a) and using the equality of (4), we have (16a) $\emptyset \vdash B$, as desired.

As we proved the subcases for both possible judgments to type (8), we have proved the proposition for the case of R-FIELD-WRITE satisfying (2).

Case R-ATOMIC-NEW: This case is similar to both R-NEW and R-START and can be proven similarly.
First, apply Lemma 16, inversion and PROC-PAR twice, inversion PROC-THREAD to (3) $E \vdash s\,\mathtt{[val}\ x\ \mathtt{=}\ \mathtt{new}\ \mathtt{Atomic}(v);\ M\mathtt{]}$. Then apply inversion and STAT-NEW to (4) $E \vdash_s \mathtt{val}\ x\ \mathtt{=}\ \mathtt{new}\ \mathtt{Atomic}(v);\ M : aT$. Applying inversion on PROC-FREE and PROC-ALLOCATED to (5) $E \vdash \mathtt{free}\ \ell$ and (6) $E \vdash \ell\mathtt{:Atomic}$ along with the typing of v from the premises of inversion and STAT-NEW applied to (4) gives us (5) $E \vdash \ell\mathtt{:Atomic\{}v\mathtt{;}2\mathtt{\}}$. Similar reasoning for how to add the begin p in R-NEW and how to add the locks in and type the thread in R-START will allow us to get (6) $E \vdash \langle s\,\mathtt{!}\,\ell\rangle\,\langle s\,\mathtt{!}\,\ell\!:\!1\rangle\,s\,\mathtt{[}M\{\!\!\{{}^{\ell}/_{x}\}\!\!\}\mathtt{]}$. Combining (5) and (6) with PROC-PAR and then applying Lemma 17 allows us to have the desired result of $\emptyset \vdash \mathbb{C}\llbracket\ell\mathtt{:Atomic\{}v\mathtt{;}2\mathtt{\}} \mid \langle s\,\mathtt{!}\,\ell\rangle\,\langle s\,\mathtt{!}\,\ell\!:\!1\rangle\,s\,\mathtt{[}M\{\!\!\{{}^{\ell}/_{x}\}\!\!\}\mathtt{]}\rrbracket$.

Case R-ATOMIC-SET: Similar to R-METHOD-CALL and R-ATOMIC-NEW.

Case R-ATOMIC-GET: Similar to R-METHOD-CALL and R-ATOMIC-NEW.

Case R-ATOMIC-GETANDSET: Similar to R-METHOD-CALL and R-ATOMIC-NEW.

Case R-ATOMIC-COMPAREANDSET-FALSE: Similar to R-METHOD-CALL and R-ATOMIC-NEW.

Case R-ATOMIC-COMPAREANDSET-TRUE: Similar to R-METHOD-CALL and R-ATOMIC-NEW.

Case R-LOCK-NEW: Similar to R-ATOMIC-NEW and R-NEW.

Case R-LOCK-ACQUIRE: Similar to R-METHOD-CALL and R-START, using STAT-LOCK and STAT-LOCK-CONTEXT at the appropriate time.

Case R-LOCK-RELEASE: Similar to R-METHOD-RETURN and R-LOCK-ACQUIRE.

Case R-LOCK-ACQUIRE-REENTRANT: Similar to R-LOCK-ACQUIRE.

Case R-LOCK-RELEASE-REENTRANT: Similar to R-METHOD-RETURN and R-LOCK-RELEASE.

Case R-CONDITION-NEW: Similar to R-ATOMIC-NEW and R-NEW.

Case R-CONDITION-RELEASE: Similar to R-START.

Case R-CONDITION-NOTIFY: Similar to R-METHOD-CALL and R-LOCK-RELEASE.

Case R-CONDITION-ACQUIRE: Similar to R-METHOD-CALL and R-LOCK-ACQUIRE.

Case R-END: Assume that R-END satisfies (2). Let us assume (3) $A = \mathbb{C}\llbracket A'\rrbracket$, (4) $B = \mathbb{C}\llbracket B'\rrbracket$, where $A' = s\,\mathtt{[end}\ p;\ M\mathtt{]}$ and $B' = s\,\mathtt{[}M\mathtt{]}$. By the equalities of (3) and (4) combined with (2), we have (5) $\mathbb{C}\llbracket s\,\mathtt{[end}\ p;\ M\mathtt{]}\rrbracket \to \mathbb{C}\llbracket s\,\mathtt{[}M\mathtt{]}\rrbracket$, and (6) $\langle t\,\mathtt{!}\,p\rangle <^{\mathbb{C},s}_{\exists hb} s\llbracket-\rrbracket$. By applying Lemma 16 on (1) and (3), we have (7) $\emptyset \vdash \mathbb{C} \triangleright E$, (8) $E \vdash s\,\mathtt{[end}\ p;\ M\mathtt{]}$, and (9) E is a sensible extension of $\emptyset$. By applying inversion and PROC-THREAD on (8), we get (10) $E \ni s\ \mathtt{returns}\ aT$ and (11) $E \vdash_s \mathtt{end}\,p;\,M : aT$. Applying STAT-END to (11) gives (12) $E \vdash_s V : \boxast D$ and (13) $E \vdash_s M : aT$. Applying PROC-THREAD to (10) and (13) gives (14) $E \vdash s\,\mathtt{[}M\mathtt{]}$. Applying Lemma 17 to (7) and (14) gives (15) $\emptyset \vdash \mathbb{C}\llbracket s\,\mathtt{[}M\mathtt{]}\rrbracket$. By the equality of (4) with (15), we have (16) $\emptyset \vdash B$, as desired.

