Types for Precise Thread Interference

University of California at Santa Cruz Technical Report UCSC-SOE-11-22

Jaheon Yi Tim Disney
Computer Science Department
University of California, Santa Cruz
Santa Cruz, CA 95064

Stephen N. Freund
Computer Science Department
Williams College
Williamstown, MA 01267

Cormac Flanagan
Computer Science Department
University of California, Santa Cruz
Santa Cruz, CA 95064

Abstract

The potential for unexpected interference between threads makes multithreaded programming notoriously difficult. Programmers use a variety of synchronization idioms such as locks and barriers to restrict where interference may actually occur. Unfortunately, the resulting *actual* interference points are typically never documented and must be manually reconstructed as the first step in any subsequent programming task (code review, refactoring, etc).

This paper proposes explicitly documenting actual interference points in the program source code, and it presents a type and effect system for verifying the correctness of these interference specifications.

Experimental results on a variety of Java benchmarks show that this approach provides a significant improvement over prior systems based on method-level atomicity specifications. In particular, it reduces the number of interference points one must consider from several hundred points per thousand lines of code to roughly 13 per thousand lines of code. Explicit interference points also serve to highlight all known concurrency defects in these benchmarks.

1. Introduction

The widespread adoption of multi-core processors necessitates effective techniques for developing reliable multithreaded software. However, developing and validating multithreaded software is very difficult, in large part because of the potential for nondeterministic interference between concurrent threads. Typically, programmers use a variety of synchronization idioms to restrict where interference may occur. For example, locks, semaphores, or barriers can be used to prevent interference between threads at program points within critical sections.

There is, however, a large gap between *actual interference points* (where interference actually happens in the given program due to its synchronization structure) and *preemptive interference points* (which potentially can occur at any program point under a typical preemptive semantics for thread interleaving). Knowledge of actual interference points is required for almost all reasoning about program behavior, but these interference points are almost never documented in the program source code. Thus every programming task (code review, refactoring, feature extension, etc) must typically begin with the programmer manually reconstructing the actual interference points by analyzing the synchronization structure of the target program.

Manual reconstruction of actual interference points is tedious and error-prone. Moreover, actual interference points are quite sparse in practice, and consequently programmers have a tendency to optimistically assume that the code being analyzed is free of interference and simply perform sequential reasoning about program behavior. This shortcut is sometimes correct, but often results in defects such as race conditions and violations of intended atomicity, determinism, and ordering constraints.

Figure 1: Traveling Salesperson Algorithm

```
1  Object lock;
2  volatile int shortestPathLength;
3
4  compound void searchFrom(Path path) {
5    if (path.length >= ..shortestPathLength)
6      return;
7
8    if (path.isComplete()) {
9      ..synchronized (lock) {
10       if (path.length < shortestPathLength)
11         shortestPathLength = path.length;
12     }
13   } else {
14     for (Path c: path.children())
15       searchFrom#(c);
16   }
17 }
```

The central philosophy behind this paper is that actual interference points should be explicit in the program source code, avoiding the need to reconstruct them before each programming task.

To illustrate the benefits of explicit interference annotations, consider the traveling salesperson algorithm shown in Figure 1. The function `searchFrom` recursively searches through all extensions of a salesperson's `path`, aborting the search whenever the `path`'s length field becomes greater than the length of the shortest complete path found so far (stored in `shortestPathLength`). To exploit multicore processors, multiple instances of `searchFrom` execute concurrently, and the code uses the notation "`..`" to document thread interference in a lightweight manner. For example, the variable `shortestPathLength` is protected by the mutex `lock` for all writes, but racy reads are permitted to maximize performance. Thus, thread interference may occur before the read of `shortestPathLength` on line 5 (because of potential concurrent writes) and before the lock acquire at line 9 (because a concurrent invocation of `searchFrom` may be racing on that lock). We refer to these explicitly annotated points of potential interference as *yield points*, and they document where interleaved actions of concurrent threads may conflict with operations performed by this function.

To facilitate modular reasoning, methods are annotated to indicate their yielding effects. Methods with no internal yield points are declared `atomic`, and methods that may yield are declared `compound`, as illustrated by the declaration of `searchFrom`. Calls to `compound` methods (as on line 15 above) are highlighted with a postfix "#" to indicate that the callee contains yield points, and so invariants over shared state that hold before the call may not hold after the call returns.

This paper presents a type and effect system to verify that explicit interference annotations in a program capture all possible thread interference. The type system is based on Lipton's theory of reduction [34], which characterizes when a sequence of instructions from one thread forms a serializable transaction. Our type system guarantees that the instructions of each well-typed thread consists of a sequence of serializable transactions separated by interference annotations. Consequently, any well-typed program behaves *as if* executing under a *cooperative scheduler*, where context switches happen only at explicitly marked interference points.

This approach provides several benefits. First, manual reconstruction of actual interference points is no longer a required first step of any programming task, since the type system guarantees that all interference points are explicitly documented in the source code. Second, sequential reasoning is applicable to any code fragment that does not include interference annotations. Third, interference annotations provide an explicit reminder to programmers about where their natural tendency to apply sequential reasoning is not applicable, and where they need to account for thread interference. Finally, a preliminary user study has shown that documenting interference points produces a statistically significant improvement in the ability of programmers to identify defects [44].

We describe a prototype implementation called JCC (Java cooperability checker) for the Java programming language. This type checker takes as input a program annotated with interference annotations (including where races may occur) and verifies that the specification holds. Experimental results show that JCC requires annotating only 13 interference points per thousand lines of code. In comparison, our prior type system for method-level atomicity [21] requires the programmer to reason about over 180 interference points per thousand lines of code, mainly because of interference points in methods that cannot be verified as atomic.

Contributions. In summary, the primary contributions of our work include:

- A lightweight and precise notation for specifying interference points.
- A type and effect system for verifying these interference specifications (Section 5).
- An implementation of this type system for the Java programming language (Section 6).
- Experimental results showing that (1) these interference specifications highlight all known concurrency bugs in our benchmarks, and (2) in comparison to prior approaches based on race conditions or atomicity, the type system reduces the number of interference points by an order of magnitude (Section 7).

Previous work explored dynamic cooperability analyses [52], a cooperability type system for a limited imperative calculus [51], and whether cooperability annotations facilitate program understanding [44]. In this paper we extend cooperability to the type system for a large, object-oriented language, implement a type checker for it, and demonstrate that it is effective on a variety of realistic programs.

Comparison to Atomic Non-Interference Specifications. A method or code block is *atomic* if it does not contain any thread interference points [30, 36, 35, 15], and previous studies have shown that as many as 90% of methods are typically atomic [18].

Unfortunately, the notion of atomicity is rather awkward for specifying interference in non-atomic methods such as `searchFrom`. Some particular code blocks in `searchFrom`, such as the synchronized block, can be marked atomic, as shown in Figure 2, but the result is rather verbose and inadequate, since atomic focuses on delimiting blocks where interference does not occur, but still suggests that interference can occur everywhere outside these atomic code blocks, such as on the access to `path` on line 18.

Figure 2: Traveling Salesperson Algorithm with atomic blocks

```
1  Object lock;
2  volatile int shortestPathLength;
3
4  compound void searchFrom(Path path) {
5    atomic {
6      if (path.length >= shortestPathLength)
7        return;
8    }
9
10   if (path.isComplete()) {
11     atomic {
12       synchronized (lock) {
13         if (path.length < shortestPathLength)
14           shortestPathLength = path.length;
15       }
16     }
17   } else {
18     for (Path c: path.children())
19       searchFrom#(c);
20   }
21 }
```

Moreover, the use of atomic blocks requires a *bimodal* reasoning style that combines sequential reasoning inside atomic blocks with preemptive reasoning (pervasive interference) outside atomic blocks. The programmer is responsible for choosing the appropriate reasoning mode for each piece of code according to whether it is inside or outside an atomic block, with obvious room for error.

Comparison to Yield Interference Specifications. An alternative notation for thread interference is the `yield` statements of cooperative multithreading [3, 4, 9], automatic mutual exclusion [31], and some of our earlier work [51, 52]. After annotating a variety of systems with `yield` statements, we concluded that `yield` clarifies *where* interference may occur, but does not address *why* interference may occur. To illustrate this limitation, consider the following revised `searchFrom` implementation that uses `yield` interference specifications:

```
compound void searchFrom(Path path) {
  yield;
  if (path.length >= shortestPathLength)
  ...
}
```

Here, the `yield` suggests that one of the subsequent reads of `path.length` or `shortestPathLength` is interfering or racy, but it is not immediately obvious which one. By comparison, the interference annotation at line 5 in Figure 1 precisely documents that interference occurs on the read of `shortestPathLength`.

2. Documenting Thread Interference

In a well-typed program, each thread should consist of a sequence of serializable transactions that are separated by yields. A programmer must therefore include a yield point right before the first operation of each transaction, using the following annotations.

Yielding Field Accesses. The following syntax denotes a yielding read or write of a racy field `f`, where the yield is performed just before the access to `f`:

```
e..f         // yield before racy read
e..f = e′    // yield before racy write
```

Yielding Lock Acquires. The following yielding synchronized block performs a yield after the evaluation of the lock expression e, but before the lock acquire. Note that a lock release cannot interfere with any concurrently executing action by another thread and thus never needs to start a new transaction.

```
..synchronized (e) { // yield before acquire
   e′
}
```

Yielding Method Calls. Next, consider an atomic method `m()` whose body performs a complete transaction. In order to sequentially compose two calls to this method, a yield point must occur between in between them. For this purpose, we also allow yield annotations on a method call, where the yield occurs after e and e' have been evaluated and right before the call is performed:

```
e..m(e′)    // yield before method call
```

Non-Atomic Method Calls. A method `m()` is non-atomic or `compound` if its body consists of multiple transactions separated by yield points. In that case, we require calls to `m()` to document the potential for yield points inside `m()`, where the postfix annotation "`#`" reminds the programmer that invariants over thread-shared state that hold before the call may not hold after the call, due to the nested yields.

```
e.m#(e′)    // yields inside m
```

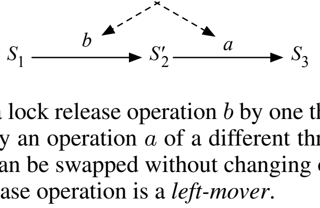
Self-References. As usual, if the target object of a field access or a method call is the self-reference variable `this`, then it can be omitted as follows:

```
..f         // yield before racy read on this
..f = e′    // yield before racy write to this
..m(e′)     // yield before call on this method
```

3. A Review of Lipton's Theory of Reduction

Our type and effect system reasons about thread interference using Lipton's theory of reduction [34], which describes how certain adjacent operations by different threads in a program trace can be swapped without changing the overall behavior of the trace, and when a sequence of instructions from one thread forms a serializable transaction.

Suppose a trace contains an acquire operation a on a lock `m` that is immediately followed by an operation b of a different thread. That operation b cannot either acquire or release `m` (as it is held by the first thread) and so the two operations *commute* — the operations a and b can be swapped without changing the overall behavior or final state of the trace. Thus, we say that acquire operations are *right-movers*.



Similarly, if a lock release operation b by one thread is immediately preceded by an operation a of a different thread, again these two operations can be swapped without changing overall behavior, so each lock release operation is a *left-mover*.

Next, consider an access to a variable `x`. If `x` is race-free, then there are no concurrent accesses to `x` by other threads, so each access commutes with both preceding and following operations of other threads. Put differently, race-free accesses are *both-movers*, in that they are both left and right movers.

Conversely, if `x` is a racy variable, then an access to `x` cannot in general be swapped with a following (or preceding) operation b, since b in general could be a conflicting access to `x`. Thus, we say that racy accesses are *non-movers*, since they are neither left nor right movers. (To avoid the complexities of Java's relaxed memory model [37], we assume here that all racy variables are `volatile`, but the JCC implementation supports non-volatile racy variables: see Section 6.)

This classification of operations as various kinds of movers then allows us to identify serializable code blocks. In particular, suppose the sequence $\alpha = a_1. \cdots .a_n$ of instructions performed by a particular thread consists of:

1. zero or more right-movers, followed by
2. at most one non-mover, followed by
3. zero or more left-movers.

Any instructions of other threads that are interleaved into α can be commuted out so that α executes serially, without interleaved operations of other threads. In this case, we consider α a *serializable transaction*, or simply a *transaction*.

4. Effects for Cooperability

Our effect system characterize the behavior of each program subexpression using two kinds of effects: *mover effects* and *atomicity effects*.

4.1 Mover Effects

A *mover effect* μ characterizes the behavior of a program expression in terms of how operations of that expression commute with operations of other threads:

$$\mu \;::=\; \mathrm{F} \mid \mathrm{Y} \mid \mathrm{M} \mid \mathrm{R} \mid \mathrm{L} \mid \mathrm{N}$$

F: The mover effect F (for functional) describes expressions whose result does not depend on any mutable state. Hence, re-evaluating a functional expression is guaranteed to produce the same result. (Our type system requires all lock names to be functional to ensure that it does not confuse distinct locks.)

Y: The mover effect Y describes yield operations, denoted as "`..`". These expressions mark transactional boundaries where the current transaction ends and a new one starts.

M: The mover effect M describes both-mover expressions that commute both left and right with concurrent operations by other threads, according to Lipton's theory.

R: The mover effect R describes right-mover expressions.

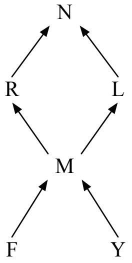
L: The mover effect L describes left-mover expressions.

N: The mover effect N describes non-mover code that may perform a racy access or that may contain right-movers followed by left-movers.

The following tables define the iterative closure (μ^*) and sequential composition ($\mu_1; \mu_2$) of mover effects. These operations are partial (indicated with a "$-$") and may fail if the code between two successive yields would not be serializable. For example, the sequential composition (L; R) is undefined, since their composition is not reducible—code containing a left-mover followed by a right-mover does not form a serializable transaction. Mover effects are ordered by the relation $\sqsubset$ shown via the lattice below.

μ	μ^*
F	F
Y	M
M	M
R	R
L	L
N	—

;	F	Y	M	R	L	N
F	F	Y	M	R	L	N
Y	Y	Y	Y	Y	L	L
M	M	Y	M	R	L	N
R	R	R	R	R	N	N
L	L	Y	L	—	L	—
N	N	R	N	—	N	—



4.2 Atomicity Effects

Each program expression also has an *atomicity effect* τ that summarizes whether the expression performs a yield operation.

$$\tau \quad ::= \quad \text{A} \mid \text{C}$$

Here, A (atomic) means the expression never yields, and C (compound) means the expression may yield. Ordering ($\sqsubset$), iterative closure (τ^*) and sequential composition ($\tau_1;\tau_2$) for atomicity effects are defined by:

$$\begin{array}{rcl} \text{A} & \sqsubset & \text{C} \\ \tau^* & \stackrel{\text{def}}{=} & \tau \\ \tau_1;\tau_2 & \stackrel{\text{def}}{=} & \tau_1 \sqcup \tau_2 \end{array}$$

4.3 Combined Effects

A *combined effect* κ is a pair of a mover effect μ and an atomicity effect τ:

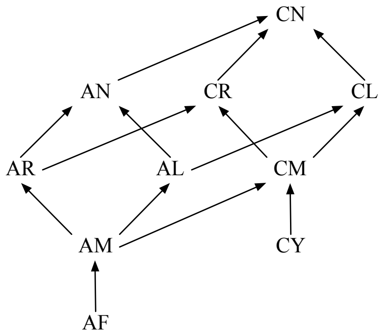
$$\kappa \quad ::= \quad \tau\mu$$

Note that not all combined effects are meaningful; in particular, AY and CF are contradictory: an atomic piece of code may not contain a yield, and code with yields cannot be considered functional.

We define the ordering relation and the join, iterative closure, and sequential composition operations on combined effects in a point-wise manner:

$$\begin{array}{rcll} \tau_1\mu_1 \sqsubseteq \tau_2\mu_2 & \text{iff} & \tau_1 \sqsubseteq \tau_2 \text{ and } \mu_1 \sqsubseteq \mu_2 & \\ \tau_1\mu_1 \sqcup \tau_2\mu_2 & \stackrel{\text{def}}{=} & \tau_3\mu_3 & \text{where } \tau_3 = \tau_1 \sqcup \tau_2 \text{ and } \mu_3 = \mu_1 \sqcup \mu_2 \\ \tau_1\mu_1;\tau_2\mu_2 & \stackrel{\text{def}}{=} & \tau_3\mu_3 & \text{where } \tau_3 = \tau_1;\tau_2 \text{ and } \mu_3 = \mu_1;\mu_2 \\ (\tau\mu)^* & \stackrel{\text{def}}{=} & \tau^*\mu^* & \end{array}$$

The following diagram summarizes the resulting lattice of combined effects·



4.4 Conditional Effects

Based on the previous discussion, the effect of acquiring a lock `m` is AR, since a lock acquire is a right-mover that contains no yield operations. However, if the lock `m` is already held by the current thread, then the re-entrant lock acquire is actually a no-op and could be more precisely characterized as a both-mover AM.

Figure 3: YIELDJAVA **Syntax**

$$\begin{array}{rcl} P \in \mathit{Program} & = & \overline{\mathit{defn}} \\ \mathit{defn} \in \mathit{Definition} & ::= & \texttt{class}\ c\ \texttt{\{}\ \overline{\mathit{field}}\ \overline{\mathit{meth}}\ \texttt{\}} \\ \mathit{field} \in \mathit{Field} & ::= & c\ f \\ \mathit{meth} \in \mathit{Method} & ::= & a\ c\ m(\overline{c\ x})\ \texttt{\{}\ e\ \texttt{\}} \\ & & \\ e, \ell \in \mathit{Expr} & ::= & x \mid \texttt{null} \\ & & \mid e_\gamma f \mid e_\gamma f = e \\ & & \mid e_\gamma m(\overline{e}) \mid e_\gamma m\texttt{\#}(\overline{e}) \\ & & \mid \texttt{new}\ c(\overline{e}) \mid e_\gamma \texttt{sync}\ e \mid \texttt{fork}\ e \\ & & \mid \texttt{let}\ x\ \texttt{=}\ e\ \texttt{in}\ e \mid \texttt{if}\ e\ e\ e \mid \texttt{while}\ e\ e \\ & & \\ \gamma \in \mathit{OptYield} & ::= & \texttt{.} \mid \texttt{..} \\ & & \\ x, y \in \mathit{Var} & & \\ c, d \in \mathit{ClassName} & & \\ f \in \mathit{FieldName} & = & \mathit{Normal} \cup \mathit{Final} \cup \mathit{Volatile} \\ m \in \mathit{MethodName} & & \\ a \in \mathit{Effect} & & \end{array}$$

We introduce *conditional effects* to capture situations like this where the effect of an operation depends on which locks are held by the current thread. We use ℓ to range over expressions that are functional (F). Such expressions always reliably denote the same lock. An effect a is then either a combined effect κ or an effect conditional on whether a lock ℓ is held:

$$a \quad ::= \quad \kappa \mid \ell\,?\,a_1 : a_2$$

We extend the calculation of iterative closure, sequential composition, and join operations to conditional effects as follows:

$$\begin{array}{rcl} (\ell\,?\,a_1 : a_2)^* & = & \ell\,?\,a_1^* : a_2^* \\ (\ell\,?\,a_1 : a_2); a & = & \ell\,?\,(a_1; a) : (a_2; a) \\ a; (\ell\,?\,a_1 : a_2) & = & \ell\,?\,(a; a_1) : (a; a_2) \\ (\ell\,?\,a_1 : a_2) \sqcup a & = & \ell\,?\,(a_1 \sqcup a) : (a_2 \sqcup a) \\ a \sqcup (\ell\,?\,a_1 : a_2) & = & \ell\,?\,(a \sqcup a_1) : (a \sqcup a_2) \end{array}$$

We also extend the effect ordering to conditional effects. To decide $a_1 \sqsubseteq a_2$, we use an auxiliary relation $\sqsubseteq^h_n$, where h is a set of locks known to be held by the current thread, and n is a set of locks known not to be held by the current thread. We define $a_1 \sqsubseteq a_2$ to be $a_1 \sqsubseteq^\emptyset_\emptyset a_2$ and check $a_1 \sqsubseteq^h_n a_2$ recursively as follows:

$$\frac{\kappa_1 \sqsubseteq \kappa_2}{\kappa_1 \sqsubseteq^h_n \kappa_2} \qquad \frac{\begin{array}{c} \ell \notin n \Rightarrow a_1 \sqsubseteq^{h \cup \{l\}}_n a \\ \ell \notin h \Rightarrow a_2 \sqsubseteq^h_{n \cup \{l\}} a \end{array}}{\ell\,?\,a_1 : a_2 \sqsubseteq^h_n a} \qquad \frac{\begin{array}{c} \ell \notin n \Rightarrow \kappa \sqsubseteq^{h \cup \{l\}}_n a_1 \\ \ell \notin h \Rightarrow \kappa \sqsubseteq^h_{n \cup \{l\}} a_2 \end{array}}{\kappa \sqsubseteq^h_n \ell\,?\,a_1 : a_2}$$

A similar notion of ordering was used for conditional atomicities in our previous work on atomicity checkers [21].

5. Type and Effect System

We now formalize our type system for the idealized language YIELDJAVA, a multithreaded subset of Java. This language does not include some Java features, such as primitive types, arrays, inheritance, and interfaces. However, it is sufficient to explore the most salient aspects of reasoning about thread interference. Section 6 describes how our implementation extends this idealized type system to support other Java features.

5.1 Syntax

Figure 3 presents the YIELDJAVA syntax. A program P consists of a sequence of class definitions $\overline{\mathit{defn}}$. Each class definition defn

associates a name with a body containing field and method declarations.

A field declaration includes a class type and a name. Field names are syntactically divided into three categories:

- *Normal* fields are mutable and free of race conditions. They include thread-local fields as well as thread-shared fields that are synchronized, for example, via locks.
- *Final* fields are immutable and thus race-free.
- *Volatile* fields are mutable, and may have concurrent conflicting accesses.

We assume that race freedom for *Normal* fields is verified separately by, for example, a race-free type system [10, 25, 1]. While not permitted in YIELDJAVA, our prototype does support racy accesses to non-`volatile` fields, as described in Section 6.

A method declaration

$$a\; c\; m(\overline{c\;x})\ \texttt{\{}\ e\ \texttt{\}}$$

defines a method m with return type c that takes parameters $\overline{x}$ of type $\overline{c}$. The self-reference variable `this` is implicitly bound to the receiving object in the method body e. The method declaration also includes its effect a, which may include lock expressions referring to any variables in scope, including `this` and $\overline{x}$.

We assume that all programs include a class `Unit`, which has no methods or fields. `Unit` is the type of expressions like `while` loops that produce no meaningful value. The language includes the special constant `null`, which has any class type, including `Unit`.

The object allocation expression $\texttt{new}\ c(\overline{e})$ creates a new object of type c and initializes its fields to the values computed from the expression sequence $\overline{e}$. Other YIELDJAVA expressions include field read and update, method calls, variable binding and reference, conditionals, loops, and fork. We also support synchronized blocks $e_1.\texttt{sync}\ e_2$, which are analogous to Java's synchronized statements

$$\texttt{synchronized (}e_1\texttt{) \{ }e_2\texttt{ \}}$$

Some of these constructs have special forms to document yield points, as summarized by the following table.

Expression Form	Non-Yielding	Yielding
Field Read	$e.f$	$e..f$
Field Write	$e.f = e$	$e..f = e$
Atomic Method Call	$e.m(\overline{e})$	$e..m(\overline{e})$
Non-Atomic Method Call	$e.m\#(\overline{e})$	$e..m\#(\overline{e})$
Lock Synchronization	$e.\texttt{sync}\ e$	$e..\texttt{sync}\ e$

5.2 Type Rules

The YIELDJAVA type system ensures that thread interference is observable only at explicitly annotated yield points. The core of the type system is a set of rules for reasoning about the effect of an expression, as captured by the judgment:

$$P; E \vdash e : c \cdot a$$

Here, e is an expression, c is the type of the expression, and a is an effect describing the behavior of e. The program P is included to provide access to class declarations, and the environment E maps free variables in e to their types:

$$E ::= \epsilon \mid E, c\;x$$

Figure 4 presents the complete set of rules for expressions, as well as additional judgments for reasoning about well-formed environments ($P \vdash E$), types ($P \vdash c$), effects ($P; E \vdash a$), methods ($P; E \vdash \mathit{meth}$), class declarations ($P \vdash \mathit{defn}$), and programs ($P \vdash \textsc{ok}$). We describe the most important rules defining these judgments:

[EXP VAR] and [EXP NULL] All variables are immutable in YIELDJAVA and thus have cooperability effect AF. That is, a variable access is atomic and yields a constant value. The constant `null` also has effect AF.

[EXP IF] and [EXP WHILE] The effect of a conditional expression is the effect of the *test* expression sequentially composed with the join of the *then* and *else* branches. Similarly, the effect of a while loop $\texttt{while}\ e_1\ e_2$ is the effect of the *test* e_1 composed with the iterative closure $(e_2; e_1)^*$ of the loop body followed by a subsequent evaluation of the test.

[EXP NEW] The object allocation rule first retrieves the definition of the class c from P, and then ensures the arguments $e_{1..n}$ match the types of the fields of c. The effect of the whole expression is the composition of effects of evaluating $e_{1..n}$ composed with the effect AM, reflecting that `new` is not functional (since re-evaluating an object allocation would not return the same object).

[EXP REF] The rule [EXP REF] handles a read $e.f$ of a *Normal* or *Final* field. The rule first checks that e has some type c, and extracts the type d of the field f from P. If f is *Normal* and thus race-free, the effect of accessing the field is AM because the access is guaranteed to commute with steps by other threads. If f is *Final*, the field's value is constant and the effect is AF.

[EXP REF RACE] A racy field read is a non-mover operation, since it may conflict with concurrent accesses by other threads. A racy read $e_\gamma f$ may be annotated with a yield point (if $\gamma =$ "$..$") or not (if $\gamma =$ "$.$"). We use the auxiliary function $\llbracket\gamma\rrbracket$ to map γ to the corresponding effect:

$$\begin{array}{rcl} \llbracket\ \rrbracket & : & \mathit{OptYield} \rightarrow \mathit{Effect} \\ \llbracket . \rrbracket & = & \textsc{af} \\ \llbracket .. \rrbracket & = & \textsc{cy} \end{array}$$

Thus, if the expression e has effect a, then the non-yielding racy access $e.f$ has effect $(a; \textsc{af}; \textsc{an})$, whereas the yielding racy access $e..f$ has effect $(a; \textsc{cy}; \textsc{an})$.

[EXP ASSIGN] and [EXP ASSIGN RACE] The rules for field updates are similar to those for field reads, with the additional requirement that *Final* fields cannot be modified.

[EXP SYNC] The type rule for the synchronized statement $\ell_\gamma\texttt{sync}\ e$ first checks that ℓ is a valid lock expression ($P; E \vdash_{\text{lock}} \ell$), meaning that ℓ must have effect AF to guarantee that it always denotes the same lock at run time.

The rule then computes the effect $\mathcal{S}(\ell, \gamma, a)$, where a is the atomicity of e, and γ specifies whether there is a yield point. The function S is defined as follows:

a	$\mathcal{S}(\ell, \gamma, a)$
κ	$\ell\ ?\ \kappa : (\llbracket\gamma\rrbracket; \textsc{ar}; \kappa; \textsc{al})$
$\ell\ ?\ a_1 : a_2$	$\mathcal{S}(\ell, \gamma, a_1)$
$\ell'\ ?\ a_1 : a_2$	$\ell'\ ?\ \mathcal{S}(\ell, \gamma, a_1) : \mathcal{S}(\ell, \gamma, a_2)$ if $\ell \neq \ell'$

If the body of the synchronized statement has a basic effect κ and the lock ℓ is already held, then the synchronized statement also has effect κ, since the acquire and release operations are no-ops. Note that in this case the yield operation is ignored, since it is unnecessary.

If the body has effect κ and the lock is not already held, then the synchronized statement has effect $(\llbracket\gamma\rrbracket; \textsc{ar}; \kappa; \textsc{al})$, since the execution consists of a potential yield point, followed by a right-mover (the acquire), followed by κ (the body), followed by a left-mover (the release).