Case R-IF-TRUE: Assume that R-IF-TRUE satisfies (2). Let us assume (3) $A = \mathbb{C}\llbracket A'\rrbracket$, (4) $B = \mathbb{C}\llbracket B'\rrbracket$, where $A' = s\,\mathtt{[if\ (true)\,\{}M\mathtt{\}\,else\,\{}N\mathtt{\}]}$ and $B' = s\,\mathtt{[}M\mathtt{]}$. By the equalities of (3) and (4) combined with (2), we have (5) $\mathbb{C}\llbracket s\,\mathtt{[if\ (true)\,\{}M\mathtt{\}\,else\,\{}N\mathtt{\}]}\rrbracket \to \mathbb{C}\llbracket s\,\mathtt{[}M\mathtt{]}\rrbracket$, and (6) $\langle t\,\mathtt{!}\,p\rangle <^{\mathbb{C},s}_{\exists hb} s\llbracket-\rrbracket$. By applying Lemma 16 on (1) and (3), we have (7) $\emptyset \vdash \mathbb{C} \triangleright E$, (8) $E \vdash s\,\mathtt{[if\ (true)\,\{}M\mathtt{\}\,else\,\{}N\mathtt{\}]}$, and (9) E is a sensible extension of $\emptyset$. By applying inversion and PROC-THREAD on (8), we get (10) $E \ni s\,\mathtt{returns}\,aT$ and (11) $E \vdash_s \mathtt{if\ (true)\,\{}M\mathtt{\}\,else\,\{}N\mathtt{\}} : aT$. Applying STAT-IF to (11) gives (12) $E \vdash_s \mathtt{true} : \square\,\mathtt{boolean}$, (13) $E \vdash_s M : aT$, and (14) $E \vdash_s N : aT$. Applying PROC-THREAD to (10) and (13) gives (15) $E \vdash s\,\mathtt{[}M\mathtt{]}$. Applying Lemma 17 to (7) and (15) gives

(16) $\emptyset \vdash \mathbb{C}\llbracket s\texttt{[}M\texttt{]}\rrbracket$. By the equality of (4) with (16), we have (17) $\emptyset \vdash B$, as desired.

Case R-IF-FALSE: Similar.

Case R-CAST: Assume that R-CAST satisfies (2). Let us assume (3) $A = \mathbb{C}\llbracket A'\rrbracket$, (4) $B = \mathbb{C}\llbracket B'\rrbracket$, where $A' = s\texttt{[val}\ x\ \texttt{=}\ (D)p\texttt{;}\ M\texttt{]}$ and $B' = s\texttt{[}M\{\!\{{}^{p}/_{x}\}\!\}\texttt{]}$. By the equalities of (3) and (4) combined with (2), we have (5) $\mathbb{C}\llbracket s\texttt{[val}\ x\ \texttt{=}\ (D)p\texttt{;}\ M\texttt{]}\rrbracket \rightarrow \mathbb{C}\llbracket s\texttt{[}M\{\!\{{}^{p}/_{x}\}\!\}\texttt{]}\rrbracket$, and (6) $\mathbb{C} \ni p\texttt{:}D$. By applying Lemma 16, and inversion and PROC-THREAD on (1) and (3), we have (7) $\emptyset \vdash \mathbb{C} \triangleright E$, E is a sensible extension of $\emptyset$, (8) $E \ni s\ \texttt{returns}\ aT$, and (9) $E \vdash_s \texttt{val}\ x\ \texttt{=}\ (D)p\texttt{;}\ M : aT$. Applying inversion and STAT-CAST gives (10) $E \vdash_s p : b\texttt{Object}$, and (11) $E, x{:}bD \vdash_s M : aT$. By inversion and VAL-SUB-TYPE, (6), and (10) gives (12) $E \vdash_s p : bD$. Applying Lemma 24 on (11) and (12) followed by PROC-THREAD gives (13) $E \vdash_s M\{\!\{{}^{p}/_{x}\}\!\} : aT$ and (14) $E \vdash s\texttt{[}M\{\!\{{}^{p}/_{x}\}\!\}\texttt{]}$. By applying Lemma 17 on (7) and (14) and using the equality of (4), we have (15) $\emptyset \vdash B$, as desired.