Figure 4: YIELDJAVA **Type Rules**

$$\boxed{P; E \vdash e : c \cdot a}$$

[EXP VAR]
$$\frac{\begin{array}{c} P \vdash E \\ E = E_1, c\,x, E_2 \end{array}}{P; E \vdash x : c \cdot \mathrm{AF}}$$

[EXP NULL]
$$\frac{\begin{array}{c} P \vdash E \\ P \vdash c \end{array}}{P; E \vdash \texttt{null} : c \cdot \mathrm{AF}}$$

[EXP IF]
$$\frac{\begin{array}{c} P; E \vdash e_1 : d \cdot a_1 \\ P; E \vdash e_i : c \cdot a_i \quad \forall i \in 2..3 \end{array}}{P; E \vdash \texttt{if}\ e_1\ e_2\ e_3 : c \cdot (a_1; (a_2 \sqcup a_3))}$$

[EXP WHILE]
$$\frac{\begin{array}{c} P; E \vdash e_1 : c_1 \cdot a_1 \\ P; E \vdash e_2 : c_2 \cdot a_2 \end{array}}{P; E \vdash \texttt{while}\ e_1\ e_2 : \texttt{Unit} \cdot (a_1; (a_2; a_1)^*)}$$

[EXP REF]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot a \\ \texttt{class}\ c\ \texttt{\{}\ \ldots\ d\ f\ \ldots\ \texttt{\}} \in P \\ f \in \mathit{Normal} \implies a' = \mathrm{AM} \\ f \in \mathit{Final} \implies a' = \mathrm{AF} \end{array}}{P; E \vdash e.f : d \cdot (a; a')}$$

[EXP ASSIGN]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot a \qquad P; E \vdash e' : d \cdot a' \\ \texttt{class}\ c\ \texttt{\{}\ \ldots\ d\ f\ \ldots\ \texttt{\}} \in P \\ f \in \mathit{Normal} \end{array}}{P; E \vdash (e.f = e') : d \cdot (a; a'; \mathrm{AM})}$$

[EXP NEW]
$$\frac{\begin{array}{c} \texttt{class}\ c\ \texttt{\{}\ d_i\ x_i\ {}^{i \in 1..n}\ \ldots\ \texttt{\}} \in P \\ P; E \vdash e_i : d_i \cdot a_i \qquad \forall i \in 1..n \end{array}}{P; E \vdash \texttt{new}\ c\texttt{(}e_{1..n}\texttt{)} : c \cdot (a_1; \cdots; a_n; \mathrm{AM})}$$

[EXP REF RACE]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot a \\ \texttt{class}\ c\ \texttt{\{}\ \ldots\ d\ f\ \ldots\ \texttt{\}} \in P \\ f \in \mathit{Volatile} \end{array}}{P; E \vdash e_\gamma f : d \cdot (a; [\![\gamma]\!]; \mathrm{AN})}$$

[EXP ASSIGN RACE]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot a \qquad P; E \vdash e' : d \cdot a' \\ \texttt{class}\ c\ \texttt{\{}\ \ldots\ d\ f\ \ldots\ \texttt{\}} \in P \\ f \in \mathit{Volatile} \end{array}}{P; E \vdash (e_\gamma f = e') : d \cdot (a; a'; [\![\gamma]\!]; \mathrm{AN})}$$

[EXP SYNC]
$$\frac{\begin{array}{c} P; E \vdash_{\mathrm{lock}} \ell \\ P; E \vdash e : c \cdot a \end{array}}{P; E \vdash \ell_\gamma \texttt{sync}\ e : c \cdot \mathcal{S}(\ell, \gamma, a)}$$

[EXP INVOKE ATOMIC]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot a \\ \texttt{class}\ c\ \texttt{\{}\ \ldots\ \mathit{meth}\ \ldots\ \texttt{\}} \in P \\ \mathit{meth} = a'\ c'\ m\texttt{(}d_i\ x_i\ {}^{i \in 1..n}\texttt{)}\ \texttt{\{}\ e'\ \texttt{\}} \\ P; E \vdash e_i : d_i \cdot a_i \qquad \forall i \in 1..n \\ P; E \vdash a'[\texttt{this} := e, x_i := e_i\ {}^{i \in 1..n}] \uparrow a'' \qquad a'' \sqsubseteq \mathrm{AN} \end{array}}{P; E \vdash e_\gamma m\texttt{(}e_{1..n}\texttt{)} : c' \cdot (a; a_1; \cdots; a_n; [\![\gamma]\!]; a'')}$$

[EXP INVOKE COMPOUND]
$$\frac{\begin{array}{c} P; E \vdash e : c \cdot a \\ \texttt{class}\ c\ \texttt{\{}\ \ldots\ \mathit{meth}\ \ldots\ \texttt{\}} \in P \\ \mathit{meth} = a'\ c'\ m\texttt{(}d_i\ x_i\ {}^{i \in 1..n}\texttt{)}\ \texttt{\{}\ e'\ \texttt{\}} \\ P; E \vdash e_i : d_i \cdot a_i \qquad \forall i \in 1..n \\ P; E \vdash a'[\texttt{this} := e, x_i := e_i\ {}^{i \in 1..n}] \uparrow a'' \end{array}}{P; E \vdash e_\gamma m\texttt{\#(}e_{1..n}\texttt{)} : c' \cdot (a; a_1; \cdots; a_n; [\![\gamma]\!]; a'')}$$

[EXP FORK]
$$\frac{P; E \vdash e : c \cdot a}{P; E \vdash \texttt{fork}\ e : \texttt{Unit} \cdot \mathrm{AL}}$$

[EXP LET]
$$\frac{\begin{array}{c} P; E \vdash e_1 : c_1 \cdot a_1 \qquad P; E, c_1\ x \vdash e_2 : c_2 \cdot a_2 \\ P; E \vdash a_2[x := e_1] \uparrow a_2' \end{array}}{P; E \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : c_2 \cdot (a_1; a_2')}$$

$$\boxed{P \vdash E}$$

[ENV EMPTY]
$$\frac{}{P \vdash \epsilon}$$

[ENV X]
$$\frac{\begin{array}{c} P \vdash E \\ P \vdash c \qquad x \notin \mathit{dom}(E) \end{array}}{P \vdash (E, c\,x)}$$

$$\boxed{P \vdash c}$$

[CLASS NAME]
$$\frac{\texttt{class}\ c\ \texttt{\{}\ \ldots\ \texttt{\}} \in P}{P \vdash c}$$

$$\boxed{P; E \vdash a}$$

[AT BASE]
$$\frac{P \vdash E}{P; E \vdash \kappa}$$

[AT COND]
$$\frac{\begin{array}{c} P; E \vdash_{\mathrm{lock}} \ell \\ P; E \vdash a_i \qquad \forall i \in 1..2 \end{array}}{P; E \vdash \ell\,?\,a_1 : a_2}$$

$$\boxed{P; E \vdash_{\mathrm{lock}} e}$$

[LOCK EXP]
$$\frac{P; E \vdash e : c \cdot \mathrm{AF}}{P; E \vdash_{\mathrm{lock}} e}$$

$$\boxed{P; E \vdash a \uparrow a'}$$

[LIFT BASE]
$$\frac{P \vdash E}{P; E \vdash \kappa \uparrow \kappa}$$

[LIFT GOOD LOCK]
$$\frac{\begin{array}{c} P; E \vdash_{\mathrm{lock}} \ell \\ P; E \vdash a_i \uparrow a_i' \qquad \forall i \in 1..2 \end{array}}{P; E \vdash (\ell\,?\,a_1 : a_2) \uparrow (\ell\,?\,a_1' : a_2')}$$

[LIFT BAD LOCK]
$$\frac{\begin{array}{c} P; E \nvdash_{\mathrm{lock}} \ell \\ P; E \vdash a_i \uparrow a_i' \qquad \forall i \in 1..2 \end{array}}{P; E \vdash (\ell\,?\,a_1 : a_2) \uparrow (a_1' \sqcup a_2')}$$

$$\boxed{P; E \vdash \mathit{meth}}$$

[METHOD]
$$\frac{\begin{array}{c} P; E, \overline{d\,x} \vdash e : c \cdot a' \\ P; E, \overline{d\,x} \vdash a \\ a' \sqsubseteq a \end{array}}{P; E \vdash a\ c\ m\texttt{(}\overline{d\,x}\texttt{)}\ \texttt{\{}\ e\ \texttt{\}}}$$

$$\boxed{P \vdash \mathit{defn}}$$

[CLASS]
$$\frac{\begin{array}{ll} \mathit{field}_i = d_i\ f_i & \forall i \in 1..m \\ P \vdash d_i & \forall i \in 1..m \\ P; c\ \texttt{this} \vdash \mathit{meth}_i & \forall i \in 1..n \end{array}}{P \vdash \texttt{class}\ c\ \texttt{\{}\ \mathit{field}_{1..m}\ \mathit{meth}_{1..n}\ \texttt{\}}}$$

$$\boxed{P \vdash \mathrm{OK}}$$

[PROGRAM]
$$\frac{\begin{array}{c} P = \mathit{defn}_{1..n} \\ P \vdash \mathit{defn}_i \qquad \forall i \in 1..n \\ \mathit{ClassesOnce}(P)\ \ \mathit{FieldsOnce}(P)\ \ \mathit{MethodsOnce}(P) \end{array}}{P \vdash \mathrm{OK}}$$

If the body has conditional effect $\ell\,?\,a_1 : a_2$, where ℓ is the lock being acquired by this synchronized statement, then we ignore a_2 and recursively apply S to a_1, since ℓ is held within the synchronized body.

If the body has an effect that is conditional on some other lock ℓ', then we recursively apply S to both branches.

[EXP LET] This rule for `let` x `=` e_1 `in` e_2 infers effects a_1 and a_2 for e_1 and e_2, respectively. Care must be taken when constructing the effect for this expression because a_2 may refer to the let-bound variable x.

For example, the body of the following `let` expression produces an effect that is conditional on whether the lock `x` is held.

```
let x = e₁ in
  x.sync ...
```

Thus, we apply the substitution $[x := e_1]$ to yield a corresponding effect $a_2[x := e_1]$ that does not mention x. However, e_1 may not have effect AF, in which case $a_2[x := e_1]$ may not be a valid effect (because it could contain e_1 as part of a non-constant lock expression). As in our previous work [21], we use the judgment

$$P; E \vdash a_2[x := e_1] \uparrow a_2'$$

to *lift* the effect $a_2[x := e_1]$ to a well-formed effect a_2' that is greater than or equal to $a_2[x := e_1]$.

This "lifting" judgment is defined in Figure 4:

[LIFT BASE] Basic effects are always well-formed and remain unchanged when lifted.

[LIFT LOCK WELL-FORMED] If a conditional effect refers to a well-formed lock, this rule recursively lifts the two component effects.

[LIFT LOCK ILL-FORMED] If a conditional effect refers to an ill-formed lock expression, this rule removes the dependency on this lock by joining together the two recursively-lifted component effects.

[EXP INVOKE ATOMIC] This rule handles calls to atomic methods. The rule first extracts the appropriate method signature from P based on the receiver's type, and it verifies that the actual arguments match the declared parameter types. Finally, the rule computes the cooperability effect of the invocation. The method's specified effect a' may refer to `this` or the parameter names $x_{1..n}$. Therefore, we substitute

- the actual receiver e for `this`, and
- the actual arguments $e_{1..n}$ for the parameters $x_{1..n}$

to produce the effect $a'[\texttt{this} := e, x_i := e_i\ {}^{i \in 1..n}]$, and ensure that the resulting effect is valid by lifting it to an effect a'' that is well-formed in the current environment. This effect a'' must be an atomic effect and less than or equal to AN.

Like racy field accesses, a method invocation may be labeled with a yield point. Thus, the overall effect also includes $[\![\gamma]\!]$.

[EXP INVOKE COMPOUND] This rule applies to a method call $e_\gamma m\texttt{\#(}e_{1..n}\texttt{)}$ to a non-atomic method. It is similar to [EXP INVOKE ATOMIC], but removes the requirement that the computed effect a'' be atomic.

[EXP FORK] A fork expression `fork` e creates a new thread to evaluate e. Since a fork operation cannot commute past the operations of its child thread, fork operations are left movers and thus never start new transactions.

[METHOD], [CLASS], and [PROG] These rules verify the basic well-formedness requirements of methods, classes, and programs. The [PROG] rule uses the following additional predicates. (See [24] for their precise definition.)

- *ClassesOnce*(P): no class is declared twice in P.
- *FieldsOnce*(P): no field name is declared twice in a class.
- *MethodsOnce*(P): no method name is declared twice in a class.

5.3 Correctness

The appendix presents a formal semantics for YIELDJAVA, where a run-time state σ contains a program's class definitions, dynamically allocated objects, and dynamically created threads. This semantics satisfies the standard preservation property, whereby evaluation preserves well-typing of the run-time state. We then define both a *preemptive* and *cooperative* semantics for program behavior. The preemptive semantics interleaves the instructions of the various threads at instruction-level granularity, essentially modeling the behavior of a preemptive scheduler. The cooperative semantics performs context switches between threads only at explicitly marked yield points in the style of cooperative multitasking [3, 4, 9].

The central correctness result for this type system is that well-typed programs behave equivalently under both semantics. That is, if a well-typed program P can reach a final state σ under the preemptive semantics, then it can also reach that final state under the cooperative semantics. Therefore, it is sufficient to reason about the correctness of well-typed programs under the simpler cooperative semantics, since this correctness result also applies to executions under the preemptive semantics (and consequently, to executions on multicore hardware).

6. Implementation

We have developed an implementation called JCC that extends the YIELDJAVA type system to support the Java language.

JCC uses the standard Java field modifiers `final` and `volatile` to classify fields as either *Final* or *Volatile*; all other fields are considered *Normal*. We also introduce one new modifier, `racy`, to capture intentionally racy *Normal* fields. Our implementation assumes that correct field annotations are provided for the input program. Such annotations could be generated using RCC/JAVA [1] or any other analysis technique. For our experiments, we leveraged both that tool, as well as the FASTTRACK [20] race detector, to identify racy fields.

JCC supports annotations on methods to describe their effects. The following three keywords are sufficient to annotate most methods:

- `atomic`: an atomic non-mover method with effect AN.
- `mover`: an atomic both-mover method with effect AM.
- `compound`: a compound non-mover method with effect CN.

These effect annotations appear alongside the standard modifiers for a method, as in:

```
atomic public void m() { ... }
```

They may also be combined to form conditional effects, such as `(this ? mover : compound)`.

To further reduce the burden of annotating methods with cooperability effects, JCC uses carefully chosen defaults when annotations are absent. In essence, it assumes:

- fields are race free and
- all methods are atomic both-movers.

Access Type	Syntax	Effect
racy read	$e_1..f$#	$a_1; \text{CL}$
racy write	$e_1..f\# = e_2$	$a_1; a_2; \text{CL}$
write-guarded read (lock held)	$e_1.f$	$a_1; \text{AM}$
write-guarded read (lock not held)	$e_1.f$	$a_1; \text{AN}$
write-guarded write (lock held)	$e_1.f = e_2$	$a_1; a_2; \text{AN}$
race-free array read	$e_1[e_2]$	$a_1; a_2; \text{AM}$
race-free array write	$e_1[e_2] = e_3$	$a_1; a_2; a_3; \text{AM}$
racy array read	$e_1[e_2]\#$	$a_1; a_2; \text{CL}$
racy array write	$e_1[e_2]\# = e_3$	$a_1; a_2; a_3; \text{CL}$

Figure 5. Effects of additional operations (a_i is the effect of e_i).

We also permit more expressive (but more verbose) annotations to describe all elements of the conditional effect lattice when the keywords are not sufficient. The following illustrates the full syntax for effect annotations:

```
atomic non-mover void m1() { ... }

this.f ? (atomic functional) : (compound non-mover)
void m2() { ... }
```

As in YieldJava, field accesses and method invocations may be written using "`..`" in place of "`.`" to indicate interference points. Yielding synchronized statements use the syntax

```
..synchronized(e) { ... }
```

The JCC checker verifies that these yield points characterize all possible thread interference. It reports a warning whenever interference may occur at a program point not corresponding to a yield, or if a method's specification is not satisfied by its implementation.

The remainder of this section describes how JCC extends the YieldJava type system to support features of Java programs including subtyping, intentional races, write-guarded data, and arrays[1].

Subtyping and Covariant Cooperability Specifications. One major extension to the presented type system is the support for inheritance and subtyping. We permit the cooperability effect of the method to change covariantly, intuitively requiring, for example, that $b \sqsubseteq a$ in the following class definitions:

```
class C {                  class D extends C {
  a t m() { ... }            b t m() { ... }
}                          }
```

Fields with Races. Although data races should in general be avoided when possible, large programs often have some intentional races, which JCC supports via a `racy` annotation on *Normal* fields. A read from a racy field must be written as $e..f$#. Here, the double dots as usual indicate a yield point, and the trailing # identifies the racy nature of the read (and that the programmer needs to consider the consequences of Java's relaxed memory model [37]). The overall effect of $e..f$# is the composition of a yield and a non-mover memory access: $(\text{CY}; \text{AN}) = \text{CL}$. Writes are handled similarly. The rules for computing the effects of these operations, and those discussed below are summarized in Figure 5.

Write-Guarded Fields. YieldJava also supports write-guarded fields (such as the `shortestPathLength` field in Section 1) for which a protecting lock is held for all writes but not necessarily for reads. For such fields, a read while holding the protecting lock is a both-mover, since there can be no concurrent writes. However, a write with the lock held is still a non-mover, since there may be concurrent reads (that do not hold the lock).

Arrays. The YieldJava checker handles array accesses in a way analogous to *Normal* fields[2]. Racy array accesses must be annotated with "#" and are assumed to be yield points.)

7. Experimental Evaluation

To evaluate its effectiveness, we applied JCC to a variety of benchmark programs, including:

- a number of standard library classes from Java 1.4.2_19: namely `Inflater`, `Deflater`, `StringBuffer`, `String`, `PrintWriter`, `Vector`, and `ZipFile`;
- `sparse`, `raytracer`, `sor`, and `moldyn` from the Java Grande suite [32];
- `tsp`, a solver for the traveling salesman problem [48];
- and `elevator`, a real-time discrete event simulator [48].

These programs use a variety of synchronization idioms, and previous work has revealed a number of interesting concurrency bugs in these programs. Thus, they show the ability of our annotations to capture thread interference under various conditions and to highlight unintended, problematic interference. Three of these programs (`raytracer`, `sor`, and `moldyn`) use broken barrier implementations [20]. We discuss those problems below and use versions with corrected barrier code (named `raytracer-fixed`, `sor-fixed`, and `moldyn-fixed`) in our experiments.

All experiments were performed on a 2 GHz dual-core computer with 3 GB memory, using the Java HotSpot 64-bit Server VM, version 1.6.0_24. The JCC checker was able to analyze each of these benchmarks in under 2 seconds.

Figure 6 shows the size of each benchmark program, the time required to manually insert the JCC annotations into each program, and the number of annotations required to enable successful type checking. This count includes all `racy` field annotations, method specifications, and occurrences of "`..`" and "#". Even for programs comprising several thousand lines, the annotation burden is quite low. Each program was annotated and checked in about 10 to 30 minutes, and roughly one annotation per 30 lines of code was required. We did have some previous experience using these programs, which facilitated the annotation process, but since we intend JCC to be used during development, we believe our experience reflects the cost incurred by the intended use of our technique.

7.1 Precision of Thread Interference Annotations

Our experiments demonstrate that cooperability annotations serve as clear documentation of where interference may occur, thereby simplifying the complex task of reasoning about program behavior. To quantify this in one specific dimension, we consider the question

> "How many potential interference points must a programmer consider in a program annotated with various forms of non-interference specifications?"

Each specification form provides a particular semantic guarantee about where interference may occur, and we believe that isolating interference to as few program points as possible facilitates reasoning about code. The five different specifications (or semantic guarantees) we consider are as follows:

[1] Our implementation does not currently support generic classes due to limitations in the front-end checker upon which JCC is built, but supporting generic types would not be fundamentally problematic.

[2] Note that in Java, it is not possible to indicate that elements of an array are `final` or `volatile`.

Program	Size (lines)	Annot. Time (min.)	Annot. Count	Interference Points Preemptive	 Race	 Atomic	 AtomRace	 Cooperative	Unintended Yields
java.util.zip.Inflater	317	9	4	36	12	0	0	0	0
java.util.zip.Deflater	381	7	8	49	13	0	0	0	0
java.lang.StringBuffer	1,276	20	10	210	81	9	2	1	1
java.lang.String	2,307	15	5	230	87	6	2	1	0
java.io.PrintWriter	534	40	109	73	99	130	97	26	9
java.util.Vector	1,019	25	43	185	106	44	24	4	1
java.util.zip.ZipFile	490	30	62	120	105	85	53	30	0
sparse	868	15	19	329	98	48	14	6	0
tsp	706	10	45	445	115	437	80	19	0
elevator	1,447	30	64	454	146	241	60	25	0
raytracer-fixed	1,915	10	50	565	200	105	39	26	2
sor-fixed	958	10	32	249	99	128	24	12	0
moldyn-fixed	1,352	10	39	983	130	657	37	30	0
Total	13,570	231	490	3,928	1,291	1,890	432	180	13

Figure 6. Interference Points and Unintended Yields

- **Preemptive**: If a program has no synchronization specification, interference must be assumed to possibly occur on any access to a field or any lock acquire, since these operations may conflict with operations of concurrent threads. We exclude operations that do not cause interference, such as accesses to method-local variables, lock releases, method calls, etc. from this count.
- **Race**: If a program's specification distinguishes race-free fields from potentially racy fields, interference may be assumed to occur only on any access to a racy field or any lock acquire.
- **Atomic**: If a program's specification distinguishes atomic methods from non-atomic methods, thread interference may be considered to occur only in non-atomic methods. These interference points include field accesses, lock acquires, and also calls to an atomic method from a non-atomic context.
- **AtomicRace**: If a program specification distinguishes both racy fields and atomic methods, interference may be considered to occur in non-atomic methods at racy accesses, lock acquires, and calls to atomic methods.
- **Cooperative**: If a program specification identifies yield points, interference may be considered to occur only at those explicit yield points.

Figure 7: StringBuffer

```
public final class StringBuffer ··· {
  (this?mover:atomic) int length() { ... }
  (this?mover:atomic) void getChars(...) { ... }

  compound
  synchronized StringBuffer append(StringBuffer sb) {
    ...
    int len = sb.length();
    int newcount = count + len;
    if (newcount > value.length)
      expandCapacity(newcount);
    sb..getChars(0, len, value, count);
    count = newcount;
    return this;
  }
}
```

Figure 6 shows the number of interference points in each benchmark under each semantics. Benchmarks in which all methods are atomic, such as `Inflater`, have zero interference points under both atomic and cooperative semantics.

Overall, the results show that information about race conditions and atomic methods provides significant benefits over the Preemptive column. In particular, the number of interference points drops from 3,928 under the Preemptive column (which assumes no non-interference information) to 432 under AtomRace (which essentially characterizes prior techniques based on method-level atomicity and race condition information).

As shown in the "Cooperative" column, our cooperative type and effect system further reduces the number of possible interference points from 432 to 180, essentially because this analysis reasons more precisely about where thread interference may occur.

We sketch two situations that illustrate why cooperative reasoning is significantly more precise than AtomRace. First, for the TSP algorithm from Figure 1, AtomRace requires an interference point before each call to the atomic methods `path.isComplete()` and `path.children()` from within a non-atomic method. In contrast, our type system identifies that these two methods are more precisely characterized as both movers (which do not interfere with other threads), and so no yield point is necessary at these calls.

As a second example, consider a non-atomic method that contains two nested synchronized blocks. Under AtomRace, both acquires are an interference point. Under our analysis, both acquires are right-movers and so no yield is required before the second.

The method `StringBuffer.append()` in Figure 7 illustrates this kind of situation. The `append()` method acquires the `this` lock and then calls `sb.length()`, which is atomic (since the lock `sb` is not held). The lock acquire of `this` can move right to just before the `sb.length()` call, so no yield is required at that call. Conversely, a yield *is* required for the call `sb..getChars()`, since it is preceded by actions (such as the call to `sb.length()`) that are not right movers. This yield at `sb..getChars()` highlights that `append()` is `compound`, and may behave erroneously if `sb` is concurrently modified by other threads.

These two examples illustrate why the notions of race conditions and atomic methods are not by themselves sufficient to identify interference points in a precise manner, and the experimental results show that the cooperative type and effect system is significantly more precise in its ability to verify interference points.

7.2 Identifying Defects

The final column in Figure 6 shows the number of *unintended yield point* in each program. These are program points that suffer from thread interference in ways that we determined are unintentional or

Figure 8: RayTracer

```
class RayTracerRunner implements Runnable {
  int id;

  compound public void run() {
    // init
    br.DoBarrier#(id);
    // render

    ..synchronized (scene) {
      for(int i = 0; i < JGFRayTracerBench.nthreads; i++)
        if (id == i)
          JGFRayTracerBench..checksum1 =
            JGFRayTracerBench..checksum1 + checksum;
    }

    br.DoBarrier#(id);
    // cleanup
  }
}
```

possibly damaging based on manual code inspection. These unintended yields highlight concurrency bugs, such as atomicity violations and data races [19, 21]. We illustrate how JCC enables the programmer to identify several concurrency bugs in our benchmarks.

RayTracer. The raytracer benchmark uses a barrier `br` to coordinate several rendering threads, as shown in Figure 8. Although the initial barrier code was incorrect, it did not cause other unintended yields. After rendering, each thread acquires the lock `scene` before adding its local `checksum` to the global shared variable `checksum1`. However, each thread creates its own `scene` object, and thus acquiring `scene` fails to ensure mutual exclusion over the updates to `checksum1`. This is made clear by the explicit yields on the reads and writes of `checksum1`. [3]

Vector. A `Vector` constructor (see Figure 9) takes as argument a collection `c` and invokes `c`'s `size` method, allocates an array of that size, and copies elements from `c` into the array. The two yield annotations highlight that `c` may be modified by a concurrent thread in between requesting the size and copying the data, potentially resulting in an incorrectly initialized `Vector` or an `ArrayIndexOutOfBounds` exception. Similar pitfalls for the methods `removeAll(c)` and `retainAll(c)` [21] are also caught by JCC.

SOR. In the `sor` benchmark (see Figure 10), the computation threads synchronize on a barrier implemented as a shared two-dimensional array `sync`. Unfortunately, the barrier is broken, since the `volatile` keyword applies only to the array reference, not the array elements. Thus, the barrier synchronization code at the bottom of the main processing loop may not properly coordinate the threads, leading to races on the data array `G`. This problem is obvious when using JCC because yield annotations must be added in dozens of places, essentially to all accesses of `sync` and `G`.

When the barrier is fixed, we obtain much cleaner code: the yield count decreases from 40 to 12. In particular, the accesses to `G` between barrier calls are free of yields, signifying that between barriers, sequential reasoning is applicable. Figure 6 includes data for `sor-fixed`, the corrected version of the benchmark, which we believe is more representative of multithreaded Java programs.

Figure 9: Vector

```
interface Collection {
  (this ? mover : atomic) int size();
  (this ? mover : atomic) Object[] toArray(Object a[]);
}
class Vector {
  protected Object elementData[];
  protected int elementCount;

  compound public Vector(Collection c) {
    elementCount = c..size();
    elementData =
      new Object[(int)Math.min( (elementCount*110L)/100,
                                Integer.MAX_VALUE )];
    c..toArray(elementData);
  }
}
```

Figure 10: Original SOR Algorithm

```
class SORRunner implements Runnable {
  double G[][];
  volatile long sync[][];

  compound public void run() {
    ...
      for (int p = 0; p < 2*num_iterations; p++) {
        for (int i = ilow + (p%2); i < iupper; i=i+2) {
          ...
          for (int j=1; j < Nm1; j = j+2){

            G[i][j]# = omega_over_four *
              ( G[i-1][j]# + G[i+1][j]# +
                G[i][j-1]# + G[i][j+1]# ) +
              one_minus_omega * G[i][j]#;
          ...
          }
        }

        sync[id][0]# = sync[id][0]# + 1;
        if (id > 0)
          while (sync[id-1][0]# < sync[id][0]#) ;
        if (id < JGFSORBench.nthreads-1)
          while (sync[id+1][0]# < sync[id][0]#) ;
      }
  }
}
```

Moldyn. The `moldyn` benchmark uses a barrier object to synchronize the actions of multiple computation threads. The barrier object maintains an array

```
volatile boolean[] IsDone;
```

to record which threads are currently waiting at the barrier, but the elements of the array are prone to race conditions because again the `volatile` keyword applies only to the array reference and not the array elements. This bug leads to potential races on all data accesses intended to be synchronized by the barrier, and a large number (58) of yield annotations were necessary to document all such cases. Again, we report on the corrected version `moldyn-fixed` in Figure 6.