Case R-METHOD-CALL: Assume that R-METHOD-CALL satisfies (2). Let us assume (3) $A = \mathbb{C}\llbracket s\texttt{[val}\ x\ \texttt{=}\ p.m(\vec{v})\texttt{;}\ M\texttt{]}\rrbracket$, (4) $B = \mathbb{C}\llbracket s\texttt{[val}\ x\ \texttt{=}\ {}_s\texttt{\{}N\{\!\{{}^{p}/_{\texttt{this}}\}\!\}\{\!\{{}^{\vec{v}}/_{\vec{y}}\}\!\}\texttt{\}}M\texttt{]}\rrbracket$. By the equalities of (3) and (4) combined with (2), we have (5) $\mathbb{C}\llbracket s\texttt{[val}\ x\ \texttt{=}\ p.m(\vec{v})\texttt{;}\ M\texttt{]}\rrbracket \rightarrow \mathbb{C}\llbracket s\texttt{[val}\ x\ \texttt{=}\ {}_s\texttt{\{}N\{\!\{{}^{p}/_{\texttt{this}}\}\!\}\{\!\{{}^{\vec{v}}/_{\vec{y}}\}\!\}\texttt{\}}M\texttt{]}\rrbracket$. By the assumptions for the rule, we have (6) $\mathbb{C} \ni p\texttt{:}D$, (7) $mbody(D.m) = \lambda\vec{y}.N$, (8) D not reserved, and (9) $m \neq \texttt{start}$. By applying Lemma 16, and inversion and PROC-THREAD on (1) and (3), we have (10) $\emptyset \vdash \mathbb{C} \triangleright E$, E is a sensible extension of $\emptyset$, (11) $E \ni s\ \texttt{returns}\ aT$ and (12) $E \vdash_s s\texttt{[val}\ x\ \texttt{=}\ p.m(\vec{v})\texttt{;}\ M\texttt{]} : aT$. Applying inversion and STAT-METHOD and then Lemma 24 on (12) gives (13) $E, x{:}bS \vdash_s M : aT$, and (14) $E \vdash_s N\{\!\{{}^{p}/_{\texttt{this}}\}\!\}\{\!\{{}^{\vec{v}}/_{\vec{y}}\}\!\} : bS$. Reordering the environments in (13) and (14), applying STAT-METHOD-CONTEXT and PROC-THREAD gives (15) $E \vdash s\texttt{[val}\ x\ \texttt{=}\ {}_s\texttt{\{}N\{\!\{{}^{p}/_{\texttt{this}}\}\!\}\{\!\{{}^{\vec{v}}/_{\vec{y}}\}\!\}\texttt{\}}M\texttt{]}$. By applying Lemma 17 on (10) and (15) and using the equality of (4), we have (16) $\emptyset \vdash B$, as desired.

Case R-METHOD-RETURN: Assume that R-METHOD-RETURN satisfies (2). Then we have $\mathbb{C}\llbracket s\texttt{[val}\ x\ \texttt{=}\ {}_s\texttt{\{}\uparrow v\texttt{\}}M\texttt{]}\rrbracket \rightarrow \mathbb{C}\llbracket s\texttt{[}M\{\!\{{}^{v}/_{x}\}\!\}\texttt{]}\rrbracket$. This case can be handled similarly to the other cases. Apply Lemma 16, inversion and PROC-THREAD, inversion and STAT-METHOD-CONTEXT, inversion and STAT-RETURN, Lemma 24, PROC-THREAD, and then Lemma 17 to get the desired result.

Case R-OPERATOR: Assume that R-OPERATOR satisfies (2). Then we have
$\mathbb{C}\llbracket s\texttt{[val}\ x\ \texttt{=}\ op(\vec{v})\texttt{;}\ M\texttt{]}\rrbracket \rightarrow \mathbb{C}\llbracket s\texttt{[}M\{\!\{{}^{w}/_{x}\}\!\}\texttt{]}\rrbracket$. This case can be handled similarly to R-METHOD-CALL. Apply Lemma 16, inversion and PROC-THREAD, inversion and STAT-OPERATOR, Lemma 24, PROC-THREAD, and then Lemma 17 to get the desired result.

Case R-FIELD-READ: Assume that R-FIELD-READ satisfies (2). Then we have $\mathbb{C}\llbracket s\texttt{[val}\ x\ \texttt{=}\ p.f\texttt{;}\ M\texttt{]}\rrbracket \rightarrow \mathbb{C}\llbracket s\texttt{[}M\{\!\{{}^{v}/_{x}\}\!\}\texttt{]}\rrbracket$. This case can be handled similarly to the other cases. The only twist is that there are three cases to consider. Applying Lemma 16, inversion and PROC-THREAD gives $E \ni s\ \texttt{returns}\ aT$ and (3) $E \vdash_s \texttt{val}\ x\ \texttt{=}\ p.f\texttt{;}\ M : aT$. There are three typing judgments which could type (3), namely STAT-RACING-READ, STAT-FINAL-READ, and STAT-GUARDED-READ. Each case is handled in the same way. Apply Lemma 19 followed by Lemma 24, PROC-THREAD, and Lemma 17 to complete each case.

Since we showed that all reduction rules satisfy the proposition, we know that subject reduction holds. □