[3] We also note that if JCC were extended to identify locks used only by a single thread, we could remove the yield on the `synchronized` operation.

8. Related Work

Cooperability. *Cooperative multithreading* is a thread execution model in which context switching between threads may occur only at `yield` statements [3, 4, 9]. That is, cooperative multithreading allows concurrency but disallows parallel execution of threads. In contrast, cooperability guarantees behavior equivalent to cooperative multithreading, but actually allows execution in a preemptive manner, enabling full use of modern multicore hardware.

Automatic mutual exclusion is an execution model that proposes ensuring mutual exclusion by default [31]; `yield` statements demarcate where thread interference is permitted. A critical difference is that these `yield` statements are enforced at run time to provide serializability via transactional memory techniques; in contrast, the JCC checker guarantee serializability statically.

In prior work, we explored a type and effect system for cooperability [51], and dynamic analyses for checking cooperability and inferring yield annotations for legacy programs [52]. Others have explored *task types*, a data-centric approach to obtaining *pervasive atomicity* [33], a notion that is closely related to cooperability.

Race Freedom. A data race occurs when two threads simultaneously access a shared variable without synchronization, and at least one thread writes to that variable. Data races often reflect problems in synchronization, and expose a weak memory model to programmers, compromising software reliability. Data race freedom remedies this issue by guaranteeing behavior equivalent to executing with sequentially consistent [2].

The JCC checker relies on a race analysis to properly annotate racy variables, used as input to the cooperability analysis. There is extensive literature on how to find and fix data races efficiently. Dynamic race detectors may track the happens-before relation [20], implement the lockset algorithm [45], or combine both [39]. Sampling techniques are also used to make race detection more lightweight [8, 16]. Static race detectors may make use of a type system [10, 1], implement a static lockset algorithm [38, 41], or use model checking [42].

Atomicity and Transactional Memory. An atomic block is a lexically-scoped annotation that declares the sequence of instructions in that block to be free of thread interference. Atomicity and transactional memory both focus on ensuring that atomic blocks execute in a serializable manner, thus enjoying freedom from thread interference.

Atomicity is an analysis approach that checks if atomic blocks are serializable. Both static [21, 27, 49] and dynamic tools [18, 50, 22, 17] have been developed to check atomicity. *Transactional memory* enforces serializability of atomic blocks (or transactions) at run time. Both hardware [28, 14] and software [46, 26] implementation techniques have been developed, and the semantics of transactional memory have also been explored in depth [6].

While atomic blocks are clearly beneficial to program reasoning, one must still ascertain whether a piece of code is inside some atomic block to enjoy freedom from thread interference, hence introducing a form of bi-modal reasoning [51]. Furthermore, the regions of code outside atomic blocks are still subject to unconstrained preemptive scheduling, with all the traditional problems such schedulings pose.

Other Properties. *Deterministic parallelism* guarantees that the result of executing multiple threads is invariant across thread schedulings. There are various approaches to obtaining deterministic parallelism: static analyses [7], dynamic analyses [43, 12], as well as run-time enforcement [40, 13].

Linearizability, a popular correctness criterion, guarantees that the concurrent calls to a shared object execute atomically and satisfy a sequential specification [29, 47]. Shape analysis [5] and abstract interpretation [47] have been used to prove linearizability for small programs, while model checking has been used to refute linearizability [11].

9. Summary

Reasoning about the correctness of multithreaded software is notoriously difficulty under the preemptive semantics provided by multiprocessor and multicore architectures. This paper proposes a more modular approach for reasoning about multithreaded software correctness.

Under our approach, software is written using traditional synchronization idioms such as locks, but the programmer also explicitly documents intended sources of thread interference, which we refer to as yield points. The type system of this paper then verifies that these annotations identify all possible situations where interference may occur. Consequently, any well-typed program behaves *as if* it is executing under a cooperative semantics where context switches between threads happen only at yield points.

This cooperative semantics provides a much nicer foundation for subsequent reasoning about program behavior and correctness. In particular, intuitive sequential reasoning is now valid, except at explicitly marked yield points. One interesting avenue of future work would be to incorporate cooperative reasoning into a proof system, such as rely-guarantee reasoning. Another would be to extend cooperability to reason about determinism properties. Finally, the formal system described here could be adapted to other languages, such as C++ or X10.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

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

[3] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Annual Technical Conference*. USENIX Association, 2002.

[4] R. M. Amadio and S. D. Zilio. Resource control for synchronous cooperative threads. In *International Conference on Concurrency Theory (CONCUR)*, 2004.

[5] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *Computer Aided Verification (CAV)*, 2007.

[6] C. Blundell, E. Christopher, L. Milo, and M. K. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5:2006, 2006.

[7] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009.

[8] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: Proportional detection of data races. In *Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[9] G. Boudol. Fair cooperative multithreading. In *International Conference on Concurrency Theory (CONCUR)*, 2007.

[10] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 56–69, 2001.

[11] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-Up: a complete and automatic linearizability checker. In *Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[12] J. Burnim and K. Sen. Asserting and checking determinism for multithreaded programs. In *International Symposium on Foundations of Software Engineering (FSE)*, 2009.

[13] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2009.

[14] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[15] J. L. Eppinger, L. B. Mummert, and A. Z. Spector. *Camelot and Avalon: A Distributed Transaction Facility*. Morgan Kaufmann, 1991.

[16] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk. Effective data-race detection for the kernel. In *Operating Systems Design and Implementation (OSDI)*, 2010.

[17] A. Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Computer Aided Verification (CAV)*, 2008.

[18] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages (POPL)*, 2004.

[19] C. Flanagan and S. N. Freund. Adversarial memory for detecting destructive races. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 244–254, 2010.

[20] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. *Commun. ACM*, 53(11):93–101, 2010.

[21] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *Transactions on Programming Languages and Systems (TOPLAS)*, 30(4):1–53, 2008.

[22] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[23] C. Flanagan and S. Qadeer. Types for atomicity. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2003.

[24] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Symposium on Principles of Programming Languages (POPL)*, pages 171–183, 1998.

[25] D. Grossman. Type-safe multithreading in Cyclone. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2003.

[26] T. Harris and K. Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[27] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2004.

[28] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *International Symposium on Computer Architecture (ISCA)*, 1993.

[29] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[30] C. A. R. Hoare. Towards a theory of parallel programming. In *Operating Systems Techniques*, volume 9 of *A.P.I.C. Studies in Data Processing*, pages 61–71, 1972.

[31] M. Isard and A. Birrell. Automatic mutual exclusion. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, 2007.

[32] Java Grande Forum. Java Grande benchmark suite. Available at `http://www.javagrande.org`, 2008.

[33] A. Kulkarni, Y. D. Liu, and S. F. Smith. Task types for pervasive atomicity. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[34] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.

[35] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the Symposium on Operating Systems Principles*, pages 111–122, 1987.

[36] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *Language Design for Reliable Software*, pages 128–137, 1977.

[37] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *Symposium on Principles of Programming Languages (POPL)*, 2005.

[38] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[39] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2003.

[40] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: efficient deterministic multithreading in software. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[41] P. Pratikakis, J. S. Foster, and M. Hicks. Context-sensitive correlation analysis for detecting races. In *Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[42] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[43] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. In *European Symposium on Programming (ESOP)*, 2009.

[44] C. Sadowski and J. Yi. Applying usability studies to correctness conditions: A case study of cooperability. In *Onward! Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2010.

[45] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4), 1997.

[46] N. Shavit and D. Touitou. Software transactional memory. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 1995.

[47] V. Vafeiadis. Automatically proving linearizability. In *Computer Aided Verification (CAV)*, 2010.

[48] C. von Praun and T. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[49] C. von Praun and T. Gross. Static detection of atomicity violations in object-oriented programs. In *Workshop on Formal Techniques for Java-like Programs*, 2003.

[50] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Transactions on Software Engineering*, 32:93–110, Feb. 2006.

[51] J. Yi and C. Flanagan. Effects for cooperable and serializable threads. In *Workshop on Types in Language Design and Implementation (TLDI)*, 2010.

[52] J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011.

A. Formal Semantics of YIELDJAVA

A.1 Runtime Syntax

To characterize the runtime behavior of YIELDJAVA programs, we extend the syntax of expressions to include object addresses ρ and the `in-sync` construct. We assume that addresses are divided into distinct sets, one for each type of object. The membership relation $\rho \in \mathit{Addr}_c$ indicates that the object at address ρ is of type c. The expression `in-sync` $\rho\, e$ describes an expression e that is executing while holding the lock ρ.

$$
\begin{array}{rclcl}
e & \in & \mathit{Expr} & ::= & \cdots \mid \texttt{in-sync}\ \rho\ e \\
v & \in & \mathit{Value} & ::= & \rho \mid \texttt{null} \\
\rho & \in & \mathit{Addr} & = & \displaystyle\bigcup_{c\ \in\ \mathit{ClassName}} \mathit{Addr}_c \\
\\
T & \in & \mathit{ThreadState} & ::= & e \mid \mathbf{ready}\ e \mid \mathbf{wrong} \\
t & \in & \mathit{Thread} & = & \{1, 2, 3, \ldots\} \\
\mathit{obj} & \in & \mathit{Object} & ::= & \{\, \overline{f = v}\, \}^o \\
o & \in & \mathit{Lock} & ::= & \bot \mid \mathit{Thread} \\
\sigma & \in & \mathit{State} & = & \mathit{Program} \\
 & & & & \hat{\cup}\ (\mathit{Addr} \rightharpoonup \mathit{Object}) \\
 & & & & \hat{\cup}\ (\mathit{Thread} \rightharpoonup \mathit{ThreadState})
\end{array}
$$

$$\text{where } A \mathbin{\hat{\cup}} B \ \overset{\text{def}}{=}\ \{a \cup b \mid a \in A \ \wedge\ b \in B\}.$$

A thread state T is either:

- an expression e,
- **ready** e, which describes a newly created thread ready to execute an expression e, or
- $\mathbf{wrong}$, which occurs when a thread tries to dereference `null`.

A *thread map* is a partial map from threads to thread state. An *object map* is a partial map from addresses to objects. An object is a sequence of field values for a given class, along with a lock o indicating which thread is holding the lock for that object, if any. (As in Java, every object has a lock implicitely associated with it at run time).

Finally, the run-time state σ is a combination of class definitions, an object map, and a thread map.

Figure 11 presents the formal transition rules for the evaluation of YIELDJAVA programs. Each rule describes a transition step $\rightarrow_t$ that could be taken by the thread t. We use an evaluation context $\mathcal{E}$, an expression with a "hole" $[\,]$, to indicate the next subexpression to evaluate. We use the form $\sigma[t \mapsto T]$ to indicate a state that agrees with σ at all threads except t, where it maps to T. Similarly, we use $\sigma[\rho \mapsto \{\, \overline{f = v}\, \}^o]$ and $\sigma[\rho.f \mapsto v]$ to indicate updates to addresses and object fields. Evaluation finishes when each thread state is either a value or $\mathbf{wrong}$.

To obtain the set of *locks held* by a particular evaluation context $\mathcal{E}$, we define the function $\mathit{locks} :: \mathit{EvalCtxt} \rightarrow \mathit{Lockset}$, such that

$$\rho \in \mathit{locks}(\mathcal{E}) \Leftrightarrow \mathcal{E} \equiv \mathcal{E}'[\texttt{in-sync}\ \rho\ \mathcal{E}'']$$

Given locksets h_1, h_2 and h_3, we indicate these are pairwise disjoint with the notation

$$h_1 \not\cap h_2 \not\cap h_3$$

We define two semantics for states. In the preemptive semantics $\rightarrow$, the steps of the various threads are interleaved nondeterministically, and a thread can perform a step at any time (provided that thread is not blocked). This nondeterminism means that reasoning about the behavior of programs under the preemptive semantics is very difficult.

In the cooperative semantics $\rightarrow_c$, context switches between threads can happen only at yield operations, which are indicated by dots ("**..**"), or on thread termination. In more detail, a thread t is *yielding in* σ (or simply *yielding* if the context is clear) if for $\sigma(t) = T$,

1. $T = \mathbf{ready}\ e$
2. $T \in \mathit{Value}$,
3. $T = \mathbf{wrong}$,
4. $T = \mathcal{E}[\rho..f]$,
5. $T = \mathcal{E}[\rho..f = v]$,
6. $T = \mathcal{E}[\rho..\texttt{sync}\ e\,]$,
7. $T = \mathcal{E}[\rho..m(\overline{v})]$, or
8. $T = \mathcal{E}[\rho..m\texttt{\#}(\overline{v})]$.

The thread t is *cooperatively enabled* in a state σ if for all t' in $\mathit{Dom}(\sigma) \setminus \{t\}$, we have t' is yielding in σ. Thus, t can only take a cooperative step if all other threads are yielding. This means that if t takes a cooperative step, then every other thread is either at a yield operation, is not enabled (either terminated at a value or $\mathbf{wrong}$), or has not started yet. A state is *yielding* if all threads are yielding in that state.

Figure 11: YIELDJAVA **Semantics**

Evaluation contexts

$$\begin{array}{rcl} \mathcal{E} & ::= & [\,] \mid \texttt{new}\, c(\overline{v}, \mathcal{E}, \overline{e}) \mid \mathcal{E}_\gamma f \mid \mathcal{E}_\gamma f \texttt{ = } e \mid v_\gamma f \texttt{ = } \mathcal{E} \\ & \mid & \mathcal{E}_\gamma m\texttt{(}\overline{e}\texttt{)} \mid \mathcal{E}_\gamma m\texttt{\#(}\overline{e}\texttt{)} \mid v_\gamma m\texttt{(}\overline{v}, \mathcal{E}, \overline{e}\texttt{)} \mid v_\gamma m\texttt{\#(}\overline{v}, \mathcal{E}, \overline{e}\texttt{)} \\ & \mid & \texttt{let}\; x \texttt{ = } \mathcal{E}\; \texttt{in}\; e \mid \texttt{if}\; \mathcal{E}\; e\; e \mid \mathcal{E}_\gamma \texttt{sync}\; e \mid \texttt{in-sync}\; \rho\, \mathcal{E} \end{array}$$

Transition rules

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma f]] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[v]] \;\; \text{if}\ \sigma(\rho) = \{ \cdots, f = v, \cdots \}^o \qquad \text{[RED READ]}$$

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma f \texttt{ = } v]] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[v], \rho.f \mapsto v] \qquad \text{[RED WRITE]}$$

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma m\texttt{(}v_{1..n}\texttt{)}]] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[e[\texttt{this} := \rho, x_i := \underline{v_i}^{i\in 1..n}]]] \qquad \text{[RED INVOKE]}$$
$$\text{if}\ \rho \in \mathit{Addr}_c\ \text{and}\ \texttt{class}\ c\ \texttt{\{}\ \cdots\ a\ d'\ m\texttt{(}\overline{d\ x}\texttt{)}\ \{\ e\ \}\ \cdots\ \texttt{\}} \in P$$

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma m\texttt{\#(}v_{1..n}\texttt{)}]] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[e[\texttt{this} := \rho, x_i := \underline{v_i}^{i\in 1..n}]]] \qquad \text{[RED INVOKE COMPOUND]}$$
$$\text{if}\ \rho \in \mathit{Addr}_c\ \text{and}\ \texttt{class}\ c\ \texttt{\{}\ \cdots\ a\ d'\ m\texttt{(}\overline{d\ x}\texttt{)}\ \{\ e\ \}\ \cdots\ \texttt{\}} \in P$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{new}\, c(v_{1..n})]] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[\rho], \rho \mapsto \{ f_i = v_i\ ^{i\in 1..n}\ \overline{\}^{\perp}]} \qquad \text{[RED NEW]}$$
$$\text{if}\ \rho \notin \mathit{dom}(\sigma)\ \text{and}\ \rho \in \mathit{Addr}_c\ \text{and}\ \texttt{class}\ c\ \texttt{\{}\ \overline{t\ f}\ \cdots\ \texttt{\}} \in P$$

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma \texttt{sync}\ e], \rho \mapsto \{ \overline{f = v} \}^{\perp}] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[\texttt{in-sync}\ \rho\, e], \rho \mapsto \{ \overline{f = v} \}^{t}] \qquad \text{[RED SYNC]}$$

$$\sigma[t \mapsto \mathcal{E}[\rho_\gamma \texttt{sync}\ e], \rho \mapsto \{ \overline{f = v} \}^{t}] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[e], \rho \mapsto \{ \overline{f = v} \}^{t}] \qquad \text{[RED SYNC REENTRANT]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{in-sync}\ \rho\, v], \rho \mapsto \{ \overline{f = v} \}^{t}] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[v], \rho \mapsto \{ \overline{f = v} \}^{\perp}] \qquad \text{[RED IN-SYNC]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{fork}\, e]] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[\texttt{null}], t' \mapsto \textbf{ready}\ e] \quad \text{if}\ t' \notin \mathit{Dom}(\sigma) \qquad \text{[RED FORK]}$$

$$\sigma[t \mapsto \textbf{ready}\ e] \quad \rightarrow_t \quad \sigma[t \mapsto [e]] \qquad \text{[RED READY]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{let}\ x \texttt{ = } v\ \texttt{in}\, e]] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[e[x := v]]] \qquad \text{[RED LET]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{if}\ v\ e_2\ e_3]] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[e_2]] \quad \text{if}\ v \neq \texttt{null} \qquad \text{[RED IF-NONNULL]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{if}\ \texttt{null}\ e_2\ e_3]] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[e_3]] \qquad \text{[RED IF-NULL]}$$

$$\sigma[t \mapsto \mathcal{E}[\texttt{while}\ e_1\ e_2]] \quad \rightarrow_t \quad \sigma[t \mapsto \mathcal{E}[\texttt{if}\ e_1\ (e_2;\ \texttt{while}\ e_1\ e_2)\ \texttt{null}]] \qquad \text{[RED WHILE]}$$

$$\sigma[t \mapsto \mathcal{E}[e]] \quad \rightarrow_t \quad \sigma[t \mapsto \textbf{wrong}] \qquad \text{[RED WRONG]}$$
$$\text{if}\ e \in \{\texttt{null}_\gamma f,\ \texttt{null}_\gamma f \texttt{ = } v,\ \texttt{null}_\gamma m\texttt{(}\overline{v}\texttt{)},\ \texttt{null}_\gamma m\texttt{\#(}\overline{v}\texttt{)},\ \texttt{null}_\gamma \texttt{sync}\ e' \}$$

Transition relations

$$\begin{array}{llll} (\textit{preemptive semantics}) & \sigma \;\; \rightarrow \;\; \sigma' & & \text{if}\ \sigma \rightarrow_t \sigma' \\ (\textit{cooperative semantics}) & \sigma \;\; \rightarrow_c \;\; \sigma' & & \text{if}\ \sigma \rightarrow_t \sigma' \\ & & & \text{and}\ \sigma(t)\ \text{is cooperatively enabled} \end{array}$$

A.2 Type Rules for Run-Time Constructs

We expand the type rules to include run-time constructs; these rules are shown in Figure 12. We extend the definition of an environment to include allocated addresses, like such:

$$E \ ::= \ \epsilon \mid E, c\, x \mid E, c\, \rho$$

- Addresses have effect AF, the constant effect, and have the appropriate type c for the object at address ρ.
- The rule [EXP INSYNC] first checks that the lock expression ρ and enclosed expression e are both valid. If so, the effect for an `in-sync` expression depends on what the effect a of e is, as determined by the following function $\mathcal{SI}(\rho, a)$.

a	$\mathcal{SI}(\rho, a)$
κ	κ; AL
$\rho\,?\,a_1 : a_2$	$\mathcal{SI}(\rho, a_1)$
$\rho'\,?\,a_1 : a_2$	$\rho'\,?\,\mathcal{SI}(\rho, a_1) : \mathcal{SI}(\rho, a_2)$ if $\rho \neq \rho'$

  Briefly, if a is a combined effect κ, then the overall effect is κ sequentially followed by AL, the effect for a lock release operation. If a is an effect conditional on holding lock ρ, and the `in-sync` expression is also holding ρ, then we simplify a by having $\mathcal{SI}$ recurse down the holding branch. However, if a is a conditional effect on some different lock ρ', then we recurse $\mathcal{SI}$ down both branches of the conditional.
- The judgment $P \vdash E$ is expanded to include well-formedness constraints on addresses in the environment E. An address must have a valid type and not already be listed in the environment.
- The judgment $P; E \vdash T$ simply checks if a thread state is well-formed. If a thread state is an expression e or the construct **ready** e, then the rule checks if e is well-formed in its antecedent. Otherwise, the thread state is **wrong**, which is always well-formed.
- The judgment $P; E \vdash \mathit{obj} : c$ checks if a given type c is a class in the program, and verifies that the sequence of types for the object values matches the sequence of types for the class definition.
- Finally, the state judgment $\vdash \sigma$ ensures the program is well-formed, and also that each object and thread state in the state is well-formed. This judgment takes all objects in the object map and creates a well-formed environment E of allocated objects, and uses E to judge objects and thread states.

B. Correctness of Type System

B.1 Preservation

We prove that at each step of the evaluation, the program remains well-typed.

THEOREM 1 (Preservation). *If* $\vdash \sigma$ *and* $\sigma \to \sigma'$ *then* $\vdash \sigma'$.

Proof Suppose $\sigma \to_t \sigma'$. Let:

$$\sigma = P \cup [\, \rho_i \mapsto \mathit{obj}_i^{\,i \in 1..n} \,] \cup [\, t_j \mapsto T_j^{\,j \in 1..m} \,]$$

From the rule [STATE] we have:

$$\begin{array}{l} P \vdash \text{OK} \\ E = c_1\, \rho_1, \ldots, c_n\, \rho_n \\ \rho_i \in \mathit{Addr}_{c_i} \qquad \forall i \in 1..n \\ P; E \vdash \mathit{obj}_i : c_i \qquad \forall i \in 1..n \\ P; E \vdash T_j \qquad \forall j \in 1..m \end{array}$$

Figure 12: Additional YIELDJAVA Type Rules

$$\boxed{P; E \vdash e : c \cdot a}$$

[EXP ADDR]

$$\frac{\begin{array}{c} P \vdash E \\ E = E_1, c\, \rho, E_2 \\ \rho \in \mathit{Addr}_c \end{array}}{P; E \vdash \rho : c \cdot \text{AF}}$$

[EXP INSYNC]

$$\frac{P; E \vdash_{\text{lock}} \rho \qquad P; E \vdash e : c \cdot a}{P; E \vdash \texttt{in-sync}\ \rho\, e : c \cdot \mathcal{SI}(\rho, a)}$$

$$\boxed{P \vdash E}$$

[ENV ADDR]

$$\frac{\begin{array}{c} P \vdash E \\ P \vdash c \qquad \rho \notin \mathit{dom}(E) \end{array}}{P \vdash (E, c\, \rho)}$$

$$\boxed{P; E \vdash T}$$

[THREAD OK] [THREAD READY] [THREAD WRONG]

$$\frac{P; E \vdash e : c \cdot a}{P; E \vdash e} \qquad \frac{P; E \vdash e : c \cdot a}{P; E \vdash \textbf{ready}\ e} \qquad \frac{}{P; E \vdash \textbf{wrong}}$$

$$\boxed{P; E \vdash \mathit{obj} : c}$$

[OBJECT]

$$\frac{\begin{array}{l} \texttt{class}\ c\, \{\, d_i\, f_i^{\ i \in 1..n} \ \ldots\, \} \in P \\ P; E \vdash v_i : d_i \cdot a_i \qquad \forall i \in 1..n \end{array}}{P; E \vdash \{\, f_i = v_i^{\ i \in 1..n} \,\}^o : c}$$

$$\boxed{\vdash \sigma}$$

[STATE]

$$\frac{\begin{array}{ll} P \vdash \text{OK} & \\ E = c_1\, \rho_1, \ldots, c_m\, \rho_m & \\ \rho_j \in \mathit{Addr}_{c_j} & \forall j \in 1..m \\ P; E \vdash \mathit{obj}_j : c_j & \forall j \in 1..m \\ P; E \vdash T_k & \forall k \in 1..n \end{array}}{\vdash P \cup [\, \rho_j \mapsto \mathit{obj}_j^{\ j \in 1..m} \,] \cup [\, t_k \mapsto T_k^{\ k \in 1..n} \,]}$$

Since the program component of the state remains unchanged (i.e. $P = P'$), we need to show, for some n' and m',

$$\begin{array}{l} P; E' \vdash \mathit{obj}_i : c_i \qquad \forall i \in 1..n' \\ P; E' \vdash T_j \qquad \forall j \in 1..m' \end{array}$$

in the environment E' capturing all allocated objects in σ':

$$E' = c_1\, \rho_1, \ldots, c_n\, \rho_{n'}$$

such that $\forall i \in 1..n'$. $\rho_i \in \mathit{Addr}_{c_i}$.

Proof is by case analysis on the reduction rule used for $\sigma \to_t \sigma'$.

- [RED READ] In this case we have:

$$\begin{array}{l} \sigma(t) = \mathcal{E}[\rho_\gamma f] \\ \sigma'(t) = \mathcal{E}[v] \\ \sigma(\rho) = \{ \ldots f = v \ldots \}^o \end{array}$$

  The object map remains constant, hence $E = E'$. Since all other threads and the object map does not change, it is sufficient to show $P; E \vdash \sigma'(t)$.

  From [THREAD OK] we know:

$$P; E \vdash \mathcal{E}[\rho_\gamma f] : c \cdot a$$

By Lemma 2 we have

$$P; E \vdash \rho_\gamma f : c_{rd} \cdot a_{rd}$$

Since this can only be concluded by [EXP REF] or [EXP REF RACE] we also know

$$\begin{array}{l} P; E \vdash \rho : d \cdot \text{AF} \\ \mathtt{class}\ d\ \{\ \ldots\ c_{rd}\ f\ \ldots\ \} \in P \end{array}$$

Since we know $\sigma(\rho) = \{\, \ldots, f = v, \ldots \}^o$ is well-formed, we can conclude from [OBJECT]

$$\begin{array}{l} P; E \vdash v : c_{rd} \cdot a_v \\ a_v = \text{AF} \end{array}$$

By rules [EXP REF] and [EXP REF RACE], we know that a_{rd} must be one of AF, AM, AN, or CL. Since AF is a subeffect of any one of these effects, $a_v \sqsubseteq a_{rd}$ and we can conclude by Lemma 4

$$P; E \vdash \mathcal{E}[v] : c \cdot a'$$

- [RED WRITE] For this case we have:

$$\begin{array}{r} \sigma(t) = \mathcal{E}[\rho_\gamma f\ \mathtt{=}\ v] \\ \sigma'(t) = \mathcal{E}[v] \\ \sigma'(\rho) = \{\, \ldots f = v \ldots \}^o \end{array}$$

Since the object map in σ' does not add or remove objects, and all objects do not change their type, we have $E' = E$. Since all other threads do not change it is sufficient to show:

$$\begin{array}{l} P; E \vdash \sigma'(t) \\ P; E \vdash \sigma'(\rho) \end{array}$$

From [THREAD OK] we know:

$$P; E \vdash \mathcal{E}[\rho_\gamma f = v] : c \cdot a$$

By Lemma 2 we have:

$$P; E \vdash \rho_\gamma f = v : c_{wr} \cdot a_{wr}$$

This judgment can only be concluded with [EXP ASSIGN] or [EXP ASSIGN RACE], so we also know:

$$\begin{array}{l} P; E \vdash \rho : d_1 \cdot \text{AF} \\ P; E \vdash v : c_{wr} \cdot \text{AF} \\ \mathtt{class}\ d_1\ \{\ \ldots\ c_{wr}\ f\ \ldots \} \in P \end{array}$$

From [EXP ASSIGN] and [EXP ASSIGN RACE], we know a_{wr} must be one of AM, AN, or CL. Since we know $P; E \vdash v : c_{wr} \cdot \text{AF}$ and AF $\sqsubseteq a_{wr}$, by Lemma 4 we can conclude:

$$P; E \vdash \mathcal{E}[v] : d \cdot a'$$

Thus we have shown $P; E \vdash \sigma'(t)$. To show $P; E \vdash \sigma'(\rho)$ we note that $P; E \vdash \sigma(\rho) : c_a$ and $\sigma'(\rho) = \{\ldots f = v \ldots\}^o$. Since we know by [EXP ASSIGN] and [EXP ASSIGN RACE] that $P; E \vdash v : d_2 \cdot \text{AF}$ we can conclude by [OBJECT]:

$$P; E \vdash \sigma'(\rho) : c_a$$

- [RED INVOKE] In this case, let $\theta = [\mathtt{this} := \rho, x_i := v_i^{\,i \in 1..n}]$, where

$$\begin{array}{l} \sigma(t) = \mathcal{E}[\rho_\gamma m\mathtt{(}v_{1..n}\mathtt{)}] \\ \sigma'(t) = \mathcal{E}[\theta(e)] \\ \quad \rho \in \mathit{Addr}_c \\ \mathtt{class}\ c\ \{\ \ldots a_m\ d_m\ m\mathtt{(}d_i\ x_i\ {}^{i \in 1..n}\mathtt{)\{}\ e\ \mathtt{\}} \ldots \mathtt{\}} \in P \end{array}$$

The object map remains constant, hence $E = E'$. Since all other threads and the object map do not change it is sufficient to show $P; E \vdash \sigma'(t)$. From [THREAD OK] we have:

$$P; E \vdash \mathcal{E}[\rho_\gamma m\mathtt{(}v_{1..n}\mathtt{)}] : d \cdot a$$

By Lemma 2 we have:

$$P; E \vdash \rho_\gamma m\mathtt{(}v_{1..n}\mathtt{)} : d_m \cdot a_{iv}$$

This can only be concluded by the rules [EXP INVOKE] so we also have:

$$\begin{array}{ll} P; E \vdash \rho : c \cdot \text{AF} & \\ P; E \vdash v_i : d_i \cdot a_i & \forall i \in 1..n \\ P; E \vdash \theta(a_m) \uparrow a'' & \\ a_{iv} = (\text{AF}; a_1; \ldots; a_n; a'') & \\ \mathtt{class}\ c\ \{\ \ldots M \ldots \} \in P & \\ M = a_m\ d_m\ m\mathtt{(}d_i\ x_i\ {}^{i \in 1..n}\mathtt{)\{}\ e\ \mathtt{\}} & \end{array}$$

From [CLASS] we know $P; c\ \mathtt{this} \vdash M$ and from [METHOD]:

$$\begin{array}{l} P; c\ \mathtt{this}, d_i\ x_i\ {}^{i \in 1..n} \vdash e : d_m \cdot a'_m \\ P; d_i\ x_i\ {}^{i \in 1..n} \vdash a_m \\ a'_m \sqsubseteq a_m \end{array}$$

By Lemma 6:

$$P; \emptyset \vdash \theta(e) : d_m \cdot \theta(a'_m)$$

Since $a'_m \sqsubseteq a_m$, by Lemma 1 we have $\theta(a'_m) \sqsubseteq \theta(a_m)$ and therefore $\theta(a'_m) \sqsubseteq a_{iv}$. Thus by Lemma 4:

$$P; E \vdash \mathcal{E}[e[\mathtt{this} := \rho, x_i := v_i^{i \in 1..n}]] : d \cdot a'$$

- [RED INVOKE COMPOUND] Similar to previous case.
- [RED NEW] In this case we have:

$$\begin{array}{l} \sigma(t) = \mathcal{E}[\mathtt{new}\ c\mathtt{(}v_{1..n}\mathtt{)}] \\ \sigma'(t) = \mathcal{E}[\rho] \\ \sigma'(\rho) = \{\, f_i = v_i\ {}^{i \in 1..n}\, \}^\perp \\ \quad \rho \notin \mathit{Dom}(\sigma) \\ \quad \rho \in \mathit{Addr}_c \\ \mathtt{class}\ c\ \{\ d_i\ f_i\ {}^{i \in 1..n}\ \ \ldots\ \} \in P \end{array}$$

Since ρ is the only new address in the environment, we have $E' = (E, c\ \rho)$. Since all other threads do not change it is sufficient to show

$$\begin{array}{l} P; E' \vdash \sigma'(t) \\ P; E' \vdash \sigma'(\rho) : c \end{array}$$

From [THREAD OK] we have:

$$P; E \vdash \mathcal{E}[\mathtt{new}\ c\mathtt{(}v_{1..n}\mathtt{)}] : d \cdot a$$

By Lemma 2 we know:

$$P; E \vdash \mathtt{new}\ c\mathtt{(}v_{1..n}\mathtt{)} : c \cdot a_{new}$$

This can only be concluded by [EXP NEW] so we also know:

$$\begin{array}{ll} P; E \vdash v_i : d_i \cdot \text{AF} & \forall i \in 1..n \\ a_{new} = (\overline{\text{AF}}; \text{AM}) & \end{array}$$

From the transition rules we know that $\rho \in \mathit{Addr}_c$ so we can conclude, via rule [EXP ADDR],

$$\begin{array}{l} P; E' \vdash \rho : c \cdot a_\rho \\ a_\rho = \text{AF} \end{array}$$

Since $a_\rho \sqsubseteq a_{new}$ we can conclude from Lemma 4:

$$P; E' \vdash \mathcal{E}[\rho] : d \cdot a'$$

So we have shown $P; E' \vdash \sigma'(t)$. To show $P \vdash \sigma'(\rho) : c$ we note that from the transition rules we know $\rho \in \mathit{Addr}_c$ and $\mathtt{class}\ c\ \{\ \overline{d}\ \overline{f}\ \ \ldots\ \} \in P$. Also from [EXP NEW] we know $P; E \vdash v_i : d_i \cdot a_i$ for all $i \in 1..n$. Therefore by [OBJECT] we can conclude $P; E' \vdash \{\, \overline{f = v}\, \}^o : c$. Thus we have shown $P; E' \vdash \sigma'(\rho) : c$.

- [RED SYNC] In this case we have:

$$\begin{array}{r} \sigma(t) = \mathcal{E}[\rho_\gamma \mathtt{sync}\ e\,] \\ \sigma(\rho) = \{\, \overline{f} = \overline{v}\, \}^\perp \\ \sigma'(t) = \mathcal{E}[\mathtt{in\text{-}sync}\ \rho\ e] \\ \sigma'(\rho) = \{\, \overline{f} = \overline{v}\, \}^t \end{array}$$

Since only the lock state of the object at ρ changes, we have $E' = E$. Since all other threads and objects besides ρ do not change it is sufficient to show:

$$\begin{array}{l} P; E \vdash \sigma'(t) \\ P; E \vdash \sigma'(\rho) \end{array}$$

From [THREAD OK] we know:

$$P; E \vdash \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,] : c \cdot a$$

By Lemma 2 we have:

$$P; E \vdash \rho_\gamma \texttt{sync}\ e\ : c_{syn} \cdot a_{syn}$$

This can only be concluded by [EXP SYNC] so we also know:

$$\begin{array}{l} P; E \vdash_{\text{lock}} \rho \\ P; E \vdash e : c_{syn} \cdot a_e \\ a_{syn} = \mathcal{S}(\rho, \gamma, a_e) \end{array}$$

By [LOCK EXP] we know $P; E \vdash \rho : d \cdot \text{AF}$. From this and the antecedents of [EXP INSYNC] we may conclude:

$$\begin{array}{l} P; E \vdash \texttt{in-sync}\ \rho\ e : c_{syn} \cdot a_{ins} \\ a_{ins} = \mathcal{SI}(\rho, a_e) \end{array}$$

Observe that ρ is *not* held by thread t in state σ. Then since the following satisfies the preconditions of Lemma 14,

$$\begin{array}{l} P; E \vdash_{\text{lock}} \rho \\ P; E \vdash a_e \\ \rho \in n \text{ for some lockset } n \end{array}$$

where n is disjoint from locks held by $\mathcal{E}$, we obtain the subeffect relation

$$\llbracket\gamma\rrbracket; \mathcal{SI}(\rho, a_e) \sqsubseteq_n \mathcal{S}(\rho, \gamma, a_e)$$

Since the following satisfies the preconditions of Lemma 5,

$$\begin{array}{l} P; E \vdash \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,] : c \cdot a \\ P; E \vdash \rho_\gamma \texttt{sync}\ e\ : c_{syn} \cdot \mathcal{S}(\rho, \gamma, a_e) \\ \rho_\gamma \texttt{sync}\ e \text{ is not a value} \\ P; E \vdash \texttt{in-sync}\ \rho\ e : c_{syn} \cdot \mathcal{SI}(\rho, a_e) \\ \exists\ n\ .\ \llbracket\gamma\rrbracket; \mathcal{SI}(\rho, a_e) \sqsubseteq_n \mathcal{S}(\rho, \gamma, a_e) \end{array}$$

we obtain our desired result

$$P; E \vdash \mathcal{E}[\texttt{in-sync}\ \rho\ e] : c \cdot a'$$

and hence we have shown

$$P; E' \vdash \sigma'(t)$$

From [OBJECT] we know

$$\begin{array}{l} P; E \vdash \{\,\overline{f} = \overline{v}\,\}^\perp : c_o \\ \texttt{class}\ c_o\ \texttt{\{}\ \overline{d}\ \overline{f} \ldots\ \texttt{\}} \in P \\ P; E \vdash \overline{v} : \overline{d} \cdot \overline{a} \end{array}$$

From this we can also conclude $P; E \vdash \{\,\overline{f} = \overline{v}\,\}^t : c_o$ and thus we have $P; E \vdash \sigma'(\rho)$.

- [RED SYNC REENTRANT] In this case we have

$$\begin{array}{r} \sigma(t) = \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,] \\ \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t \\ \sigma'(t) = \mathcal{E}[e] \\ \sigma'(\rho) = \{\,\overline{f} = \overline{v}\,\}^t \end{array}$$

Since the object map does not change, we have $E' = E$. Since all other threads besides t and objects do not change, it is sufficient to show

$$P; E \vdash \sigma'(t)$$

From [THREAD OK] we know

$$P; E \vdash \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,] : c \cdot a$$

By Lemma 2, the expression in the hole is well-typed:

$$P; E \vdash \rho_\gamma \texttt{sync}\ e\ : c_{syn} \cdot a_{syn}$$

This can only be concluded by rule [EXP SYNC] so we also know:

$$\begin{array}{l} P; E \vdash_{\text{lock}} \rho \\ P; E \vdash e : c_{syn} \cdot a_e \\ a_{syn} = \mathcal{S}(\rho, \gamma, a_e) \end{array}$$

By Lemma 15, we have

$$a_e \sqsubseteq^{\{\rho\}} \mathcal{S}(\rho, \gamma, a_e)$$

Observe that ρ is held by thread t at states σ and σ', thus $\rho \in \mathit{locks}(\mathcal{E})$. It follows that

$$\text{AF}; a_e \sqsubseteq_{\emptyset}^{\emptyset \cup \mathit{locks}(\mathcal{E})} \mathcal{S}(\rho, \gamma, a_e)$$

By Lemma 5, we may conclude

$$P; E \vdash \mathcal{E}[e] : c \cdot a'$$

and hence we have shown our desired result

$$P; E' \vdash \sigma'(t)$$

- [RED IN-SYNC] In this case we have

$$\begin{array}{r} \sigma(t) = \mathcal{E}[\texttt{in-sync}\ \rho\ v] \\ \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t \\ \sigma'(t) = \mathcal{E}[v] \\ \sigma'(\rho) = \{\,\overline{f} = \overline{v}\,\}^\perp \end{array}$$

Since only the lock state of the object at ρ changes, we have $E' = E$. Since all other threads and objects besides ρ do not change it is sufficient to show:

$$\begin{array}{l} P; E \vdash \sigma'(t) \\ P; E \vdash \sigma'(\rho) \end{array}$$

From [THREAD OK] we know:

$$P; E \vdash \mathcal{E}[\texttt{in-sync}\ \rho\ v] : c \cdot a$$

By Lemma 2 we have:

$$P; E \vdash \texttt{in-sync}\ \rho\ v : c_{ins} \cdot a_{ins}$$

This can only be concluded by [EXP INSYNC] so we also know:

$$\begin{array}{l} P; E \vdash_{\text{lock}} \rho \\ P; E \vdash v : c_{ins} \cdot \text{AF} \\ a_{ins} = \mathcal{SI}(\rho, \text{AF}) = \text{AF}; \text{AL} \end{array}$$

Since $\text{AF} \sqsubseteq \text{AF}; \text{AL}$, by Lemma 4 we have $P; E \vdash \mathcal{E}[v] : c \cdot a'$. Thus we have shown $P; E \vdash \sigma'(t)$.
From [OBJECT] we know

$$\begin{array}{l} P; E \vdash \{\,\overline{f} = \overline{v}\,\}^t : c_o \\ \texttt{class}\ c_o\ \texttt{\{}\ \overline{d}\ \overline{f} \ldots\ \texttt{\}} \in P \\ P; E \vdash \overline{v} : \overline{d} \cdot \overline{a} \end{array}$$

From this we can also conclude $P; E \vdash \{\,\overline{f} = \overline{v}\,\}^\perp : c_o$ and thus we have $P; E \vdash \sigma'(\rho)$.

- [RED FORK] In this case we have:

$$\begin{array}{r} \sigma(t) = \mathcal{E}[\texttt{fork}\ e] \\ \sigma'(t) = \mathcal{E}[\texttt{null}] \\ \sigma'(t') = \textbf{ready}\ e \end{array}$$

where $t' \notin \mathit{Dom}(\sigma)$.
The object map remains constant, hence $E = E'$. Since all threads other than t and t' and the object map do not change it is sufficient to show:

$$\begin{array}{l} P; E \vdash \sigma'(t) \\ P; E \vdash \sigma'(t') \end{array}$$

From [THREAD OK] we know

$$P; E \vdash \mathcal{E}[\texttt{fork}\, e] : c \cdot a$$

By Lemma 2 we have

$$P; E \vdash \texttt{fork}\, e : c_f \cdot a_f$$

Since this can only be concluded by [EXP FORK] we also know:

$$\begin{aligned} &P; E \vdash e : d \cdot a_e \\ &c_f = \texttt{Unit} \\ &a_f = \text{AL} \end{aligned}$$

From [EXP NULL] we can conclude $P; E \vdash \texttt{null} : \texttt{Unit} \cdot \text{AF}$. Since $\text{AF} \sqsubseteq \text{AL}$, we can conclude by Lemma 4:

$$P; E \vdash \mathcal{E}[\texttt{null}] : c \cdot a'$$

This shows $P; E \vdash \sigma'(t)$ and since we know $P; E \vdash e : d \cdot a_e$, by rule [THREAD READY] we have $P; E \vdash \sigma'(t')$.

- [RED READY] In this case we have

$$\begin{aligned} \sigma(t) &= \textbf{ready}\ e \\ \sigma'(t) &= [e] \end{aligned}$$

Since the object map does not change, we have $E' = E$. Since all other threads besides t and objects do not change, it is sufficient to show

$$P; E \vdash \sigma'(t)$$

From [THREAD READY] we know

$$P; E \vdash e : c \cdot a$$

By [THREAD OK] we may conclude

$$P; E \vdash e$$

Thus we have shown $P; E \vdash \sigma'(t)$.

- [RED LET] In this case we have:

$$\begin{aligned} \sigma(t) &= \mathcal{E}[\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\, e] \\ \sigma'(t) &= \mathcal{E}[e[x := v]] \end{aligned}$$

The object map remains constant, hence $E = E'$. Since all other threads and the object map does not change, it is sufficient to show $P; E \vdash \sigma'(t)$.
From [THREAD OK] we know

$$P; E \vdash \mathcal{E}[\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\, e] : c \cdot a$$

By Lemma 2 we have:

$$P; E \vdash \texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\, e : c_{let} \cdot a_{let}$$

Since this judgment can only be concluded from [EXP LET] we also have:

$$\begin{aligned} &P; E \vdash v : c_v \cdot \text{AF} \\ &P; E, c_v\ x \vdash e : c_{let} \cdot a_e \\ &P; E \vdash a_e[x := v] \uparrow a'_e \\ &a_{let} = (\text{AF}; a'_e) \end{aligned}$$

From Lemma 6 we have:

$$P; E \vdash e[x := v] : c_{let} \cdot a_e[x := v]$$

By Lemma 7 we know:

$$P; E, c_v\ x \vdash a_e$$

This allows us to conclude via Lemma 9 and $P; E \vdash v : c_v \cdot \text{AF}$ that:

$$P; E \vdash a_e[x := v] \uparrow a_e[x := v]$$

By Lemma 8 we know that a'_e is well-typed: $P; E \vdash a'_e$. Thus $a'_e = a_e[x := v]$ (since lifting only changes effects if locks are not held and we know that all locks in $a_e[x := v]$ are held).

This allows us to conclude that $a_e[x := v] \sqsubseteq a_{let}$ and thus by Lemma 4 we have:

$$P; E \vdash \mathcal{E}[e[x := v]] : c \cdot a'$$

- [RED IF-NULL] In this case we have

$$\begin{aligned} \sigma(t) &= \mathcal{E}[\texttt{if null}\ e_2\ e_3] \\ \sigma'(t) &= \mathcal{E}[e_3] \end{aligned}$$

The object map remains constant, hence $E = E'$. Since all other threads and the object map does not change, it is sufficient to show $P; E \vdash \sigma'(t)$.
From [THREAD OK] we know:

$$P; E \vdash \mathcal{E}[\texttt{if null}\ e_2\ e_3] : c \cdot a$$

By Lemma 2 we know that:

$$P; E \vdash \texttt{if null}\ e_2\ e_3 : c_{if} \cdot a_{if}$$

which can only be concluded by [EXP IF] so we also have:

$$\begin{aligned} &P; E \vdash \texttt{null} : d \cdot \text{AF} \\ &P; E \vdash e_2 : c_{if} \cdot a_2 \\ &P; E \vdash e_3 : c_{if} \cdot a_3 \\ &a_{if} = (\text{AF}; a_2 \sqcup a_3) \end{aligned}$$

By Lemma 1 we know that $a_3 \sqsubseteq a_{if}$ so by Lemma 4 we can conclude that

$$P; E \vdash \mathcal{E}[e_3] : c \cdot a'$$

- [RED IF-NONNULL] In this case we have

$$\begin{aligned} \sigma(t) &= \mathcal{E}[\texttt{if}\ v\ e_2\ e_3] \\ \sigma'(t) &= \mathcal{E}[e_3] \end{aligned}$$

The object map remains constant, hence $E = E'$. Since all other threads and the object map does not change, it is sufficient to show $P; E \vdash \sigma'(t)$.
From [THREAD OK] we know:

$$P; E \vdash \mathcal{E}[\texttt{if}\ v\ e_2\ e_3] : c \cdot a$$

By Lemma 2 we know that:

$$P; E \vdash \texttt{if}\ v\ e_2\ e_3 : c_{if} \cdot a_{if}$$

which can only be concluded by [EXP IF] so we also have:

$$\begin{aligned} &P; E \vdash v : d \cdot \text{AF} \\ &P; E \vdash e_2 : c_{if} \cdot a_2 \\ &P; E \vdash e_3 : c_{if} \cdot a_3 \\ &a_{if} = (\text{AF}; a_2 \sqcup a_3) \end{aligned}$$

By Lemma 1 we know that $a_2 \sqsubseteq a_{if}$ so by Lemma 4 we can conclude that

$$P; E \vdash \mathcal{E}[e_2] : c \cdot a'$$

- [RED WHILE] In this case we have:

$$\begin{aligned} \sigma(t) &= \mathcal{E}[\texttt{while}\ e_1\ e_2] \\ \sigma'(t) &= \mathcal{E}[\texttt{if}\ e_1 (e_2; \texttt{while}\ e_1\ e_2)\ \texttt{null}] \end{aligned}$$

The object map remains constant, hence $E = E'$. Since all other threads and the object map does not change it is sufficient to show $P; E \vdash \sigma'(t)$. From [THREAD OK] we know:

$$P; E \vdash \mathcal{E}[\texttt{while}\ e_1\ e_2] : c \cdot a$$

By Lemma 2 we know that:

$$P; E \vdash \texttt{while}\ e_1\ e_2 : \texttt{Unit} \cdot a_w$$

and from [EXP WHILE] we have

$$\begin{aligned} &P; E \vdash e_1 : d \cdot a_1 \\ &P; E \vdash e_2 : d' \cdot a_2 \\ &a_w = (a_1; (a_2; a_1)^*) \end{aligned}$$

By Lemma 10 we know:

$$P; E \vdash \texttt{if}\ e_1(e_2; \texttt{while}\ e_1\ e_2)\ \texttt{null} : \texttt{Unit} \cdot a_w$$

So by Lemma 4 we can conclude

$$P; E \vdash \mathcal{E}[\texttt{if}\ e_1(e_2; \texttt{while}\ e_1\ e_2)\ \texttt{null}] : c \cdot a'$$

- [RED WRONG] In this case we have

  $$\begin{aligned} \sigma(t) &= \mathcal{E}[e] \\ \sigma'(t) &= \mathbf{wrong} \end{aligned}$$

  Since the object map does not change, we have $E' = E$. Since all other threads and the object map do no change it is sufficient to show

  $$P; E \vdash \sigma'(t)$$

  Since $\sigma'(t) = \mathbf{wrong}$ we can conclude $P; E \vdash \mathbf{wrong}$ directly from the rule [THREAD WRONG].

B.2 Mutual Exclusion

In this section, we demonstrate that evaluation preserves the mutually exclusive usage of locks: at any time, at most only a single thread is in a critical section on a lock.

The judgment $ls \vdash_{cs} T$ relates locksets ls to thread states T: if T contains an expression e, the judgment holds if ls contains all locks ρ for which $e \equiv \mathcal{E}[\texttt{in-sync}\ \rho\ e']$. We abuse notation slightly to say $\texttt{in-sync} \in e$ and $\texttt{in-sync} \in \mathcal{E}$ if $e \equiv \mathcal{E}'[\texttt{in-sync}\ \rho\ e']$ or $\mathcal{E} \equiv \mathcal{E}'[\texttt{in-sync}\ \rho\ \ \mathcal{E}'']$, respectively, for some e, e', ρ, $\mathcal{E}'$, and $\mathcal{E}''$.

The judgment $\vdash_{cs} \sigma$ checks the well-formedness of states: the judgment holds if for every thread t, when we obtain the set of locks ls held by t in the state, then for the thread state T corresponding to t, the judgment $ls \vdash_{cs} T$ holds. Informally, the set of locks held by t in the state describes all critical sections that t is in. Since every lock in the state is held by at most one thread, this implies that at most one thread is in a critical section for that lock.

Figure 13: Judgments for Mutual Exclusion

$$\boxed{ls \vdash_{cs} T}$$

[CS EXP]

$$\frac{\texttt{in-sync} \notin e}{\emptyset \vdash_{cs} e}$$

[CS IN-SYNC]

$$\frac{ls \vdash_{cs} e \qquad \rho \notin ls}{ls \cup \{\rho\} \vdash_{cs} \texttt{in-sync}\ \rho\ e}$$

[CS NOT IN-SYNC]

$$\frac{ls \vdash_{cs} e \qquad \texttt{in-sync} \notin \mathcal{E}}{ls \vdash_{cs} \mathcal{E}[e]}$$

[CS READY]

$$\frac{\emptyset \vdash_{cs} e}{\emptyset \vdash_{cs} \mathbf{ready}\ \ e}$$

[CS WRONG]

$$\frac{}{ls \vdash_{cs} \mathbf{wrong}}$$

$$\boxed{\vdash_{cs} \sigma}$$

[CS STATE]

$$\frac{\begin{array}{ll} ls_k \vdash_{cs} T_k & \forall k \in 1..n \\ ls_k = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^{t_k}\} & \forall k \in 1..n \end{array}}{\vdash_{cs} P \cup [\ \rho_j \mapsto obj_j\ ^{j \in 1..m}\ ] \cup [\ t_k \mapsto T_k\ ^{k \in 1..n}\ ]}$$

We show that well-formed critical sections are preserved under evaluation:

THEOREM 2 (Mutual Exclusion). *If* $\vdash_{cs}\ \sigma$ *and* $\sigma \rightarrow \sigma'$, *then* $\vdash_{cs} \sigma'$.

Proof Suppose that $\sigma \rightarrow_t \sigma'$. Let

$$\sigma = P \cup [\ \rho_j \mapsto obj_j^{\,j \in 1..m}\ ] \cup [\ t_k \mapsto T_k^{\,k \in 1..n}\ ]$$

From the rule [CS STATE] we have

$$\begin{array}{ll} ls_k \vdash_{cs} T_k & \forall k \in 1..n \\ ls_k = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^{t_k}\} & \forall k \in 1..n \end{array}$$

We need to show

$$ls_k \vdash_{cs} T_k \qquad \forall k \in 1..n'$$

for some n', where

$$ls_k = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^{t_k}\} \qquad \forall k \in 1..n'$$

in the state

$$\sigma' = P \cup [\ \rho_j \mapsto obj_j^{\,j \in 1..m'}\ ] \cup [\ t_k \mapsto T_k^{\,k \in 1..n'}\ ]$$

Proof is by case analysis on the reduction rule used for $\sigma \rightarrow_t \sigma'$.

- [RED READ]. In this case, we have

  $$\begin{aligned} \sigma(t) &= \mathcal{E}[\rho_\gamma f] \\ \sigma'(t) &= \mathcal{E}[v] \\ \sigma(\rho) &= \{\ldots f = v \ldots\}^o \end{aligned}$$

  All objects and all threads other than t do not change, hence it is sufficient to show

  $$ls \vdash_{cs} \mathcal{E}[v]$$

  where $ls = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t\}$.
  We have, by assumption,

  $$ls \vdash_{cs} \mathcal{E}[\rho_\gamma f]$$

  By Lemma 21, the derivation for this judgment includes the following judgment

  $$ls_r \vdash_{cs} \rho_\gamma f$$

  where, by rule [CS EXP], we have $ls_r = \emptyset$.
  Also, we know the following, by rule [CS EXP]:

  $$\begin{aligned} ls_v &\vdash_{cs} v \\ ls_v &= \emptyset \end{aligned}$$

  Then by Lemma 22, we may conclude with our desired result

  $$\begin{aligned} ls' &\vdash_{cs} \mathcal{E}[v] \\ ls' &= ls \end{aligned}$$

- [RED WRITE]. In this case, we have

  $$\begin{aligned} \sigma(t) &= \mathcal{E}[\rho_\gamma f\ \texttt{=}\ v] \\ \sigma'(t) &= \mathcal{E}[v] \\ \sigma'(\rho) &= \{\ldots f = v \ldots\}^o \end{aligned}$$

  No object has a lock changed, and all threads other than t do not change, hence it is sufficient to show

  $$ls \vdash_{cs} \mathcal{E}[v]$$

  where $ls = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t\}$.
  We have, by assumption,

  $$ls \vdash_{cs} \mathcal{E}[\rho_\gamma f\ \texttt{=}\ v]$$

  By Lemma 21, the derivation for this judgment includes the following judgment

  $$ls_w \vdash_{cs} \rho_\gamma f\ \texttt{=}\ v$$

where, by rule [CS EXP], we have $ls_w = \emptyset$.
Also, we know the following, by rule [CS EXP]:

$$\begin{array}{l} ls_v \vdash_{cs} v \\ ls_v = \emptyset \end{array}$$

Then by Lemma 22, we may conclude with our desired result

$$\begin{array}{l} ls' \vdash_{cs} \mathcal{E}[v] \\ ls' = ls \end{array}$$

- [RED INVOKE]. In this case, let $\theta = [\texttt{this} := \rho, x_i := v_i{}^{i \in 1..n}]$, where

$$\begin{array}{r} \sigma(t) = \mathcal{E}[\rho_\gamma m\texttt{(}v_{1..n}\texttt{)}] \\ \sigma'(t) = \mathcal{E}[\theta(e)] \\ \rho \in \mathit{Addr}_c \\ \texttt{class}\ c\ \texttt{\{}\ \ldots a_m\ d_m\ m\texttt{(}d_i\ x_i{}^{i \in 1..n}\texttt{)\{}\ e\ \texttt{\}}\ldots\texttt{\}} \in P \end{array}$$

All objects and all threads other than t do not change; hence it is sufficient to show

$$ls \vdash_{cs} \mathcal{E}[\theta(e)]$$

where $ls = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t\}$.
We have, by assumption,

$$ls \vdash_{cs} \mathcal{E}[\rho_\gamma m\texttt{(}v_{1..n}\texttt{)}]$$

By Lemma 21, the derivation for this judgment includes the following judgment

$$ls_m \vdash_{cs} \rho_\gamma m\texttt{(}v_{1..n}\texttt{)}$$

where, by rule [CS EXP], we have $ls_m = \emptyset$.
Also, the method body e cannot contain an `in-sync` run-time expression (due to well-formedness), and the θ substitution does not replace any part of e with an `in-sync` expression. Hence we know the following, by rule [CS EXP]:

$$\begin{array}{l} ls_b \vdash_{cs} \theta(e) \\ ls_b = \emptyset \end{array}$$

Then by Lemma 22, we may conclude with our desired result

$$\begin{array}{l} ls' \vdash_{cs} \mathcal{E}[\theta(e)] \\ ls' = ls \end{array}$$

- [RED INVOKE COMPOUND]. Similar to previous argument.
- [RED NEW]. In this case, we have

$$\begin{array}{r} \sigma(t) = \mathcal{E}[\texttt{new}\ c\texttt{(}v_{1..n}\texttt{)}] \\ \sigma'(t) = \mathcal{E}[\rho] \\ \sigma'(\rho) = \{\, f_i = v_i{}^{i \in 1..n}\,\}^\perp \\ \rho \notin \mathit{Dom}(\sigma) \\ \rho \in \mathit{Addr}_c \\ \texttt{class}\ c\ \texttt{\{}\ d_i\ f_i{}^{i \in 1..n}\ \ldots\ \texttt{\}} \in P \end{array}$$

We have a new object in the environment that is not locked by any thread. All other objects and all threads other than t do not change, hence it is sufficient to show

$$ls \vdash_{cs} \mathcal{E}[\rho]$$

where $ls = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t\}$.
We have, by assumption,

$$ls \vdash_{cs} \mathcal{E}[\texttt{new}\ c\texttt{(}v_{1..n}\texttt{)}]$$

By Lemma 21, the derivation for this judgment includes the following judgment

$$ls_n \vdash_{cs} \texttt{new}\ c\texttt{(}v_{1..n}\texttt{)}$$

where, by rule [CS EXP], we have $ls_n = \emptyset$.
Also, we know the following, by rule [CS EXP]:

$$\begin{array}{l} ls_v \vdash_{cs} \rho \\ ls_v = \emptyset \end{array}$$

Then by Lemma 22, we may conclude with our desired result

$$\begin{array}{l} ls' \vdash_{cs} \mathcal{E}[\rho] \\ ls' = ls \end{array}$$

- [RED SYNC]. In this case we have:

$$\begin{array}{r} \sigma(t) = \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,] \\ \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^\perp \\ \sigma'(t) = \mathcal{E}[\texttt{in-sync}\ \rho\ e] \\ \sigma'(\rho) = \{\,\overline{f} = \overline{v}\,\}^t \end{array}$$

The object at ρ changes from being unlocked by any thread to being locked by t. All other objects and all threads other than t do not change; hence it is sufficient to show

$$ls \cup \{\rho\} \vdash_{cs} \mathcal{E}[\texttt{in-sync}\ \rho\ e]$$

where $ls = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t\}$. Observe that we have $\rho \notin ls$.
We have, by assumption,

$$ls \vdash_{cs} \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,]$$

By Lemma 21, the derivation for this judgment includes the following judgment

$$ls_s \vdash_{cs} \rho_\gamma \texttt{sync}\ e$$

where, necessarily, we have $ls_s = \emptyset$ by rule [CS EXP]: by inspection of the reduction rules, e cannot contain an `in-sync` expression.
By rule [CS IN-SYNC], we have

$$\{\rho\} \vdash_{cs} \texttt{in-sync}\ \rho\ e$$

Then by Lemma 22, we may conclude with our desired result

$$\begin{array}{l} ls' \vdash_{cs} \mathcal{E}[\texttt{in-sync}\ \rho\ e] \\ ls' = ls \cup \{\rho\} \end{array}$$

- [RED SYNC REENTRANT]. In this case we have

$$\begin{array}{r} \sigma(t) = \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,] \\ \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t \\ \sigma'(t) = \mathcal{E}[e] \\ \sigma'(\rho) = \{\,\overline{f} = \overline{v}\,\}^t \end{array}$$

All objects and all threads other than t do not change; hence it is sufficient to show

$$ls \vdash_{cs} \mathcal{E}[e]$$

where $ls = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t\}$.
We have, by assumption,

$$ls \vdash_{cs} \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,]$$

By Lemma 21, the derivation for this judgment includes the following judgment

$$ls_s \vdash_{cs} \rho_\gamma \texttt{sync}\ e$$

where, necessarily, we have $ls_s = \emptyset$ by rule [CS EXP]: by inspection of the reduction rules, e cannot contain an `in-sync` expression.
Again by rule [CS EXP], we have

$$ls_e \vdash_{cs} e$$

where $ls_e = \emptyset$.
Then by Lemma 22, we may conclude with our desired result

$$\begin{array}{l} ls' \vdash_{cs} \mathcal{E}[e] \\ ls' = ls \end{array}$$

- [RED IN-SYNC]. In this case we have

$$\begin{aligned}\sigma(t) &= \mathcal{E}[\texttt{in-sync}\ \rho\ v]\\ \sigma(\rho) &= \{\,\overline{f} = \overline{v}\,\}^t\\ \sigma'(t) &= \mathcal{E}[v]\\ \sigma'(\rho) &= \{\,\overline{f} = \overline{v}\,\}^\perp\end{aligned}$$

The object at ρ changes from being locked by t to being unlocked by any thread. All other objects and all threads other than t do not change; hence it is sufficient to show

$$\mathit{ls} \setminus \{\rho\} \vdash_{cs} \mathcal{E}[v]$$

where $\mathit{ls} = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t\}$.
We have, by assumption,

$$\mathit{ls} \vdash_{cs} \mathcal{E}[\texttt{in-sync}\ \rho\ v]$$

By Lemma 21, the derivation for this judgment includes the following judgment

$$\mathit{ls}_s \vdash_{cs} \texttt{in-sync}\ \rho\ v$$

where, by rule [CS IN-SYNC], we have $\mathit{ls}_s = \{\rho\}$.
Also, we know the following, by rule [CS EXP]:

$$\begin{aligned}&\mathit{ls}_v \vdash_{cs} v\\ &\mathit{ls}_v = \emptyset\end{aligned}$$

Then by Lemma 22, we may conclude with our desired result

$$\begin{aligned}&\mathit{ls}' \vdash_{cs} \mathcal{E}[v]\\ &\mathit{ls}' = \mathit{ls} \setminus \{\rho\}\end{aligned}$$

- [RED FORK]. In this case we have:

$$\begin{aligned}\sigma(t) &= \mathcal{E}[\texttt{fork}\ e]\\ \sigma'(t) &= \mathcal{E}[\texttt{null}]\\ \sigma'(t') &= \textbf{ready}\ e\end{aligned}$$

where $t' \notin \mathit{Dom}(\sigma)$.
Here, a new thread t' is introduced into the state σ'. All objects and all threads other than t and t' do not change; hence it is sufficient to show

$$\begin{aligned}&\mathit{ls} \vdash_{cs} \mathcal{E}[\texttt{null}]\\ &\emptyset \vdash_{cs} \textbf{ready}\ \ e\end{aligned}$$

where $\mathit{ls} = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t\}$.
We have, by assumption,

$$\mathit{ls} \vdash_{cs} \mathcal{E}[\texttt{fork}\ e]$$

By Lemma 21, the derivation for this judgment includes the following judgment

$$\mathit{ls}_f \vdash_{cs} \texttt{fork}\ e$$

where, necessarily, we have $\mathit{ls}_f = \emptyset$ by rule [CS EXP]: inspection of the typing rules indicates that e cannot contain a runtime `in-sync` sub-expression.
Also, we know the following, by rule [CS EXP]:

$$\begin{aligned}&\mathit{ls}_v \vdash_{cs} \texttt{null}\\ &\mathit{ls}_v = \emptyset\end{aligned}$$

Then by Lemma 22, and by rule [CS READY], we may conclude with our desired result

$$\begin{aligned}&\mathit{ls}' \vdash_{cs} \mathcal{E}[\texttt{null}]\\ &\mathit{ls}' = \mathit{ls}\\ &\emptyset \vdash_{cs} \textbf{ready}\ \ e\end{aligned}$$

- [RED READY]. In this case we have

$$\begin{aligned}\sigma(t) &= \textbf{ready}\ e\\ \sigma'(t) &= [e]\end{aligned}$$

All objects and all threads other than t do not change; hence it is sufficient to show

$$\mathit{ls} \vdash_{cs} [e]$$

where $\mathit{ls} = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t\}$.
By inspection of the reduction rules, we observe that no lock is held by t in σ: a **ready** expression is created only with rule [RED FORK] with no locks held, and no thread manipulates another thread's lock. Also, by inspection of the typing rules, we observe that a **ready** expression may not contain a runtime `in-sync` sub-expression.
Thus we have, by assumption,

$$\emptyset \vdash_{cs} \textbf{ready}\ e$$

This judgment may be derived only through rule [CS READY]; hence we obtain our desired result:

$$\emptyset \vdash_{cs} e$$

- [RED LET]. In this case we have:

$$\begin{aligned}\sigma(t) &= \mathcal{E}[\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e]\\ \sigma'(t) &= \mathcal{E}[e[x := v]]\end{aligned}$$

All objects and all threads other than t do not change; hence it is sufficient to show

$$\mathit{ls} \vdash_{cs} \mathcal{E}[e[x := v]]$$

where $\mathit{ls} = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t\}$.
We have, by assumption,

$$\mathit{ls} \vdash_{cs} \mathcal{E}[\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e]$$

By Lemma 21, the derivation for this judgment includes the following judgment

$$\mathit{ls}_\ell \vdash_{cs} \texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e$$

where, necessarily, we have $\mathit{ls}_\ell = \emptyset$ by rule [CS EXP]: inspection of the reduction rules indicates that e cannot contain a runtime `in-sync` sub-expression.
Also, the substitution $[x\ :=\ v]$ performed does not replace any part of e with an `in-sync` expression. Hence we know the following, by rule [CS EXP]:

$$\emptyset \vdash_{cs} e[x := v]$$

Then by Lemma 22, we may conclude with our desired result

$$\begin{aligned}&\mathit{ls}' \vdash_{cs} \mathcal{E}[e[x := v]]\\ &\mathit{ls}' = \mathit{ls}\end{aligned}$$

- [RED IF-NONNULL]. Similar to previous argument.
- [RED IF-NULL]. Similar to previous argument.
- [RED WHILE]. Similar to previous argument.
- [RED WRONG]. In this case we have

$$\begin{aligned}\sigma(t) &= \mathcal{E}[e]\\ \sigma'(t) &= \textbf{wrong}\end{aligned}$$

All objects and all threads other than t do not change; hence it is sufficient to show

$$\mathit{ls} \vdash_{cs} \textbf{wrong}$$

where $\mathit{ls} = \{\rho \mid \sigma(\rho) = \{\,\overline{f} = \overline{v}\,\}^t\}$.
Immediately by rule [CS WRONG], we obtain our desired result:

$$\mathit{ls} \vdash_{cs} \textbf{wrong}$$

B.3 Reduction Theorem

We prove that for a well-typed program, preemptive and cooperative semantics coincide: if a program, starting from a yielding state, can reach a yielding state σ under $\rightarrow$, then it can reach σ under $\rightarrow_c$ as well.

We depend on the following reduction theorem for this result. This theorem is stated in terms of an arbitrary transition system, and requires some additional notation: For any state predicate $S \subseteq \mathit{State}$ and transition relation $T \subseteq \mathit{State} \times \mathit{State}$, the *left restriction* of T to S, or S/T, is the transition relation T with each pair's first component contained in S. Similarly, the *right restriction* of T to S, or $T \backslash S$, is the transition relation T with each pair's second component contained in S. The *composition* of two transition relations T and U, or $T \circ U$, is the set of all transitions (p, r) such that there exists a state q with transitions $(p, q) \in T$ and $(q, r) \in U$. For transition relations T and U, we say that T *right-commutes* with U, and U *left-commutes* with T, if we have $T \circ U \subseteq U \circ T$.

Each thread executes in a series of transactions, each of which consists of a sequence of right-movers, followed by at most one non-mover, followed by a sequence of left-movers. For each thread i, we describe four predicates that partition the set of states:

- $\mathcal{N}_i$ is the set of states where the thread i is not in any transaction;
- $\mathcal{R}_i$ is the set of states where the thread i is in the right-mover part of some transaction;
- $\mathcal{L}_i$ is the set of states where the thread i is in the left-mover part of some transaction; and
- $\mathcal{W}_i$ is the set of states where the thread i has gone wrong.

The reduction theorem relates the following three transition relations:

- $\hookrightarrow_i$ is the transition relation that describes the behavior of the thread i.
- $\hookrightarrow$ is the transition relation where, at each state, one may choose a thread i and use the transition $\hookrightarrow_i$.
- $\hookrightarrow_c$ is the transition relation that describes the serial behavior of a program, in which at most one thread may be in a transaction at any time.

THEOREM 3 (Reduction Theorem). *For all threads i, let $\mathcal{R}_i$, $\mathcal{L}_i$, and $\mathcal{W}_i$ be sets of states, and $\hookrightarrow_i$ be a transition relation. Suppose for all i that the following is true:*

1. *$\mathcal{R}_i$, $\mathcal{L}_i$, and $\mathcal{W}_i$ are pairwise disjoint,*
2. *$(\mathcal{L}_i / \hookrightarrow_i \backslash \mathcal{R}_i)$ is false,*
3. *$\mathcal{W}_i / \hookrightarrow_i$ is false,*

and for all $j \neq i$,

4. *$\hookrightarrow_i$ and $\hookrightarrow_j$ are disjoint,*
5. *$(\hookrightarrow_i \backslash \mathcal{R}_i)$ right-commutes with $\hookrightarrow_j$,*
6. *$(\mathcal{L}_i / \hookrightarrow_i)$ left-commutes with $\hookrightarrow_j$, and*
7. *if $p \hookrightarrow_i q$, then $\mathcal{R}_j(p) \Leftrightarrow \mathcal{R}_j(q)$, $\mathcal{L}_j(p) \Leftrightarrow \mathcal{L}_j(q)$, and $\mathcal{W}_j(p) \Leftrightarrow \mathcal{W}_j(q)$.*

We define $\mathcal{N}_i$, $\mathcal{N}$, $\mathcal{W}$, $\hookrightarrow$, and $\hookrightarrow_c$ as follows:

- $\mathcal{N}_i = \neg(\mathcal{R}_i \vee \mathcal{L}_i)$
- $\mathcal{N} = \forall i \,.\, \mathcal{N}_i$
- $\mathcal{W} = \exists i \,.\, \mathcal{W}_i$
- $\hookrightarrow = \exists i \,.\, \hookrightarrow_i$
- $\hookrightarrow_c = \exists i \,.\, ((\forall j \neq i \,.\, \mathcal{N}_j) / \hookrightarrow_i)$

Now suppose $p \in \mathcal{N}$ and $p \hookrightarrow^* q$. Then the following statements are true.

1. *If $q \in \mathcal{N}$, then $p \hookrightarrow_c^* q$.*
2. *If $q \in \mathcal{W}$ and $\forall i \,.\, q \notin \mathcal{L}_i$, then $p \hookrightarrow_c^* q'$ and $q' \in \mathcal{W}$.*

Proof Refer to [23].

We now turn our attention to the specific case of YIELDJAVA. Consider a fixed program P. Any well-typed state σ with an object map $[\rho_j \mapsto \mathit{obj}_j{}^{j \in 1..m}]$ has a corresponding object environment $E_\sigma = c_1\ \rho_1, \ldots, c_n\ \rho_n$ where $\rho_j \in \mathit{Addr}_{c_j}$.

For an expression e that occurs at an evaluation context position within a thread of σ, we define the function $\alpha :: \mathit{State} \times \mathit{Expr} \rightarrow \mathit{Effect}$

$$\alpha(\sigma, e) = a$$

if we have

$$P; E_\sigma \vdash e : c \cdot a$$

Let WT be the set of well-typed states for P:

$$WT = \{\sigma \mid\ \vdash \sigma\}$$

We define the states:

$$\begin{aligned} N_i &= WT \cap \{\sigma \mid i \notin \mathit{dom}(\sigma) \\ &\qquad\qquad \vee (i \text{ is yielding in } \sigma \wedge \sigma(i) \not\equiv \mathbf{wrong})\} \\ W_i &= WT \cap \{\sigma \mid \sigma(i) \equiv \mathbf{wrong}\} \\ R_i &= WT \cap \{\sigma \mid \alpha(\sigma, \sigma(i)) \not\sqsubseteq \text{CL}\} \setminus N_i \\ L_i &= WT \cap \{\sigma \mid \alpha(\sigma, \sigma(i)) \sqsubseteq \text{CL}\} \setminus N_i \end{aligned}$$

Then we define the auxiliary states N and W:

- $N = \bigcap_{i \in \mathbb{N}} N_i$. That is, N is the set of well-typed states that are yielding.
- $W = \bigcup_{i \in \mathbb{N}} W_i$. That is, W is the set of well-typed states for which some thread is $\mathbf{wrong}$.

We increase the precision of conditional effects by the function $Y :: \mathit{EvalCtxt} \times \mathit{Expr} \rightarrow \mathit{Effect}$, defined as follows:

$$\begin{array}{rcll} Y(\mathcal{E}, \kappa) &=& \kappa & \\ Y(\mathcal{E}, \rho\,?\,a_1 : a_2) &=& Y(\mathcal{E}, a_1) & \text{if } \rho \in \mathit{locks}(\mathcal{E}) \\ Y(\mathcal{E}, \rho\,?\,a_1 : a_2) &=& \rho\,?\,Y(\mathcal{E}, a_1) : Y(\mathcal{E}, a_2) & \text{otherwise} \end{array}$$

That is, we take lockset information from the evaluation context and simplify the conditional effect under that lockset.

THEOREM 4 (Cooperability). *Let σ be a state such that $\vdash \sigma$. Suppose $\sigma \in N$, and there exists state σ' such that $\sigma \rightarrow^* \sigma'$. Then the following statements are true:*

1. *If $\sigma' \in N$, then $\sigma \rightarrow_c^* \sigma'$.*
2. *If $\sigma' \in W$ and $\forall i \,.\, \sigma' \notin L_i$, then there exists a state σ'' such that $\sigma \rightarrow_c^* \sigma''$ and $\sigma'' \in W$.*

Proof We show that for all threads i, the seven preconditions for instantiating the Reduction Theorem hold:

1. R_i, L_i, and W_i are pairwise disjoint,
2. $L_i / \rightarrow_i \backslash R_i$ is false,
3. $W_i / \rightarrow_i$ is false,
4. $\rightarrow_i$ and $\rightarrow_j$ are disjoint for all $j \neq i$,
5. $(\rightarrow_i \backslash R_i)$ right-commutes with $\rightarrow_j$ for all $j \neq i$,
6. $(L_i / \rightarrow_i)$ left-commutes with $\rightarrow_j$ for all $j \neq i$,
7. if $p \rightarrow_i q$, then $R_j(p) \Leftrightarrow R_j(q)$, $L_j(p) \Leftrightarrow L_j(q)$, and $W_j(p) \Leftrightarrow W_j(q)$, for all $j \neq i$.

We apply the Reduction Theorem by substituting the following:

- the set W_i for $\mathcal{W}_i$,
- the set R_i for $\mathcal{R}_i$,
- the set L_i for $\mathcal{L}_i$,
- the relation $\rightarrow_i$ for $\hookrightarrow_i$,
- the relation $\rightarrow_c$ for $\hookrightarrow_c$,
- the state σ for p,
- the state σ' for q.

1. R_i, L_i, and W_i are pairwise disjoint.
   Proof: Inspection of definitions.
2. $L_i / \rightarrow_i \backslash R_i$ is false.
   Proof: Case analysis on $\sigma \rightarrow_i \sigma'$.
   (a) Cases [RED NEW], [RED READ], [RED WRITE], [RED FORK], and [RED IN-SYNC]: We assume that $\sigma'(i) \in R_i$ and thus $\alpha(\sigma', \sigma'(i)) \not\sqsubseteq \mathrm{CL}$. For some well-formed state σ_0,
   - [RED NEW]: We have

     $$\begin{array}{rcl}\sigma & = & \sigma_0[i \mapsto \mathcal{E}[\texttt{new}\, c\texttt{(}\overline{v}\texttt{)}]] \\ \sigma' & = & \sigma_0[i \mapsto \mathcal{E}[\rho], \rho \mapsto \{\, \overline{f} = \overline{v}\,\}^{\perp}]\end{array}$$

     where $\rho \notin \mathit{Dom}(\sigma)$ and $\rho \in \mathit{Addr}_c$ and $\texttt{class}\; c\, \texttt{\{}\, \overline{d}\; \overline{f} \cdots \texttt{\}} \in P$.
   - [RED READ]: We have

     $$\begin{array}{rcl}\sigma & = & \sigma_0[i \mapsto \mathcal{E}[\rho_\gamma f]] \\ \sigma' & = & \sigma_0[i \mapsto \mathcal{E}[v]]\end{array}$$

     where $\sigma(\rho.f) = v$.
   - [RED WRITE]: We have

     $$\begin{array}{rcl}\sigma & = & \sigma_0[i \mapsto \mathcal{E}[\rho_\gamma f \;\texttt{=}\; v]] \\ \sigma' & = & \sigma_0[i \mapsto \mathcal{E}[v], \rho.f \mapsto v]\end{array}$$

   - [RED FORK]: We have

     $$\begin{array}{rcl}\sigma & = & \sigma_0[i \mapsto \mathcal{E}[\texttt{fork}\, e]] \\ \sigma' & = & \sigma_0[i \mapsto \mathcal{E}[\texttt{null}], i' \mapsto \textbf{ready}\;\; e]\end{array}$$

     where $i' \notin \mathit{Dom}(\sigma)$.
   - [RED IN-SYNC]: We have

     $$\begin{array}{rcl}\sigma & = & \sigma_0[i \mapsto \mathcal{E}[\texttt{in-sync}\, \rho\, v], \rho \mapsto \{\, \overline{f} = \overline{v}\,\}^{i}] \\ \sigma' & = & \sigma_0[i \mapsto \mathcal{E}[v], \rho \mapsto \{\, \overline{f} = \overline{v}\,\}^{\perp}]\end{array}$$

   For cases [RED READ] and [RED WRITE], $\gamma = .$ since $\sigma \in L_i$, which excludes any yield by thread i. Let $\sigma(i) = \mathcal{E}[e]$ for each case, where e is some redex e, and $\sigma'(i) = \mathcal{E}[v]$. Then we have $\alpha(\sigma, \mathcal{E}[e]) \sqsubseteq \mathrm{CL}$ and $\alpha(\sigma', \mathcal{E}[v]) \not\sqsubseteq \mathrm{CL}$. Since

   $$\alpha(\sigma', v) \sqsubseteq \alpha(\sigma, e)$$

   we obtain, by Lemma 4,

   $$\alpha(\sigma', \mathcal{E}[v]) \sqsubseteq \alpha(\sigma, \mathcal{E}[e]) \sqsubseteq \mathrm{CL}$$

   which contradicts our assumption of $\sigma'(i) \in R_i$.

   (b) [RED INVOKE]: For some well-formed state σ_0, we have

   $$\begin{array}{rcl}\sigma & = & \sigma_0[i \mapsto \mathcal{E}[\rho_\gamma m\texttt{(}\overline{v}\texttt{)}]] \\ \sigma' & = & \sigma_0[i \mapsto \mathcal{E}[e[\texttt{this} := \rho, \overline{x} := \overline{v}]]]\end{array}$$

   where

   $$\begin{array}{l}\rho \in \mathit{Addr}_{d'} \\ \texttt{class}\; d'\; \texttt{\{}\; \cdots a'\; c\, m\texttt{(}\overline{d}\; \overline{x}\texttt{)\{}\, e\, \texttt{\}} \cdots \texttt{\}} \in P\end{array}$$

   Here, $\gamma = .$ since $\mathcal{L}_i$ excludes yielding thread states for i. We show that

   $$\alpha(\sigma', e[\texttt{this} := \rho, \overline{x} := \overline{v}]) \sqsubseteq \alpha(\sigma, \rho.m\texttt{(}\overline{v}\texttt{)})$$

   and use Lemma 4 to obtain a contradiction. From $\alpha(\sigma, \rho.m\texttt{(}\overline{v}\texttt{)})$ being well-defined, we know the following judgment must hold for the well-formed environment E_σ:

   $$P; E_\sigma \vdash \rho.m\texttt{(}\overline{v}\texttt{)}\, \texttt{:}\, c \cdot a$$

   and from the antecedents of this judgment we know that

   $$P; E_\sigma \vdash a'[\texttt{this} := \rho, \overline{x} := \overline{v}] \uparrow a$$

   where a' is the annotation for method m. Hence, by definition of $\uparrow$, we know that

   $$a'[\texttt{this} := \rho, \overline{x} := \overline{v}] \sqsubseteq a \tag{1}$$

   Since P is well-typed, m is well-typed:

   $$P; c\, \texttt{this} \vdash a'\; c\, m\texttt{(}\overline{d}\; \overline{x}\texttt{)\{}\, e\, \texttt{\}}$$

   and by inversion on this judgment, we know

   $$P; c\, \texttt{this}, \overline{d}\; \overline{x} \vdash e\, \texttt{:}\, c \cdot a''$$

   such that

   $$a'' \sqsubseteq a' \tag{2}$$

   By the substitution lemma, we have

   $$P; \emptyset \vdash e[\texttt{this} := \rho, \overline{x} := \overline{v}]\, \texttt{:}\, c \cdot \big(a''[\texttt{this} := \rho, \overline{x} := \overline{v}]\big)$$

   Thus we have the equality

   $$\alpha(\sigma', e[\texttt{this} := \rho, \overline{x} := \overline{v}]) = a''[\texttt{this} := \rho, \overline{x} := \overline{v}]$$

   By Lemma 1 and Formula 2,

   $$a''[\texttt{this} := \rho, \overline{x} := \overline{v}] \sqsubseteq a'[\texttt{this} := \rho, \overline{x} := \overline{v}] \tag{3}$$

   The combination of Formulas 3 and 1 gives us our desired result

   $$a''[\texttt{this} := \rho, \overline{x} := \overline{v}] \sqsubseteq a$$

   Applying Lemma 4, we get

   $$\alpha(\sigma', \mathcal{E}[e[\texttt{this} := \rho, \overline{x} := \overline{v}]]) \sqsubseteq \alpha(\sigma, \mathcal{E}[\rho.m\texttt{(}\overline{v}\texttt{)}]) \tag{4}$$

   And by assumption of $\sigma \in L_i$, we get

   $$\alpha(\sigma, \mathcal{E}[\rho.m\texttt{(}\overline{v}\texttt{)}]) \sqsubseteq \mathrm{CL} \tag{5}$$

   We have obtained a contradiction: by assumption of $\sigma' \in R_i$,

   $$\alpha(\sigma', \mathcal{E}[e[\texttt{this} := \rho, \overline{x} := \overline{v}]]) \not\sqsubseteq \mathrm{CL}$$

   but by Formulas 4 and 5, we have

   $$\alpha(\sigma', \mathcal{E}[e[\texttt{this} := \rho, \overline{x} := \overline{v}]]) \sqsubseteq \mathrm{CL}$$

   Hence this case is trivially true.

   (c) [RED INVOKE COMPOUND]: Similar to [RED INVOKE].

   (d) [RED SYNC]: For some well-formed state σ_0, we have

   $$\begin{array}{rcl}\sigma & = & \sigma_0[i \mapsto \mathcal{E}[\rho_\gamma \texttt{sync}\; e\,], \quad \rho \mapsto \{\, \overline{f} = \overline{v}\,\}^{\perp}] \\ \sigma' & = & \sigma_0[i \mapsto \mathcal{E}[\texttt{in-sync}\, \rho\, e], \rho \mapsto \{\, \overline{f} = \overline{v}\,\}^{i}]\end{array}$$

   We have $\gamma = .$ since $\sigma \in L_i$ excludes yielding thread states for i. By $\mathcal{S}$ and $\mathcal{SI}$, we have

   $$\begin{array}{c}\alpha(\sigma, \rho.\texttt{sync}\; e\,) = \mathcal{S}(\rho, ., \alpha(\sigma, e)) \\ \alpha(\sigma', \texttt{in-sync}\, \rho\, e) = \mathcal{SI}(\rho, \alpha(\sigma', e))\end{array}$$

   The thread i may only transition from $\mathcal{E}[\rho_\gamma \texttt{sync}\; e\,]$ to $\mathcal{E}[\texttt{in-sync}\; \rho\; e]$ if i does *not* hold lock ρ. Hence we may apply Lemma 14 and get the subeffect relation

   $$\alpha(\sigma', \texttt{in-sync}\, \rho\, e) \sqsubseteq_{\{\rho\}} \alpha(\sigma, \rho.\texttt{sync}\; e\,)$$

   We use Lemma 5 to obtain the subeffect relation

   $$\alpha(\sigma', \mathcal{E}[\texttt{in-sync}\, \rho\, e]) \sqsubseteq_{\{\rho\}} \alpha(\sigma, \mathcal{E}[\rho.\texttt{sync}\; e\,])$$

   But this is a contradiction: By assumption of $\sigma \in L_i$, we know

   $$\alpha(\sigma, \mathcal{E}[\rho.\texttt{sync}\; e\,]) \sqsubseteq \mathrm{CL}$$

   and so

   $$\alpha(\sigma', \mathcal{E}[\texttt{in-sync}\, \rho\, e]) \sqsubseteq_{\{\rho\}} \mathrm{CL}$$

But also $\sigma' \in R_i$ by assumption, and we have

$$\alpha(\sigma', \mathcal{E}[\mathtt{in\text{-}sync}\ \rho\, e]) \not\sqsubseteq \textsc{cl}$$

Hence this case is trivially true.

(e) [RED SYNC REENTRANT]: For some well-formed state σ_0, we have

$$\begin{array}{rcll} \sigma & = & \sigma_0[i \mapsto \mathcal{E}[\rho_\gamma \mathtt{sync}\ e\,], & \rho \mapsto \{\,\overline{f} = \overline{v}\,\}^i] \\ \sigma' & = & \sigma_0[i \mapsto \mathcal{E}[e], & \rho \mapsto \{\,\overline{f} = \overline{v}\,\}^i] \end{array}$$

We have $\gamma = .$ since $\sigma \in L_i$ and thus non-yielding. Thread i may transition from $\mathcal{E}[\rho.\mathtt{sync}\ e\,]$ to $\mathcal{E}[e]$ only if ρ is held by i. Hence we have $\rho \in \mathit{locks}(\mathcal{E})$. By Lemma 15, we have

$$\alpha(\sigma', e) \sqsubseteq^{\{\rho\}} \mathcal{S}(\rho, .., \alpha(\sigma, e))$$

This implies

$$\textsc{af}; \alpha(\sigma', e) \sqsubseteq_{\emptyset}^{\emptyset \cup \mathit{locks}(\mathcal{E})} \mathcal{S}(\rho, .., \alpha(\sigma, e))$$

Applying Lemma 5, we obtain

$$\textsc{af}; \alpha(\sigma', \mathcal{E}[e]) \sqsubseteq_{\emptyset}^{\emptyset} \alpha(\sigma, \mathcal{E}[\rho.\mathtt{sync}\ e\,])$$

which is equivalent to

$$\alpha(\sigma', \mathcal{E}[e]) \sqsubseteq \alpha(\sigma, \mathcal{E}[\rho.\mathtt{sync}\ e\,])$$

This is a contradiction: By assumption of $\sigma \in L_i$, we know

$$\alpha(\sigma, \mathcal{E}[\rho.\mathtt{sync}\ e\,]) \sqsubseteq \textsc{cl}$$

and so

$$\alpha(\sigma', \mathcal{E}[e]) \sqsubseteq \textsc{cl}$$

But also $\sigma' \in R_i$ by assumption, and we have

$$\alpha(\sigma', \mathcal{E}[e]) \not\sqsubseteq \textsc{cl}$$

Hence this case is trivially true.

(f) [RED READY]: $\sigma(i) = \mathcal{E}[\mathbf{ready}\ e]$ and is yielding, thus we ignore this case.

(g) [RED LET]: For some well-formed state σ_0, we have

$$\begin{array}{rcl} \sigma & = & \sigma_0[i \mapsto \mathcal{E}[\mathtt{let}\ x\ \mathtt{=}\ v\ \mathtt{in}\, e]] \\ \sigma' & = & \sigma_0[i \mapsto \mathcal{E}[e[x := v]]] \end{array}$$

We show that

$$\alpha(\sigma', e[x := v]) \sqsubseteq \alpha(\sigma, \mathtt{let}\ x\ \mathtt{=}\ v\ \mathtt{in}\, e)$$

and use Lemma 4 to obtain a contradiction. Because $\alpha(\sigma, \mathtt{let}\ x\ \mathtt{=}\ v\ \mathtt{in}\, e)$ is well-defined, we know the following judgment must hold for the well-formed environment E_σ:

$$P; E_\sigma \vdash \mathtt{let}\ x\ \mathtt{=}\ v\ \mathtt{in}\, e\ \mathbf{:}\ c \cdot a$$

and the following antecedents in this judgment must also be true:

$$\begin{array}{l} P; E_\sigma \vdash v\ \mathbf{:}\ \textsc{af} \cdot c' \\ P; E_\sigma, c'\ x \vdash e\ \mathbf{:}\ c \cdot a' \\ P; E_\sigma \vdash a'[x := v] \uparrow a \end{array}$$

Applying the substitution lemma on the second antecedent, we get

$$P; E_\sigma \vdash e[x := v]\ \mathbf{:}\ c \cdot a'[x := v]$$

By definition of $\uparrow$, we have

$$a'[x := v] \sqsubseteq a$$

(h) [RED IF-NONNULL] and [RED IF-NULL]: For some well-formed state σ_0, we have

$$\begin{array}{rcll} \sigma & = & \sigma_0[i \mapsto \mathcal{E}[\mathtt{if}\ v\ e_2\ e_3]] & \\ \sigma' & = & \sigma_0[i \mapsto \mathcal{E}[e_j]] & \text{for } j = 2..3 \end{array}$$

and by assumption, $\sigma \in L_i$ and $\sigma' \in R_i$. Then by definition of $\sqcup$, for $j = 2..3$,

$$\alpha(\sigma', e_j) \sqsubseteq \alpha(\sigma, \mathtt{if}\ v\ e_2\ e_3) = (\alpha(\sigma, e_2) \sqcup \alpha(\sigma, e_3))$$

By Lemma 4, we obtain a contradiction for $\sigma'(i)$.

(i) [RED WRONG]: Since $\sigma'(i) = \mathbf{wrong}$, we have $\sigma' \notin \mathcal{R}_i$.

3. $W_i / \rightarrow_i$ is false.
Proof: By definition of $W_i = WT \cap \{\sigma \mid \sigma(i) \equiv \mathbf{wrong}\}$ and inspection of transition rules, we see that the thread i cannot advance.

4. $\rightarrow_i$ and $\rightarrow_j$ are disjoint for all $j \neq i$. That is, no states p, q exist such that $p \rightarrow_i q$ and $p \rightarrow_j q$.
Proof: Observe that the transition relation $\rightarrow_i$ changes the thread i, and leaves all other existing threads unchanged. Since $j \neq i$, when starting from some state p, we have $p \rightarrow_i q$ and $p \rightarrow_j q'$ and $q \neq q'$.

5. $(\rightarrow_i \backslash R_i)$ right-commutes with $\rightarrow_j$ for all $j \neq i$.
Proof: We proceed by case analysis on $p_1 \rightarrow_i p_2$, assuming $p_2 \in R_i$, and $p_2 \rightarrow_j p_3$. In the arguments that follow, we assume a well-formed state σ.

- [RED READ]: Let $p_1 = \sigma[i \mapsto \mathcal{E}[\rho_\gamma f]]$ and $p_2 = \sigma[i \mapsto \mathcal{E}[v]]$, where $\sigma(\rho.f) = v$. We proceed by case analysis on γ and field type.

  (a) $\gamma = .$ and $f \in \mathit{Normal}$: In this case, no other thread reads or writes to $\rho.f$. Thus $\rightarrow_i \backslash R_i$ and $\rightarrow_j$ commute, since they operate on disjoint data.

  (b) $\gamma = .$ and $f \in \mathit{Final}$: In this case, other threads may only read $\rho.f$. Thus $\rightarrow_i \backslash R_i$ and $\rightarrow_j$ commute, since reads of $\rho.f$ by threads i and j commute.

  (c) $\gamma = .$ and $f \in \mathit{Volatile}$: By Lemma 3, we have

  $$Y(\mathcal{E}, \alpha(p_1, \rho.f)); \alpha(p_1, \mathcal{E}[v]) \sqsubseteq \alpha(p_1, \mathcal{E}[\rho.f])$$

  This is a contradiction: we have

  $$Y(\mathcal{E}, \alpha(p_1, \rho.f)) = \alpha(p_1, \rho.f) = \textsc{an}$$

  and $\textsc{an}$ sequentially composed with $\alpha(p_1, \mathcal{E}[v]) = \alpha(p_2, \mathcal{E}[v]) \not\sqsubseteq \textsc{cl}$ is undefined.

  (d) $\gamma = ..$ and $f \in \mathit{Volatile}$: By Lemma 3, we have

  $$Y(\mathcal{E}, \alpha(p_1, \rho..f)); \alpha(p_1, \mathcal{E}[v]) \sqsubseteq \alpha(p_1, \mathcal{E}[\rho..f])$$

  By rule [EXP REF RACE], we have

  $$Y(\mathcal{E}, \alpha(p_1, \rho..f)) = \alpha(p_1, \rho..f) = (\textsc{cy}; \textsc{an}) = \textsc{cl}$$

  while by assumption we have

  $$\alpha(p_1, \mathcal{E}[v]) = \alpha(p_2, \mathcal{E}[v]) \not\sqsubseteq \textsc{cl}$$

  This is a contradiction, since this sequential composition is undefined.

- [RED WRITE]: Let $p_1 = \sigma[i \mapsto \mathcal{E}[\rho_\gamma f\ \mathtt{=}\ v]]$ and $p_2 = \sigma[i \mapsto \mathcal{E}[v]]$. We proceed by case analysis on γ and field type.

  (a) $\gamma = .$ and $f \in \mathit{Normal}$: In this case, the threads i and j operate on disjoint data – hence the operations commute.

  (b) $\gamma = .$ and $f \in \mathit{Volatile}$: By Lemma 3, we have

  $$\begin{array}{rl} & Y(\mathcal{E}, \alpha(p_1, \rho.f\ \mathtt{=}\ v)); \alpha(p_1, \mathcal{E}[v]) \\ \sqsubseteq & \alpha(p_1, \mathcal{E}[\rho.f\ \mathtt{=}\ v]) \end{array}$$

  This is a contradiction, since

  $$Y(\mathcal{E}, \alpha(p_1, \rho.f\ \mathtt{=}\ v)) = \alpha(p_1, \rho.f\ \mathtt{=}\ v) = \textsc{an}$$

  and $\textsc{an}$ sequentially composed with $\alpha(p_1, \mathcal{E}[v]) = \alpha(p_1, \mathcal{E}[v]) \not\sqsubseteq \textsc{cl}$ is undefined.

(c) $\gamma = ..$ and $f \in \mathit{Volatile}$: By Lemma 3, we have

$$\begin{array}{rl} & Y(\mathcal{E}, \alpha(p_1, \rho..f \texttt{ = } v)); \alpha(p_1, \mathcal{E}[v]) \\ \sqsubseteq & \alpha(p_1, \mathcal{E}[\rho..f \texttt{ = } v]) \end{array}$$

By rule [EXP ASSIGN VOLATILE], we have

$$Y(\mathcal{E}, \alpha(p_1, \rho..f \texttt{ = } v)) = \alpha(p_1, \rho..f \texttt{ = } v) = \text{CL}$$

while by assumption we have

$$\alpha(p_1, \mathcal{E}[v]) = \alpha(p_2, \mathcal{E}[v]) \not\sqsubseteq \text{CL}$$

This is a contradiction, since this sequential composition is undefined.

- [RED NEW]: Suppose the step $\to_i$ creates a new object at address ρ. Then observe that the step $\to_j$ cannot access ρ, since thread j must be well-typed in an environment that does not contain ρ. Thus, $\to_i \backslash R_i$ changes only thread i and ρ, and $\to_j$ changes only the thread j and maybe some other address $\rho' \neq \rho$. So $(\to_i \backslash R_i)$ and $\to_j$ commute.
- [RED SYNC]: Let $p_1 = \sigma[i \mapsto \mathcal{E}[\rho.\texttt{sync}\ e]]$ and $p_2 = \sigma[i \mapsto \mathcal{E}[\texttt{in-sync}\ \rho\ v]]$. To interfere with $\to_i$, the step $\to_j$ would need to obtain the lock ρ. By inspection, observe that only rules [RED SYNC], [RED SYNC REENTRANT], and [RED IN-SYNC] hold ρ, all three of which cannot occur.
- [RED SYNC REENTRANT]: Similar to previous case.
- [RED IN-SYNC]: Let $p_1 = \sigma[i \mapsto \mathcal{E}[\texttt{in-sync}\ \rho\ v]]$ and $p_2 = \sigma[i \mapsto \mathcal{E}[v]]$. By Lemma 3, we have

  $$\begin{array}{rl} & Y(\mathcal{E}, \alpha(p_1, \texttt{in-sync}\ \rho\ v)); \alpha(p_1, \mathcal{E}[v]) \\ \sqsubseteq & \alpha(p_1, \mathcal{E}[\texttt{in-sync}\ \rho\ v]) \end{array}$$

  The function $\mathcal{SI}$ tells us that

  $$\begin{array}{rl} & Y(\mathcal{E}, \alpha(p_1, \texttt{in-sync}\ \rho\ v)) \\ = & \alpha(p_1, \texttt{in-sync}\ \rho\ v) \\ = & \mathcal{SI}(\rho, \text{AF}) \\ = & \text{AF}; \text{AL} = \text{AL} \end{array}$$

  However, $\alpha(p_1, \mathcal{E}[v]) = \alpha(p_2, \mathcal{E}[v]) \not\sqsubseteq \text{CL}$. Hence the sequential composition is undefined, a contradiction.
- [RED FORK]: Let $p_1 = \sigma[i \mapsto \mathcal{E}[\texttt{fork}\ e]]$ and $p_2 = \sigma[i \mapsto \mathcal{E}[\texttt{null}]]$. By Lemma 3, we have

  $$Y(\mathcal{E}, \alpha(p_1, \texttt{fork}\ e)); \alpha(p_1, \mathcal{E}[\texttt{null}]) \sqsubseteq \alpha(p_1, \mathcal{E}[\texttt{fork}\ e])$$

  The rule [EXP FORK] tells us that

  $$Y(\mathcal{E}, \alpha(p_1, \texttt{fork}\ e)) = \alpha(p_1, \texttt{fork}\ e) = \text{AL}$$

  However, $\alpha(p_1, \mathcal{E}[\texttt{null}]) = \alpha(p_2, \mathcal{E}[\texttt{null}]) \not\sqsubseteq \text{CL}$. Hence the sequential composition is undefined, a contradiction.
- [RED INVOKE], [RED INVOKE COMPOUND], [RED LET], [RED IF-NONNULL], [RED IF-NULL], [RED WHILE], [RED READY], [RED WRONG]: Since the operation by thread i is entirely local, it commutes with the operation of any other thread j.

6. $(L_i / \to_i)$ left-commutes with $\to_j$ for all $j \neq i$.

   Proof: We proceed by case analysis on $p_2 \to_i p_3$, assuming $p_2 \in L_i$, and $p_1 \to_j p_2$. In the arguments that follow, we assume a well-formed state σ.

   - [RED READ]: Let $p_2 = \sigma[i \mapsto \mathcal{E}[\rho_\gamma f]]$ and $p_3 = \sigma[i \mapsto \mathcal{E}[v]]$. We proceed by case analysis on γ and field type.

     (a) $\gamma = .$ and $f \in \mathit{Normal}$: In this case, field f is synchronized, and thread j may not read or write to $\rho.f$. Since the two operations access disjoint data, they commute.

     (b) $\gamma = .$ and $f \in \mathit{Final}$: In this case, thread j may read from (but not write to) $\rho.f$. Since two reads commute freely, the two operations commute.

     (c) $\gamma = .$ and $f \in \mathit{Volatile}$: By the Lemma 13, we have

     $$\alpha(p_2, \mathcal{E}[v]) = \alpha(p_3, \mathcal{E}[v])$$

     By Lemma 3, we have

     $$Y(\mathcal{E}, \alpha(p_2, \rho.f)); \alpha(p_2, \mathcal{E}[v]) \sqsubseteq \alpha(p_2, \mathcal{E}[\rho.f])$$

     where $\sigma(\rho.f) = v$. By rule [EXP REF RACE], we have

     $$Y(\mathcal{E}, \alpha(p_2, \rho.f)) = \alpha(p_2, \rho.f) = \text{AN}$$

     Since sequential composition of N with any mover effect is either undefined or $\not\sqsubseteq$ L, this contradicts our assumption of $p_2 \in L_i$.

     (d) $\gamma = ..$ and $f \in \mathit{Volatile}$: Since $\gamma = ..$, we have $p_2 \in N_i$ and also $p_2 \in L_i$: a contradiction.

   - [RED WRITE]: Let $p_2 = \sigma[i \mapsto \mathcal{E}[\rho_\gamma f \texttt{ = } v]]$ and $p_3 = \sigma[i \mapsto \mathcal{E}[v]]$. We proceed by case analysis on γ and field type.

     (a) $\gamma = .$ and $f \in \mathit{Normal}$: In this case, field f is synchronized, and thread j may not read or write to $\rho.f$. Since the two operations access disjoint data, they commute.

     (b) $\gamma = .$ and $f \in \mathit{Volatile}$: By Lemma 13, we have

     $$\alpha(p_2, \mathcal{E}[v]) = \alpha(p_3, \mathcal{E}[v])$$

     By Lemma 3, we have

     $$Y(\mathcal{E}, \alpha(p_2, \rho.f \texttt{ = } v)); \alpha(p_2, \mathcal{E}[v]) \sqsubseteq \alpha(p_2, \mathcal{E}[\rho.f \texttt{ = } v])$$

     By rule [EXP WRITE VOLATILE], we have

     $$Y(\mathcal{E}, \alpha(p_2, \rho.f \texttt{ = } v)) = \alpha(p_2, \rho.f \texttt{ = } v) = \text{AN}$$

     Since sequential composition of N with any mover effect is either undefined or $\not\sqsubseteq$ L, this contradicts our assumption of $p_2 \in L_i$.

     (c) $\gamma = ..$ and $f \in \mathit{Volatile}$: Since $\gamma = ..$, we have $p_2 \in N_i$ and also $p_2 \in L_i$: a contradiction.

     (d) [RED NEW]: Let $p_2 = \sigma[i \mapsto \mathcal{E}[\texttt{new}\ c(\overline{v})]]$ and $p_3 = \sigma[i \mapsto \mathcal{E}[\rho]]$. Observe that the step by thread j cannot refer to ρ and yet $\sigma(j)$ is well-typed. Hence, threads i and j access disjoint data, and the operations commute.

   - [RED SYNC]: Let $p_2 = \sigma[i \mapsto \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,], \rho \mapsto \{\,\overline{f} = \overline{v}\,\}^\perp]$ and $p_3 = \sigma[i \mapsto \mathcal{E}[\texttt{in-sync}\ \rho\ e], \rho \mapsto \{\,\overline{f} = \overline{v}\,\}^i]$. We have $\gamma = .$ since $p_2 \in L_i$. By Lemma 3, we have, for some value v,

     $$\begin{array}{rl} & Y(\mathcal{E}, \alpha(p_2, \rho.\texttt{sync}\ e\,)); \alpha(p_2, \mathcal{E}[v]) \\ \sqsubseteq & \alpha(p_2, \mathcal{E}[\rho.\texttt{sync}\ e\,]) \end{array} \tag{6}$$

     By rule [EXP SYNC] and the definition of function $\mathcal{S}$,

     $$Y(\mathcal{E}, \alpha(p_2, \rho.\texttt{sync}\ e\,)) = Y(\mathcal{E}, \mathcal{S}(\rho, ., \alpha(p_2, e))) \tag{7}$$

     Thread i's transition from $\mathcal{E}[\rho_\gamma \texttt{sync}\ e\,]$ to $\mathcal{E}[\texttt{in-sync}\ \rho\ e]$ may occur only if ρ is *not* held; hence we have $\rho \not\in \mathit{locks}(\mathcal{E})$. Then by Lemma 16,

     $$Y(\mathcal{E}, \mathcal{S}(\rho, ., \alpha(p_2, e))) \not\sqsubseteq \text{CL} \tag{8}$$

     This is a contradiction: By assumption of $p_2 \in L_i$, we have

     $$\alpha(p_2, \mathcal{E}[\rho.\texttt{sync}\ e\,]) \sqsubseteq \text{CL}$$

     but by Formulas 6 and 7 and 8, we get

     $$\alpha(p_2, \mathcal{E}[\rho.\texttt{sync}\ e\,]) \not\sqsubseteq \text{CL}$$

     Hence this case is trivially true.

   - [RED IN-SYNC]: Let $p_2 = \sigma[i \mapsto \mathcal{E}[\texttt{in-sync}\ \rho\ v], \rho \mapsto \{\,\overline{f} = \overline{v}\,\}^i]$ and $p_3 = \sigma[i \mapsto \mathcal{E}[v], \rho \mapsto \{\,\overline{f} = \overline{v}\,\}^\perp]$. The step by thread j cannot be one of [RED SYNC], [RED SYNC REENTRANT], or [RED IN-SYNC] that operate on lock ρ: at p_2, the lock ρ is held by i, which contradicts the lock state if one of these operations was performed. These are the only operations that may interfere with

[RED IN-SYNC], and any other operation by j successfully commutes with $\rightarrow_i$.

- [RED SYNC REENTRANT]: Let $p_2 = \sigma[i \mapsto \mathcal{E}[\rho_\gamma \texttt{sync}\ e\,], \rho \mapsto \{\,\overline{f} = \overline{v}\,\}^i]$ and $p_3 = \sigma[i \mapsto \mathcal{E}[e], \rho \mapsto \{\,\overline{f} = \overline{v}\,\}^i]$. The argument proceeds similar to the previous case.
- [RED FORK]: Let $p_2 = \sigma[i \mapsto \mathcal{E}[\texttt{fork}\ e]]$ and $p_3 = \sigma[i \mapsto \mathcal{E}[\texttt{null}], i' \mapsto \textbf{ready}\ e]$, where $i' \notin \mathit{Dom}(p_2)$. Then p_1 cannot refer to i' and i' is not executing; hence the fork operation can left-commute with $\rightarrow_j$.
- [RED READY]: Since $p_2 = \sigma[i \mapsto \textbf{ready}\ e]$, we have $p_2 \in N_i$: a contradiction.
- [RED INVOKE], [RED INVOKE COMPOUND], [RED LET], [RED IF-NONNULL], [RED IF-NULL], [RED WHILE], [RED WRONG]: If there is a γ in any of these operations, we have $\gamma = .$ since p_2 is non-yielding in L_i. For these operations, all modifications to state are local, and cannot be affected by $\rightarrow_j$. Thus $\rightarrow_j$ and $\rightarrow_i$ commute.

7. If $p \rightarrow_i q$, then $R_j(p) \Leftrightarrow R_j(q)$, $L_j(p) \Leftrightarrow L_j(q)$, and $W_j(p) \Leftrightarrow W_j(q)$, for all $j \neq i$.
   Proof: We proceed by case analysis on the expansion of p's thread map.
   - Assume i does not fork a new thread. By inspection of the transition rules, we see that thread i does not change the thread state of another thread j. Thus $R_j(p) \Leftrightarrow R_j(q)$ and $L_j(p) \Leftrightarrow L_j(q)$ and $W_j(p) \Leftrightarrow W_j(q)$.
   - Assume i forks a new thread k. Since only these two threads have changes, for all threads j other than i and k, we have $R_j(p) \Leftrightarrow R_j(q)$ and $L_j(p) \Leftrightarrow L_j(q)$ and $W_j(p) \Leftrightarrow W_j(q)$. For thread k, we have $N_k(p)$, since $k \notin \mathit{Dom}(p)$, and $N_k(q)$, since the thread state at k is yielding in q. This implies $R_k(p) \Leftrightarrow R_k(q)$ and $L_k(p) \Leftrightarrow L_k(q)$ and $W_k(p) \Leftrightarrow W_k(q)$.

LEMMA 1 (Effect Monotonicity).

1. *If $a_1 \sqsubseteq a_2$, then for all a:*

$$\begin{array}{cc} a_1; a \sqsubseteq a_2; a & a \sqcup a_1 \sqsubseteq a \sqcup a_2 \\ a; a_1 \sqsubseteq a; a_2 & a_1^* \sqsubseteq a_2^* \\ a_1 \sqcup a \sqsubseteq a_2 \sqcup a & \theta(a_1) \sqsubseteq \theta(a_2) \end{array}$$

   where θ is a set of substitutions $[\overline{x} := \overline{e}]$.
2. *If $a_1 \sqsubseteq a_2$ and $P; E \vdash a_1 \uparrow a_1'$ and $P; E \vdash a_2 \uparrow a_2'$ then $a_1' \sqsubseteq a_2'$.*
3. *If $a_1 \sqsubseteq_n^h a_2$, then for any valid lock expression ℓ such that $\ell \notin h$ and $\ell \notin n$, we have*

$$\begin{array}{c} a_1 \sqsubseteq_n^{h \cup \{\ell\}} a_2 \\ a_1 \sqsubseteq_{n \cup \{\ell\}}^h a_2 \end{array}$$

4. *If $a_1 \sqsubseteq_n^h a_2$, then for all a:*

$$\begin{array}{c} a_1; a \sqsubseteq_n^h a_2; a \\ a; a_1 \sqsubseteq_n^h a; a_2 \end{array}$$

Proof Follows from the definitions of these operations.

LEMMA 2 (Context Subexpression). *Suppose there is a deduction that concludes $P; E \vdash \mathcal{E}[e] : c \cdot a$. Then that deduction contains, at a position corresponding to the hole in $\mathcal{E}$, a subdeduction that concludes $P; E \vdash e : c' \cdot a'$.*

Proof Induction over the derivation of $P; E \vdash \mathcal{E}[e] : c \cdot a$.

LEMMA 3 (Sequentiality). *Let $\mathcal{E}$ a context, E a well-formed environment, e an expression, and v a value. Suppose the following:*

$$\begin{array}{l} P; E \vdash \mathcal{E}[e] : c \cdot a \\ P; E \vdash \mathcal{E}[v] : c \cdot a' \\ e \textit{ is not a value} \end{array}$$

Then we may conclude

$$\begin{array}{l} P; E \vdash e : c' \cdot a'' \\ Y(\mathcal{E}, a''); a' \sqsubseteq a \end{array}$$

Proof By induction over the derivation of $P; E \vdash \mathcal{E}[e] : c \cdot a$. First, by Lemma 2, we obtain $P; E \vdash e : c' \cdot a''$.

- $\mathcal{E} \equiv [\,]$: As assumptions, we have

$$\begin{array}{l} P; E \vdash [e] : c \cdot a \\ P; E \vdash [v] : c \cdot a' \end{array}$$

  where $a' = $ AF. From the first assumption, we immediately have $a'' = a$. Also AF is the right identity for sequential composition. Hence we have the equality

$$(a''; a') = (a; AF) = a$$

  By Lemma 17, we know

$$Y(\mathcal{E}, a'') \sqsubseteq a''$$

  so by Lemma 1, we have

$$Y(\mathcal{E}, a''); a' \sqsubseteq a''; a'$$

  and we may conclude with our desired result

$$Y(\mathcal{E}, a''); a' \sqsubseteq a$$

- $\mathcal{E} \equiv \texttt{new}\ c(\overline{v},\ \mathcal{E}',\ \overline{e})$: As assumptions, we have

$$\begin{array}{l} P; E \vdash \texttt{new}\ c(\overline{v},\ \mathcal{E}'[e],\ \overline{e}) : c \cdot a \\ P; E \vdash \texttt{new}\ c(\overline{v},\ \mathcal{E}'[v],\ \overline{e}) : c \cdot a' \end{array}$$

  Let us take $\mathcal{E}'[e]$ to be in the k^{th} position in the argument list. These assumptions are derivable only through rule [EXP NEW]; hence, we know

$$\begin{array}{l} \texttt{class}\ c\,\{\ d_i\ x_i\ ^{i \in 1..n}\} \in P \\ P; E \vdash v_i : c_i \cdot a_i \qquad \forall i \in 1..k-1 \\ P; E \vdash \mathcal{E}'[e] : c_k \cdot a_k \\ P; E \vdash \mathcal{E}'[v] : c_k \cdot a_k' \\ P; E \vdash e_i : c_i \cdot a_i \qquad \forall i \in k+1..n \\ a_i = \mathrm{AF} \qquad \forall i \in 1..k-1 \\ a = (a_1; \ldots; a_k; \ldots; a_n; \mathrm{AM}) \\ a' = (a_1; \ldots; a_k'; \ldots; a_n; \mathrm{AM}) \end{array}$$

  Since AF is identity for sequential composition, we can simplify a and a' as follows:

$$\begin{array}{l} a = (a_k; \ldots; a_n; \mathrm{AM}) \\ a' = (a_k'; \ldots; a_n; \mathrm{AM}) \end{array}$$

  By IH, we have

$$Y(\mathcal{E}', a''); a_k' \sqsubseteq a_k$$

  Since $\mathit{locks}(\mathcal{E}') = \mathit{locks}(\mathcal{E})$, we have

$$Y(\mathcal{E}, a''); a_k' \sqsubseteq a_k$$

  By Lemma 1, we get

$$(\,Y(\mathcal{E}, a''); a_k'; \ldots; a_n; \mathrm{AM}) \sqsubseteq (a_k; \ldots; a_n; \mathrm{AM})$$

  Hence, by substitution, we obtain our desired result

$$Y(\mathcal{E}, a''); a' \sqsubseteq a$$

- $\mathcal{E} \equiv \mathcal{E}'\,_\gamma f$: As assumptions, we have

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e]_\gamma f : c \cdot a \\ P; E \vdash \mathcal{E}'[v]_\gamma f : c \cdot a' \end{array}$$

These assumptions are derivable only through rules [EXP REF] and [EXP REF RACE]; hence, we know

$$\begin{array}{l}
P; E \vdash \mathcal{E}'[e] : c_o \cdot a_o \\
P; E \vdash \mathcal{E}'[v] : c_o \cdot a'_o \\
\mathtt{class}\ c_o\ \mathtt{\{}\ \ldots c\ f\ \ldots \mathtt{\}} \in P \\
a = a_o; a_f \\
a' = a'_o; a_f
\end{array}$$

where a_f is one of four cases

$$\begin{array}{ll}
\mathrm{AM} & \text{if } f \in \textit{Normal} \text{ and } \gamma = . \\
\mathrm{AF} & \text{if } f \in \textit{Final} \text{ and } \gamma = . \\
\mathrm{AN} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = . \\
\mathrm{CL} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = ..
\end{array}$$

By IH, we have

$$Y(\mathcal{E}', a''); a'_o \sqsubseteq a_o$$

Since $\textit{locks}(\mathcal{E}') = \textit{locks}(\mathcal{E})$, we have

$$Y(\mathcal{E}, a''); a'_o \sqsubseteq a_o$$

By Lemma 1, we get

$$(Y(\mathcal{E}, a''); a'_o; a_f) \sqsubseteq (a_o; a_f)$$

Hence we obtain our desired result

$$Y(\mathcal{E}, a''); a' \sqsubseteq a$$

- $\mathcal{E} \equiv \mathcal{E}'{}_\gamma f\ \mathtt{=}\ e'$: As assumptions, we have

$$\begin{array}{l}
P; E \vdash \mathcal{E}'[e]_\gamma f\ \mathtt{=}\ e' : c \cdot a \\
P; E \vdash \mathcal{E}'[v]_\gamma f\ \mathtt{=}\ e' : c \cdot a'
\end{array}$$

These assumptions are derivable only through rules [EXP ASSIGN] and [EXP ASSIGN RACE]; hence, we know

$$\begin{array}{l}
P; E \vdash \mathcal{E}'[e] : c_o \cdot a_o \\
P; E \vdash \mathcal{E}'[v] : c_o \cdot a'_o \\
P; E \vdash e' : c \cdot a_t \\
\mathtt{class}\ c_o\ \mathtt{\{}\ \ldots c\ f\ \ldots \mathtt{\}} \in P \\
a = a_o; a_t; a_f \\
a' = a'_o; a_t; a_f
\end{array}$$

where a_f is one of four cases

$$\begin{array}{ll}
\mathrm{AM} & \text{if } f \in \textit{Normal} \text{ and } \gamma = . \\
\mathrm{AF} & \text{if } f \in \textit{Final} \text{ and } \gamma = . \\
\mathrm{AN} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = . \\
\mathrm{CL} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = ..
\end{array}$$

By IH, we have

$$Y(\mathcal{E}', a''); a'_o \sqsubseteq a_o$$

Since $\textit{locks}(\mathcal{E}') = \textit{locks}(\mathcal{E})$, we have

$$Y(\mathcal{E}, a''); a'_o \sqsubseteq a_o$$

By Lemma 1, we get

$$(Y(\mathcal{E}, a''); a'_o; a_t; a_f) \sqsubseteq (a_o; a_t; a_f)$$

Hence, by substitution, we obtain our desired result

$$Y(\mathcal{E}, a''); a' \sqsubseteq a$$

- $\mathcal{E} \equiv v'{}_\gamma f\ \mathtt{=}\ \mathcal{E}'$: As assumptions, we have

$$\begin{array}{l}
P; E \vdash v'{}_\gamma f\ \mathtt{=}\ \mathcal{E}'[e] : c \cdot a \\
P; E \vdash v'{}_\gamma f\ \mathtt{=}\ \mathcal{E}'[v] : c \cdot a'
\end{array}$$

These assumptions are derivable only through rules [EXP ASSIGN] and [EXP ASSIGN RACE]; hence, we know

$$\begin{array}{l}
P; E \vdash v' : c_o \cdot a_o \\
P; E \vdash \mathcal{E}'[e] : c \cdot a_t \\
P; E \vdash \mathcal{E}'[v] : c \cdot a'_t \\
\mathtt{class}\ c_o\ \mathtt{\{}\ \ldots c\ f\ \ldots \mathtt{\}} \in P \\
a_o = \mathrm{AF} \\
a = a_o; a_t; a_f \\
a' = a_o; a'_t; a_f
\end{array}$$

where a_f is one of four cases

$$\begin{array}{ll}
\mathrm{AM} & \text{if } f \in \textit{Normal} \text{ and } \gamma = . \\
\mathrm{AF} & \text{if } f \in \textit{Final} \text{ and } \gamma = . \\
\mathrm{AN} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = . \\
\mathrm{CL} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = ..
\end{array}$$

By IH, we have

$$Y(\mathcal{E}', a''); a'_t \sqsubseteq a_t$$

Since $\textit{locks}(\mathcal{E}') = \textit{locks}(\mathcal{E})$, we have

$$Y(\mathcal{E}, a''); a'_t \sqsubseteq a_t$$

By Lemma 1, and since AF is identity for sequential composition, we get

$$(Y(\mathcal{E}, a''); a_o; a'_t; a_f) \sqsubseteq (a_o; a_t; a_f)$$

Hence, by substitution, we obtain our desired result

$$Y(\mathcal{E}, a''); a' \sqsubseteq a$$

- $\mathcal{E} \equiv \mathcal{E}'{}_\gamma m(\overline{e})$: As assumptions, we have

$$\begin{array}{l}
P; E \vdash \mathcal{E}'[e]_\gamma m(\overline{e}) : c \cdot a \\
P; E \vdash \mathcal{E}'[v]_\gamma m(\overline{e}) : c \cdot a'
\end{array}$$

These assumptions are derivable only through the rule [EXP INVOKE]; hence, we know

$$\begin{array}{l}
P; E \vdash \mathcal{E}'[e] : c_o \cdot a_o \\
P; E \vdash \mathcal{E}'[v] : c_o \cdot a'_o \\
\mathtt{class}\ c_o\ \mathtt{\{}\ \ldots \textit{meth} \ldots \mathtt{\}} \\
\textit{meth} = a_m\ c\ m\mathtt{(}d_i\ x_i\ {}^{i \in 1..n}\mathtt{)\{}\ e'\ \mathtt{\}} \\
P; E \vdash e_i : d_i \cdot a_i \qquad \forall i \in 1..n \\
P; E \vdash a_m[\mathtt{this} := \mathcal{E}'[e], x_i := e_i\ {}^{i \in 1..n}] \uparrow a_\ell \\
P; E \vdash a_m[\mathtt{this} := \mathcal{E}'[v], x_i := e_i\ {}^{i \in 1..n}] \uparrow a'_\ell \\
a = a_o; a_1; \ldots; a_n; [\![\gamma]\!]; a_\ell \\
a' = a'_o; a_1; \ldots; a_n; [\![\gamma]\!]; a'_\ell \\
a_\ell \sqsubseteq \mathrm{AN} \\
a'_\ell \sqsubseteq \mathrm{AN}
\end{array}$$

By IH, we have

$$Y(\mathcal{E}', a''); a'_o \sqsubseteq a_o$$

Since $\textit{locks}(\mathcal{E}') = \textit{locks}(\mathcal{E})$, we have

$$Y(\mathcal{E}, a''); a'_o \sqsubseteq a_o$$

By Lemma 1, we get

$$\begin{array}{rl}
(Y(\mathcal{E}, a''); a') = & (Y(\mathcal{E}, a''); a'_o; a_1; \ldots; a_n; [\![\gamma]\!]; a'_\ell) \\
& \sqsubseteq (a_o; a_1; \ldots; a_n; [\![\gamma]\!]; a'_\ell)
\end{array}$$

Since $\mathcal{E}'[e]$ cannot be a value due to the presence of e, by Lemma 11, we have $a'_\ell \sqsubseteq a_\ell$. Again by Lemma 1, we get

$$(a_o; a_1; \ldots; a_n; [\![\gamma]\!]; a'_\ell) \sqsubseteq (a_o; a_1; \ldots; a_n; [\![\gamma]\!]; a_\ell) = a$$

Combining, we get our desired result

$$Y(\mathcal{E}, a''); a' \sqsubseteq a$$

- $\mathcal{E} \equiv \mathcal{E}'{}_\gamma m\mathtt{\#}(\overline{e})$: Similar argument to previous case.

- $\mathcal{E} \equiv v'{}_\gamma m(\overline{v},\ \mathcal{E}', \overline{e})$: As assumptions, we have

$$\begin{array}{l} P; E \vdash v'{}_\gamma m(\overline{v},\ \mathcal{E}'[e],\ \overline{e})\ \mathtt{:}\ c \cdot a \\ P; E \vdash v'{}_\gamma m(\overline{v},\ \mathcal{E}'[v],\ \overline{e})\ \mathtt{:}\ c \cdot a' \end{array}$$

These assumptions are derivable only through the rule [EXP INVOKE]; hence, we know the following, where we take $\mathcal{E}'[e]$ to be in the k^{th} position in the list of arguments.

$$\begin{array}{l} P; E \vdash v'\ \mathtt{:}\ c_o \cdot a_o \\ \mathtt{class}\ c_o\ \mathtt{\{}\ \ldots \mathit{meth} \ldots \mathtt{\}} \\ \mathit{meth} = a_m\ c\ m\mathtt{(}d_i\ x_i\ {}^{i \in 1..n}\mathtt{)\{}\ e'\ \mathtt{\}} \\ P; E \vdash v_i\ \mathtt{:}\ d_i \cdot a_i \qquad \forall i \in 1..k-1 \\ P; E \vdash \mathcal{E}'[e]\ \mathtt{:}\ c_k \cdot a_k \\ P; E \vdash \mathcal{E}'[v]\ \mathtt{:}\ c_k \cdot a'_k \\ P; E \vdash e_i\ \mathtt{:}\ d_i \cdot a_i \qquad \forall i \in k+1..n \\ P; E \vdash a_m[\mathtt{this} := v', x_i := v_i\ {}^{i \in 1..k-1}, \\ \quad x_k := \mathcal{E}'[e], x_i := e_i\ {}^{i \in k+1..n}] \uparrow a_\ell \\ P; E \vdash a_m[\mathtt{this} := v', x_i := v_i\ {}^{i \in 1..k-1}, \\ \quad x_k := \mathcal{E}'[v], x_i := e)i\ {}^{i \in k+1..n}] \uparrow a'_\ell \\ a_o = \mathrm{AF} \\ a_i = \mathrm{AF} \qquad \forall i \in 1..k-1 \\ a = a_o; a_1; \ldots; a_k; \ldots; a_n; \llbracket\gamma\rrbracket; a_\ell \\ a' = a_o; a_1; \ldots; a'_k; \ldots; a_n; \llbracket\gamma\rrbracket; a'_\ell \\ a_\ell \sqsubseteq \mathrm{AN} \\ a'_\ell \sqsubseteq \mathrm{AN} \end{array}$$

Both a and a' simplify to the following, due to AF being the identity of sequential composition:

$$\begin{array}{l} a = a_k; \ldots; a_n; \llbracket\gamma\rrbracket; a_\ell \\ a' = a'_k; \ldots; a_n; \llbracket\gamma\rrbracket; a'_\ell \end{array}$$

By IH, we have

$$Y(\mathcal{E}', a''); a'_k \sqsubseteq a_k$$

Since $\mathit{locks}(\mathcal{E}') = \mathit{locks}(\mathcal{E})$, we have

$$Y(\mathcal{E}, a''); a'_k \sqsubseteq a_k$$

By Lemma 1, we get

$$\begin{array}{l} (\,Y(\mathcal{E}, a''); a') \\ = (\,Y(\mathcal{E}, a''); a'_k; \ldots; a_n; \llbracket\gamma\rrbracket; a'_\ell) \\ \sqsubseteq (a_k; \ldots; a_n; \llbracket\gamma\rrbracket; a'_\ell) \end{array}$$

Since $\mathcal{E}'[e]$ cannot be a value due to the presence of e, by Lemma 11, we have $a'_\ell \sqsubseteq a_\ell$. Again by Lemma 1, we get

$$(a_k; \ldots; a_n; \llbracket\gamma\rrbracket; a'_\ell) \sqsubseteq (a_k; \ldots; a_n; \llbracket\gamma\rrbracket; a_\ell) = a$$

Combining, we get our desired result

$$Y(\mathcal{E}, a''); a' \sqsubseteq a$$

- $\mathcal{E} \equiv \rho_\gamma m\mathtt{\#(}\overline{v},\ \mathcal{E}', \overline{e}\mathtt{)}$: Similar argument to previous case.
- $\mathcal{E} \equiv \mathtt{let}\ x\ \mathtt{=}\ \mathcal{E}'\ \mathtt{in}\ e'$: As assumptions, we have

$$\begin{array}{l} P; E \vdash \mathtt{let}\ x\ \mathtt{=}\ \mathcal{E}'[e]\ \mathtt{in}\ e'\ \mathtt{:}\ c \cdot a \\ P; E \vdash \mathtt{let}\ x\ \mathtt{=}\ \mathcal{E}'[v]\ \mathtt{in}\ e'\ \mathtt{:}\ c \cdot a' \end{array}$$

These assumptions are derivable only through rule [EXP LET]; hence we know

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e]\ \mathtt{:}\ c_x \cdot a_x \\ P; E \vdash \mathcal{E}'[v]\ \mathtt{:}\ c_x \cdot a'_x \\ P; E, c_x\ x \vdash e'\ \mathtt{:}\ c \cdot a_b \\ P; E \vdash a_b[x := \mathcal{E}'[e]] \uparrow a_\ell \\ P; E \vdash a_b[x := \mathcal{E}'[v]] \uparrow a'_\ell \\ a = (a_x; a_\ell) \\ a' = (a'_x; a'_\ell) \end{array}$$

By IH, we have

$$Y(\mathcal{E}', a''); a'_x \sqsubseteq a_x$$

Since $\mathit{locks}(\mathcal{E}') = \mathit{locks}(\mathcal{E})$, we have

$$Y(\mathcal{E}, a''); a'_x \sqsubseteq a_x$$

By Lemma 1, we get

$$(\,Y(\mathcal{E}, a''); a') = (\,Y(\mathcal{E}, a''); a'_x; a'_\ell) \sqsubseteq (a_x; a'_\ell)$$

Since $\mathcal{E}'[e]$ cannot be a value due to the presence of e, by Lemma 11, we have $a'_\ell \sqsubseteq a_\ell$. Applying Lemma 1, we get

$$(a_x; a'_\ell) \sqsubseteq (a_x; a_\ell) = a$$

Combining, we get our desired result

$$Y(\mathcal{E}, a''); a' \sqsubseteq a$$

- $\mathcal{E} \equiv \mathtt{if}\ \mathcal{E}'\ e_1\ e_2$: As assumptions, we have

$$\begin{array}{l} P; E \vdash \mathtt{if}\ \mathcal{E}'[e]\ e_1\ e_2\ \mathtt{:}\ c \cdot a \\ P; E \vdash \mathtt{if}\ \mathcal{E}'[v]\ e_1\ e_2\ \mathtt{:}\ c \cdot a' \end{array}$$

These assumptions are derivable only through rule [EXP IF]; hence we know

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e]\ \mathtt{:}\ c_g \cdot a_g \\ P; E \vdash \mathcal{E}'[v]\ \mathtt{:}\ c_g \cdot a'_g \\ P; E \vdash e_1\ \mathtt{:}\ c \cdot a_1 \\ P; E \vdash e_2\ \mathtt{:}\ c \cdot a_2 \\ a = (a_g; (a_1 \sqcup a_2)) \\ a' = (a'_g; (a_1 \sqcup a_2)) \end{array}$$

By IH, we have

$$Y(\mathcal{E}', a''); a'_g \sqsubseteq a_g$$

Since $\mathit{locks}(\mathcal{E}') = \mathit{locks}(\mathcal{E})$, we have

$$Y(\mathcal{E}, a''); a'_g \sqsubseteq a_g$$

By Lemma 1, we get

$$(\,Y(\mathcal{E}, a''); a'_g; (a_1 \sqcup a_2)) \sqsubseteq (a_g; (a_1 \sqcup a_2))$$

Hence we obtain our desired result

$$Y(\mathcal{E}, a''); a' \sqsubseteq a$$

- $\mathcal{E} \equiv \mathcal{E}'{}_\gamma\mathtt{sync}\ e'$: As assumptions, we have

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e]_\gamma\mathtt{sync}\ e'\ \mathtt{:}\ c \cdot a \\ P; E \vdash \mathcal{E}'[v]_\gamma\mathtt{sync}\ e'\ \mathtt{:}\ c \cdot a' \end{array}$$

We encounter a contradiction in the first assumption. The judgment $P; E \vdash \mathcal{E}'[e]_\gamma\mathtt{sync}\ e'\ \mathtt{:}\ c \cdot a$ can only be derived with rule [EXP SYNC], and hence it must be that

$$P; E \vdash_{\mathrm{lock}} \mathcal{E}'[e]$$

However, with non-value e contained in this lock expression, we cannot actually make this judgment. Thus by contradiction we conclude this case is trivially true.

- $\mathcal{E} \equiv \mathtt{in\text{-}sync}\ \rho\ \mathcal{E}'$: As assumptions, we have

$$\begin{array}{l} P; E \vdash \mathtt{in\text{-}sync}\ \rho\ \mathcal{E}'[e]\ \mathtt{:}\ c \cdot a \\ P; E \vdash \mathtt{in\text{-}sync}\ \rho\ \mathcal{E}'[v]\ \mathtt{:}\ c \cdot a' \end{array}$$

These assumptions are derivable only through rule [EXP INSYNC]; hence we know

$$\begin{array}{l} P; E \vdash_{\mathrm{lock}} \rho \\ P; E \vdash \mathcal{E}'[e]\ \mathtt{:}\ c \cdot a_b \\ P; E \vdash \mathcal{E}'[v]\ \mathtt{:}\ c \cdot a'_b \\ a = \mathcal{SI}(\rho, a_b) \\ a' = \mathcal{SI}(\rho, a'_b) \end{array}$$

By IH, we have

$$Y(\mathcal{E}', a''); a'_b \sqsubseteq a_b$$

By repeated application of Lemma 17, we have

$$\begin{array}{c} Y(\mathcal{E}, a'') \sqsubseteq a'' \\ Y(\mathcal{E}', Y(\mathcal{E}, a'')) \sqsubseteq Y(\mathcal{E}', a'') \end{array}$$

Since $\mathit{locks}(\mathcal{E}') \subset \mathit{locks}(\mathcal{E})$, we have

$$Y(\mathcal{E}, a'') = Y(\mathcal{E}', Y(\mathcal{E}, a''))$$

and so by transitivity and Lemma 1,

$$\begin{array}{c} Y(\mathcal{E}, a'') \sqsubseteq Y(\mathcal{E}', a'') \\ Y(\mathcal{E}, a''); a'_b \sqsubseteq a_b \end{array}$$

We may strengthen the last relation by inclusion of $\{\rho\}$ in the held lockset:

$$Y(\mathcal{E}, a''); a'_b \sqsubseteq^{\{\rho\}} a_b$$

Then by Lemma 20, we have

$$Y(\mathcal{E}, a''); \mathcal{SI}(\rho, a'_b) \sqsubseteq^{\{\rho\}} \mathcal{SI}(\rho, a_b)$$

Since the left and right hand effects both are not conditional on ρ, by Lemma 18 we have

$$Y(\mathcal{E}, a''); \mathcal{SI}(\rho, a'_b) \sqsubseteq \mathcal{SI}(\rho, a_b)$$

Hence we obtain our desired result

$$Y(\mathcal{E}, a''); a' \sqsubseteq a$$

LEMMA 4 (Context Replacement). *Suppose the following:*

$$\begin{array}{l} P; E \vdash \mathcal{E}[e_1] : c \cdot a, \ \textit{and} \\ P; E \vdash e_1 : d \cdot a_1, \ \textit{and} \\ e_1 \ \textit{not a value, and} \\ P; E \vdash e_2 : d \cdot a_2, \ \textit{and} \\ a_2 \sqsubseteq a_1. \end{array}$$

Then $P; E \vdash \mathcal{E}[e_2] : c \cdot a'$ *and* $a' \sqsubseteq a$.

Proof Let $h = \emptyset$, $n = \emptyset$, and $\kappa_y = \text{AF}$. The assumption $a_2 \sqsubseteq a_1$ implies

$$\kappa_y; a_2 \sqsubseteq_n^{h \cup \mathit{locks}(\mathcal{E})} a_1$$

Then applying Lemma 5, we obtain the judgment $P; E \vdash \mathcal{E}[e_2] : c \cdot a'$ where $a' \sqsubseteq a$.

LEMMA 5 (Context Replacement 2). *Suppose the following:*

$$\begin{array}{l} P; E \vdash \mathcal{E}[e_1] : c \cdot a'_1, \ \textit{and} \\ P; E \vdash e_1 : d \cdot a_1, \ \textit{and} \\ e_1 \ \textit{not a value, and} \\ P; E \vdash e_2 : d \cdot a_2, \ \textit{and} \\ \exists\, h, n \,.\, h \not\pitchfork n \not\pitchfork \mathit{locks}(\mathcal{E}) \\ \kappa_y; a_2 \sqsubseteq_n^{h \cup \mathit{locks}(\mathcal{E})} a_1 \end{array}$$

Then $P; E \vdash \mathcal{E}[e_2] : c \cdot a'_2$ *and* $(\kappa_y; a'_2) \sqsubseteq_n^h a'_1$.

Proof By induction over the structure of $\mathcal{E}$.

- $\mathcal{E} \equiv [\,]$: As assumptions, we have

  $$\begin{array}{c} P; E \vdash [e_1] : c \cdot a'_1 \\ P; E \vdash e_2 : d \cdot a_2 \end{array}$$

  and may obtain the equalities

  $$\begin{array}{c} a'_1 = a_1 \\ a'_2 = a_2 \end{array}$$

  Since $\mathcal{E} \equiv [\,]$, we have $\mathit{locks}(\mathcal{E}) = \emptyset$. From the assumption $\kappa_y; a_2 \sqsubseteq_n^{h \cup \mathit{locks}(\mathcal{E})} a_1$ and substituting, we obtain

  $$\kappa_y; a'_2 \sqsubseteq_n^h a'_1$$

  which is our desired result.

- $\mathcal{E} \equiv \texttt{new}\, c\ \texttt{(}\ \overline{v}, \mathcal{E}', \overline{e}\texttt{)}$: As an assumption, we have

  $$P; E \vdash \texttt{new}\, c\ \texttt{(}\ \overline{v}, \mathcal{E}'[e_1], \overline{e}\texttt{)} : c \cdot a'_1$$

  Let us take $\mathcal{E}'[e_1]$ to be in the k^{th} position in the list of arguments. This assumption is derivable only through rule [EXP NEW]; hence we know

  $$\begin{array}{l} \texttt{class}\, c\, \texttt{\{}\, d_i\ x_i\ {}^{i \in 1..n} \texttt{\}} \in P \\ P; E \vdash v_i : c_i \cdot a_i \qquad \forall i \in 1..k-1 \\ P; E \vdash \mathcal{E}'[e_1] : c_k \cdot a_k \\ P; E \vdash e_i : c_i \cdot a_i \qquad \forall i \in k+1..n \\ a_i = \text{AF} \qquad \forall i \in 1..k-1 \\ a'_1 = (a_1; \ldots; a_k; \ldots; a_n; \text{AM}) \end{array}$$

  We have $\mathit{locks}(\mathcal{E}) = \mathit{locks}(\mathcal{E}')$, since the $\mathcal{E} \not\equiv \texttt{in-sync}\ \ell\ \mathcal{E}'$. By IH, we get

  $$\begin{array}{l} P; E \vdash \mathcal{E}'[e_2] : c_k \cdot a'_k \\ (\kappa_y; a'_k) \sqsubseteq_n^h a_k \end{array}$$

  Using rule [EXP NEW], we may conclude

  $$\begin{array}{l} P; E \vdash \texttt{new}\, c\ \texttt{(}\ \overline{v}, \mathcal{E}'[e_2], \overline{e}\texttt{)} : c \cdot a'_2 \\ a'_2 = (a_1; \ldots; a'_k; a_n; \text{AM}) \end{array}$$

  Since AF is the identity for sequential composition, we may simplify a'_1 and a'_2 as follows:

  $$\begin{array}{l} a'_1 = (a_k; \ldots; a_n; \text{AM}) \\ a'_2 = (a'_k; \ldots; a_n; \text{AM}) \end{array}$$

  By Lemma 1 and $(\kappa_y; a'_k) \sqsubseteq_n^h a_k$, we have

  $$(\kappa_y; a'_k; \ldots; a_n; \text{AM}) \sqsubseteq_n^h (a_k; \ldots; a_n; \text{AM})$$

  Hence we obtain our desired result

  $$(\kappa_y; a'_2) \sqsubseteq_n^h a'_1$$

- $\mathcal{E} \equiv \mathcal{E}'{}_\gamma f$: As an assumption, we have

  $$P; E \vdash \mathcal{E}'[e_1]_\gamma f : c \cdot a'_1$$

  This assumption is derivable only through rules [EXP REF] and [EXP REF RACE]; hence we know

  $$\begin{array}{l} P; E \vdash \mathcal{E}'[e_1] : c_o \cdot a_o \\ \texttt{class}\, c_o\, \texttt{\{}\ \ldots c\ f \ldots \texttt{\}} \in P \\ a'_1 = (a_o; a_f) \end{array}$$

  where a_f is one of four cases

  $$\begin{array}{ll} \text{AM} & \text{if } f \in \textit{Normal} \text{ and } \gamma = . \\ \text{AF} & \text{if } f \in \textit{Final} \text{ and } \gamma = . \\ \text{AN} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = . \\ \text{CL} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = .. \end{array}$$

  We have $\mathit{locks}(\mathcal{E}) = \mathit{locks}(\mathcal{E}')$, since $\mathcal{E} \not\equiv \texttt{in-sync}\ \ell\ \mathcal{E}'$. By IH, we have

  $$\begin{array}{l} P; E \vdash \mathcal{E}'[e_2] : c_o \cdot a'_o \\ (\kappa_y; a'_o) \sqsubseteq_n^h a_o \end{array}$$

  Using rules [EXP REF] and [EXP REF RACE], we may conclude

  $$\begin{array}{l} P; E \vdash \mathcal{E}'[e_1]_\gamma f : c \cdot a'_2 \\ a'_2 = (a'_o; a_f) \end{array}$$

  By Lemma 1 and $(\kappa_y; a'_o) \sqsubseteq_n^h a_o$, we get

  $$(\kappa_y; a'_o; a_f) \sqsubseteq_n^h (a_o; a_f)$$

Hence we obtain our desired result

$$(\kappa_y; a_2') \sqsubseteq_n^h a_1'$$

- $\mathcal{E} \equiv \mathcal{E}'{}_\gamma f$ `=` e: As an assumption, we have

$$P; E \vdash \mathcal{E}'[e_1]_\gamma f \texttt{ = } e \texttt{ : } c \cdot a_1'$$

This assumption is derivable only through rules [EXP ASSIGN] and [EXP ASSIGN RACE]; hence we know

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_1] \texttt{ : } c_o \cdot a_o \\ P; E \vdash e \texttt{ : } c \cdot a_t \\ \texttt{class}\ c_o\ \texttt{\{}\ \ldots c\ f \ldots \texttt{\}} \in P \\ a_1' = (a_o; a_t; a_f) \end{array}$$

where a_f is one of three cases

$$\begin{array}{ll} \text{AM} & \text{if } f \in \mathit{Normal} \text{ and } \gamma = . \\ \text{AN} & \text{if } f \in \mathit{Volatile} \text{ and } \gamma = . \\ \text{CL} & \text{if } f \in \mathit{Volatile} \text{ and } \gamma = .. \end{array}$$

We have $\mathit{locks}(\mathcal{E}) = \mathit{locks}(\mathcal{E}')$, since $\mathcal{E} \not\equiv \texttt{in-sync}\ \ell\ \mathcal{E}'$. By IH, we have

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_2] \texttt{ : } c_o \cdot a_o' \\ (\kappa_y; a_o') \sqsubseteq_n^h a_o \end{array}$$

Using rules [EXP ASSIGN] and [EXP ASSIGN RACE], we may conclude

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_1]_\gamma f \texttt{ = } e \texttt{ : } c \cdot a_2' \\ a_2' = (a_o'; a_t; a_f) \end{array}$$

By Lemma 1 and $(\kappa_y; a_o') \sqsubseteq_n^h a_o$, we get

$$(\kappa_y; a_o'; a_t; a_f) \sqsubseteq_n^h (a_o; a_t; a_f)$$

Hence we obtain our desired result

$$(\kappa_y; a_2') \sqsubseteq_n^h a_1'$$

- $\mathcal{E} \equiv v_\gamma f$ `=` $\mathcal{E}'$: As an assumption, we have

$$P; E \vdash v_\gamma f \texttt{ = } \mathcal{E}'[e_1] \texttt{ : } c \cdot a_1'$$

This assumption is derivable only through rules [EXP ASSIGN] and [EXP ASSIGN RACE]; hence we know

$$\begin{array}{l} P; E \vdash v \texttt{ : } c_o \cdot a_o \\ P; E \vdash \mathcal{E}'[e_1] \texttt{ : } c \cdot a_t \\ \texttt{class}\ c_o\ \texttt{\{}\ \ldots c\ f \ldots \texttt{\}} \in P \\ a_o = \text{AF} \\ a_1' = (a_o; a_t; a_f) \end{array}$$

where a_f is one of three cases

$$\begin{array}{ll} \text{AM} & \text{if } f \in \mathit{Normal} \text{ and } \gamma = . \\ \text{AN} & \text{if } f \in \mathit{Volatile} \text{ and } \gamma = . \\ \text{CL} & \text{if } f \in \mathit{Volatile} \text{ and } \gamma = .. \end{array}$$

We have $\mathit{locks}(\mathcal{E}) = \mathit{locks}(\mathcal{E}')$, since $\mathcal{E} \not\equiv \texttt{in-sync}\ \ell\ \mathcal{E}'$. By IH, we have

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_2] \texttt{ : } c \cdot a_t' \\ (\kappa_y; a_t') \sqsubseteq_n^h a_t \end{array}$$

Using rules [EXP ASSIGN] and [EXP ASSIGN RACE], we may conclude

$$\begin{array}{l} P; E \vdash v_\gamma f \texttt{ = } \mathcal{E}'[e_2] \texttt{ : } c \cdot a_2' \\ a_2' = (a_o; a_t'; a_f) \end{array}$$

Since AF is the identity for sequential composition, we may simplify a_1' and a_2' as follows:

$$\begin{array}{l} a_1' = (a_t; a_f) \\ a_2' = (a_t'; a_f) \end{array}$$

By Lemma 1 and $(\kappa_y; a_t') \sqsubseteq_n^h a_t$, we get

$$(\kappa_y; a_t'; a_f) \sqsubseteq_n^h (a_t; a_f)$$

Hence we obtain our desired result

$$(\kappa_y; a_2') \sqsubseteq_n^h a_1'$$

- $\mathcal{E} \equiv \mathcal{E}'{}_\gamma m(\overline{e})$: As an assumption, we have

$$P; E \vdash \mathcal{E}'[e_1]_\gamma m(\overline{e}) \texttt{ : } c \cdot a_1'$$

This assumption is derivable only through rule [EXP INVOKE]; hence we know

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_1] \texttt{ : } c_o \cdot a_o \\ \texttt{class}\ c_o\ \texttt{\{}\ \ldots \mathit{meth} \ldots \texttt{\}} \in P \\ \mathit{meth} = a_m\ c\ m(d_i\ x_i\ {}^{i \in 1..n})\texttt{\{}\ e'\texttt{\}} \\ P; E \vdash e_i \texttt{ : } d_i \cdot a_i \qquad \forall i \in 1..n \\ P; E \vdash a_m[\texttt{this} := \mathcal{E}'[e_1], x_i := e_i\ {}^{i \in 1..n}] \uparrow a_m' \\ a_m' \sqsubseteq \text{AN} \\ a_1' = (a_o; a_1; \ldots; a_n; [\![\gamma]\!]; a_m') \end{array}$$

We have $\mathit{locks}(\mathcal{E}) = \mathit{locks}(\mathcal{E}')$, since $\mathcal{E} \not\equiv \texttt{in-sync}\ \ell\ \mathcal{E}'$. By IH, we have

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_2] \texttt{ : } c_o \cdot a_o' \\ (\kappa_y; a_o') \sqsubseteq_n^h a_o \end{array}$$

From $P \vdash E$ and the lifting rules we may derive

$$P; E \vdash a_m[\texttt{this} := \mathcal{E}'[e_2], x_i := e_i\ {}^{i \in 1..n}] \uparrow a_m''$$

By Lemma 11 and the fact that $\mathcal{E}'[e_1]$ cannot be a value, we have

$$a_m'' \sqsubseteq a_m'$$

Using rule [EXP INVOKE], we may conclude

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_2]_\gamma m(\overline{e}) \texttt{ : } c \cdot a_2' \\ a_2' = (a_o'; a_1; \ldots; a_n; [\![\gamma]\!]; a_m'') \end{array}$$

By Lemma 1 and $(\kappa_y; a_o') \sqsubseteq_n^h a_o$ we have

$$(\kappa_y; a_o'; a_1; \ldots; a_n; [\![\gamma]\!]; a_m'') \sqsubseteq_n^h (a_o; a_1; \ldots; a_n; [\![\gamma]\!]; a_m'')$$

By a separate application of Lemma 1 to $a_m'' \sqsubseteq a_m'$, we have

$$(a_o; a_1; \ldots; a_n; [\![\gamma]\!]; a_m'') \sqsubseteq (a_o; a_1; \ldots; a_n; [\![\gamma]\!]; a_m')$$

Combining these, and substituting definitions of a_1' and a_2', we obtain our desired result

$$(\kappa_y; a_2') \sqsubseteq_n^h a_1'$$

- $\mathcal{E} \equiv \mathcal{E}'{}_\gamma m\texttt{\#}(\overline{e})$: Similar argument to previous case.
- $\mathcal{E} \equiv v_\gamma m(\overline{v}, \mathcal{E}', \overline{e})$: As an assumption, we have

$$P; E \vdash v_\gamma m(\overline{v}, \mathcal{E}'[e_1], \overline{e}) \texttt{ : } c \cdot a_1'$$

Let us take $\mathcal{E}'[e_1]$ to be in the k^{th} position in the list of arguments. This assumption is derivable only through rule [EXP INVOKE]; hence we know

$$\begin{array}{l} P; E \vdash v \texttt{ : } c_o \cdot a_o \\ \texttt{class}\ c_o\ \texttt{\{}\ \ldots \mathit{meth} \ldots \texttt{\}} \in P \\ \mathit{meth} = a_m\ c\ m(d_i\ x_i\ {}^{i \in 1..n})\texttt{\{}\ e'\texttt{\}} \\ P; E \vdash v_i \texttt{ : } d_i \cdot a_i \qquad \forall i \in 1..k-1 \\ P; E \vdash \mathcal{E}'[e_1] \texttt{ : } d_k \cdot a_k \\ P; E \vdash e_i \texttt{ : } d_i \cdot a_i \qquad \forall i \in k+1..n \\ P; E \vdash a_m[\texttt{this} := v, x_i := v_i\ {}^{i \in 1..k-1}, \\ \quad x_k := \mathcal{E}'[e_1], x_i := e_i\ {}^{i \in k+1..n}] \uparrow a_m' \\ a_o = \text{AF} \\ a_m' \sqsubseteq \text{AN} \\ a_1' = (a_o; a_1; \ldots; a_k; \ldots; a_n; [\![\gamma]\!]; a_m') \end{array}$$

We have $\mathit{locks}(\mathcal{E}) = \mathit{locks}(\mathcal{E}')$, since $\mathcal{E} \not\equiv \texttt{in-sync}\ \ell\ \mathcal{E}'$. By IH, we have

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_2] \texttt{ : } d_k \cdot a_k' \\ (\kappa_y; a_k') \sqsubseteq_n^h a_k \end{array}$$

From $P \vdash E$ and the lifting rules we may derive

$$\begin{array}{l} P; E \vdash a_m[\texttt{this} := v, x_i := v_i \ ^{i \in 1..k-1}, \\ \quad x_k := \mathcal{E}'[e_2], x_i := e_i \ ^{i \in k+1..n}] \uparrow a''_m \end{array}$$

By Lemma 11 and the fact that $\mathcal{E}'[e_1]$ cannot be a value, we have

$$a''_m \sqsubseteq a'_m$$

Using rule [EXP INVOKE], we may conclude

$$\begin{array}{l} P; E \vdash v_\gamma m(\overline{v}, \mathcal{E}'[e_2], \overline{e}) : c \cdot a'_2 \\ a'_2 = (a_o; a_1; \ldots; a'_k; \ldots; a_n; [\![\gamma]\!]; a''_m) \end{array}$$

Since AF is the identity for sequential composition, we may simplify a'_1 and a'_2 to the following:

$$\begin{array}{l} a'_1 = a_k; \ldots; a_n; [\![\gamma]\!]; a'_m \\ a'_2 = a'_k; \ldots; a_n; [\![\gamma]\!]; a''_m \end{array}$$

By Lemma 1 and $(\kappa_y; a'_k) \sqsubseteq^h_n a_k$ we have

$$(\kappa_y; a'_k; \ldots; a_n; [\![\gamma]\!]; a''_m) \sqsubseteq^h_n (a_k; \ldots; a_n; [\![\gamma]\!]; a''_m)$$

By a separate application of Lemma 1 to $a''_m \sqsubseteq a'_m$, we have

$$(a_k; \ldots; a_n; [\![\gamma]\!]; a''_m) \sqsubseteq (a_k; \ldots; a_n; [\![\gamma]\!]; a'_m)$$

Combining these, and substituting definitions of a'_1 and a'_2, we obtain our desired result

$$(\kappa_y; a'_2) \sqsubseteq^h_n a'_1$$

- $\mathcal{E} \equiv \rho_\gamma m\texttt{\#}(\overline{v}, \mathcal{E}', \overline{e})$: Similar argument to previous case.
- $\mathcal{E} \equiv \texttt{let}\ x\ \texttt{=}\ \mathcal{E}'\ \texttt{in}\ e$: As an assumption, we have

$$P; E \vdash \texttt{let}\ x\ \texttt{=}\ \mathcal{E}'[e_1]\ \texttt{in}\ e : c \cdot a'_1$$

This assumption is derivable only through rule [EXP LET]; hence we know

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_1] : c_x \cdot a_x \\ P; E, c_x\ x \vdash e : c \cdot a_b \\ P; E \vdash a_b[x := \mathcal{E}'[e_1]] \uparrow a_\ell \\ a'_1 = (a_x; a_\ell) \end{array}$$

We have $\mathit{locks}(\mathcal{E}) = \mathit{locks}(\mathcal{E}')$, since $\mathcal{E} \not\equiv \texttt{in-sync}\ \ell\ \mathcal{E}'$. By IH, we have

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_2] : c_x \cdot a'_x \\ (\kappa_y; a'_x) \sqsubseteq^h_n a_x \end{array}$$

From $P \vdash E$ and the lifting rules we may derive

$$P; E \vdash a_b[x := \mathcal{E}'[e_2]] \uparrow a'_\ell$$

By Lemma 11 and the fact that $\mathcal{E}'[e_1]$ cannot be a value, we have

$$a'_\ell \sqsubseteq a_\ell$$

Using rule [EXP LET], we may conclude

$$\begin{array}{l} P; E \vdash \texttt{let}\ x\ \texttt{=}\ \mathcal{E}'[e_2]\ \texttt{in}\ e : c \cdot a'_2 \\ a'_2 = (a'_x; a'_\ell) \end{array}$$

By Lemma 1 and $(\kappa_y; a'_x) \sqsubseteq^h_n a_x$, we get

$$(\kappa_y; a'_2) = (\kappa_y; a'_x; a'_\ell) \sqsubseteq^h_n (a_x; a'_\ell)$$

By a separate application of Lemma 1 to $a'_\ell \sqsubseteq a_\ell$, we have

$$(a_x; a'_\ell) \sqsubseteq (a_x; a_\ell) = a'_1$$

Combining these, we get our desired result

$$(\kappa_y; a'_2) \sqsubseteq^h_n a'_1$$

- $\mathcal{E} \equiv \texttt{if}\ \mathcal{E}'\ e_1\ e_2$: As an assumption, we have

$$P; E \vdash \texttt{if}\ \mathcal{E}'[e_1]\ e_\ell\ e_r : c \cdot a'_1$$

This assumption is derivable only through rule [EXP IF]; hence we know

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_1] : d \cdot a_g \\ P; E \vdash e_\ell : c \cdot a_\ell \\ P; E \vdash e_r : c \cdot a_r \\ a'_1 = (a_g; (a_\ell \sqcup a_r)) \end{array}$$

We have $\mathit{locks}(\mathcal{E}) = \mathit{locks}(\mathcal{E}')$, since $\mathcal{E} \not\equiv \texttt{in-sync}\ \ell\ \mathcal{E}'$. By IH, we have

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_2] : c \cdot a'_g \\ (\kappa_y; a'_g) \sqsubseteq^h_n a_g \end{array}$$

Using rule [EXP IF], we may conclude

$$\begin{array}{l} P; E \vdash \texttt{if}\ \mathcal{E}'[e_1]\ e_\ell\ e_r : c \cdot a'_2 \\ a'_2 = (a'_g; (a_\ell \sqcup a_r)) \end{array}$$

By Lemma 1 and $(\kappa_y; a'_g) \sqsubseteq^h_n a_g$, we get

$$(\kappa_y; a'_g; (a_\ell \sqcup a_r)) \sqsubseteq^h_n (a_g; (a_\ell \sqcup a_r))$$

By substituting definitions of a'_1 and a'_2, we obtain our desired result

$$(\kappa_y; a'_2) \sqsubseteq^h_n a'_1$$

- $\mathcal{E} \equiv \mathcal{E}'\,_\gamma\texttt{sync}\ e$: As an assumption, we have

$$P; E \vdash \mathcal{E}'[e_1]_\gamma\texttt{sync}\ e : c \cdot a'_1$$

We encounter a contradiction in this assumption. This assumption is derivable only through rule [EXP SYNC]; hence it must be that

$$P; E \vdash_{\text{lock}} \mathcal{E}'[e_1]$$

However, with non-value e_1 contained in this lock expression, we cannot actually make this judgment. Thus by contradiction this case is trivially true.

- $\mathcal{E} \equiv \texttt{in-sync}\ \rho\ \mathcal{E}'$: As an assumption, we have

$$P; E \vdash \texttt{in-sync}\ \rho\ \mathcal{E}'[e_1] : c \cdot a'_1$$

This assumption is derivable only through rule [EXP INSYNC]; hence we know

$$\begin{array}{l} P; E \vdash_{\text{lock}} \rho \\ P; E \vdash \mathcal{E}'[e_1] : c \cdot a_b \\ a'_1 = \mathcal{SI}(\rho, a_b) \end{array}$$

Rewriting our assumption, since

$$\begin{array}{l} \mathit{locks}(\mathcal{E}) = \{\rho\} \cup \mathit{locks}(\mathcal{E}') \\ \{\rho\} \not\cap \mathit{locks}(\mathcal{E}') \end{array}$$

we have

$$\kappa_y; a_2 \sqsubseteq^{h \cup \{\rho\} \cup \mathit{locks}(\mathcal{E}')}_n a_1$$

By IH, we have

$$\begin{array}{l} P; E \vdash \mathcal{E}'[e_2] : c \cdot a'_b \\ (\kappa_y; a'_b) \sqsubseteq^{h \cup \{\rho\}}_n a_b \end{array}$$

By Lemma 20, we have

$$\kappa_y; \mathcal{SI}(\rho, a'_b) \sqsubseteq^{h \cup \{\rho\}}_n \mathcal{SI}(\rho, a_b)$$

Since the left and right sides both do not contain a conditional on ρ, by Lemma 18, we have

$$\kappa_y; \mathcal{SI}(\rho, a'_b) \sqsubseteq^h_n \mathcal{SI}(\rho, a_b)$$

By substituting definitions of a'_1 and a'_2, we obtain our desired result

$$\kappa_y; a'_2 \sqsubseteq^h_n a'_1$$

LEMMA 6 (Substitution). *Suppose we have $P; E \vdash v \mathbf{:} c{\cdot}\textsc{af}$. Then the following is true:*

1. *If $P \vdash (E, c\,x, E')$ then $P \vdash (E, E'[x := v])$.*
2. *If $P; (E, c\ x, E') \vdash \mathit{meth}$ then $P; (E, E'[x := v]) \vdash \mathit{meth}[x := v]$.*
3. *If $P; (E, c\,x, E') \vdash a$ then $P; (E, E'[x := v]) \vdash a[x := v]$.*
4. *If $P; (E, c\ x, E') \vdash_{\text{lock}} \ell$ then $P; (E, E'[x := v]) \vdash_{\text{lock}} \ell[x := v]$.*
5. *If $P; (E, c\,x, E') \vdash a \uparrow a'$ then $P; (E, E'[x := v]) \vdash a[x := v] \uparrow a'[x := v]$.*
6. *If $P; (E, c\,x, E') \vdash e \mathbf{:} c' \cdot a$ then $P; (E, E'[x := v]) \vdash e[x := v] \mathbf{:} c' \cdot a[x := v]$.*

Proof By simultaneous induction on all parts of the lemma.

- Assume $P \vdash (E, c\,x, E')$. Then we know, by rule [ENV X] and by induction on length of E', that for all $(d\ y) \in E'$, we have $y \neq x$. Furthermore, a type d does not capture variables. Hence x is not referred to in E'. By rule [ENV X], removing $(c\,x)$ does not affect the well-formedness of the remaining environment bindings,

  $$P \vdash (E, E')$$

  and hence we obtain our desired result

  $$P \vdash (E, E'[x := v])$$

- Assume $P; (E, c\,x, E') \vdash \mathit{meth}$, where $\mathit{meth} = a\ d\ m\texttt{(}\overline{d}\ \overline{x}\texttt{)\{}\ e\ \texttt{\}}$. Then we know, by rule [METHOD], that

  $$\begin{array}{l} P; (E, c\,x, E', \overline{d}\,\overline{x}) \vdash e \mathbf{:} d \cdot a' \\ P; (E, c\,x, E', \overline{d}\,\overline{x}) \vdash a \\ a' \sqsubseteq a \end{array}$$

  By IH, we have

  $$\begin{array}{l} P; (E, E'[x := v], (\overline{d}\,\overline{x})[x := v]) \vdash e[x := v] \mathbf{:} d \cdot a'[x := v] \\ P; (E, E'[x := v], (\overline{d}\,\overline{x})[x := v]) \vdash a[x := v] \end{array}$$

  By Lemma 1,

  $$a'[x := v] \sqsubseteq a[x := v]$$

  By rule [METHOD], we may conclude

  $$P; (E, E'[x := v]) \vdash a[x := v]\, d\, m\texttt{(}(\overline{d}\,\overline{x})[x := v]\texttt{)\{}\ e[x := v]\ \texttt{\}}$$

  and hence we obtain our desired result

  $$P; (E, E'[x := v]) \vdash \mathit{meth}[x := v]$$

- Assume $P; (E, c\ x, E') \vdash a$. We proceed by case analysis on the structure of effect a.
  - $a = b$. Then we know by rule [AT BASE] that

    $$P \vdash (E, c\,x, E')$$

    By IH, we have

    $$P \vdash (E, E'[x := v])$$

    Thus by rule [AT BASE] we may conclude

    $$P; (E, E'[x := v]) \vdash b$$

    Since $b = (b[x := v])$, we obtain our desired result

    $$P; E, E'[x := v] \vdash a[x := v]$$

  - $a = \ell\,?\,a_1 : a_2$. Then we know by rule [AT COND] that

    $$\begin{array}{l} P; (E, c\,x, E') \vdash_{\text{lock}} \ell \\ P; (E, c\,x, E') \vdash a_1 \\ P; (E, c\,x, E') \vdash a_2 \end{array}$$

    By IH, we have

    $$\begin{array}{l} P; (E, E'[x := v]) \vdash_{\text{lock}} \ell[x := v] \\ P; (E, E'[x := v]) \vdash a_1[x := v] \\ P; (E, E'[x := v]) \vdash a_2[x := v] \end{array}$$

    By rule [AT COND], we may conclude

    $$P; (E, E'[x := v]) \vdash \ell[x := v]\,?\,a_1[x := v] : a_2[x := v]$$

    Hence we obtain our desired result

    $$P; (E, E'[x := v]) \vdash a[x := v]$$

- Assume $P; (E, c\ x, E') \vdash_{\text{lock}} \ell$. Then we know by rule [LOCK EXP] that

  $$P; (E, c\,x, E') \vdash \ell \mathbf{:} d \cdot \textsc{af}$$

  By IH, we have

  $$P; (E, E'[x := v]) \vdash \ell[x := v] \mathbf{:} d \cdot \textsc{af}[x := v]$$

  By rule [LOCK EXP], we may conclude

  $$P; (E, E'[x := v]) \vdash_{\text{lock}} \ell[x := v]$$

- Assume $P; (E, c\,x, E') \vdash a \uparrow a'$. We proceed by case analysis on the structure of effect a.
  - $a = b$. By rule [LIFT BASE], we know

    $$\begin{array}{l} P \vdash (E, c\,x, E') \\ a' = a \end{array}$$

    By IH, we know

    $$P \vdash (E, E'[x := v])$$

    By rule [LIFT BASE], we may conclude

    $$P; (E, E'[x := v]) \vdash a \uparrow a'$$

    Since both a and a' are combined (basic) effects, we have

    $$\begin{array}{l} a = a[x := v] \\ a' = a'[x := v] \end{array}$$

    By substituting these equalities into our judgment $P; (E, E'[x := v]) \vdash a \uparrow a'$, we obtain our desired result

    $$P; (E, E'[x := v]) \vdash a[x := v] \uparrow a'[x := v]$$

  - $a = \ell\,?\,a_1 : a_2$. We proceed by case analysis on ℓ.
    - $P; (E, c\ x, E') \vdash_{\text{lock}} \ell$. By rule [LIFT GOOD LOCK], we know

      $$\begin{array}{l} P; (E, c\,x, E') \vdash a_1 \uparrow a_1' \\ P; (E, c\,x, E') \vdash a_2 \uparrow a_2' \end{array}$$

      By IH, we have

      $$\begin{array}{l} P; (E, E'[x := v]) \vdash_{\text{lock}} \ell[x := v] \\ P; (E, E'[x := v]) \vdash a_1[x := v]] \uparrow a_1'[x := v] \\ P; (E, E'[x := v]) \vdash a_2[x := v] \uparrow a_2'[x := v] \end{array}$$

      By rule [LIFT GOOD LOCK], we may conclude

      $$\begin{array}{l} P, (E, E'[x := v]) \vdash \\ \quad \ell[x := v]\,?\,a_1[x := v] : a_2[x := v] \uparrow \\ \quad \ell[x := v]\,?\,a_1'[x := v] : a_2'[x := v] \end{array}$$

      Hence we obtain our desired result

      $$P; (E, E'[x := v]) \vdash a[x := v] \uparrow a'[x := v]$$

    - $P; (E, c\ x, E') \not\vdash_{\text{lock}} \ell$. Then by rule [LIFT BAD LOCK] we know

      $$\begin{array}{l} P; (E, c\,x, E') \vdash a_1 \uparrow a_1' \\ P; (E, c\,x, E') \vdash a_2 \uparrow a_2' \end{array}$$

      Since the lock judgment does not hold, ℓ must contain some non-constant expression e. Thus $\ell[x := v]$ would still contain e and hence the locking judgment will not hold for $\ell[x := v]$. By IH, we have

$$\begin{array}{l} P; (E, E'[x := v]) \nvdash_{\text{lock}} \ell[x := v] \\ P; (E, E'[x := v]) \vdash a_1[x := v]] \uparrow a'_1[x := v] \\ P; (E, E'[x := v]) \vdash a_2[x := v] \uparrow a'_2[x := v] \end{array}$$

Then by rule [LIFT BAD LOCK], we may conclude

$$\begin{array}{l} P, (E, E'[x := v]) \vdash \\ \quad (\ell[x := v] \mathbin{?} a_1[x := v] : a_2[x := v]) \uparrow \\ \quad (a'_1[x := v] \sqcup a'_2[x := v]]) \end{array}$$

Hence we obtain our desired result

$$P; (E, E'[x := v]) \vdash a[x := v] \uparrow a'[x := v]$$

- Assume $P; (E, c\,x, E') \vdash e : d \cdot a$. We proceed by induction over the derivation of this judgment.
  - $e \equiv y$. Then our assumption is

    $$P; (E, c\,x, E') \vdash y : d \cdot a$$

    Since y is a variable, this assumption is derivable only through rule [EXP VAR]; hence we know

    $$\begin{array}{l} P \vdash (E, c\,x, E') \\ a = \mathrm{AF} \end{array}$$

    We proceed by case analysis of membership of $(d\,y)$ in the environment; one of three cases holds:

    $$\begin{array}{l} (d\,y) \in E \\ (d\,y) = (c\,x) \\ (d\,y) \in E' \end{array}$$

    - $(d\,y) \in E$. By IH, we have

      $$P \vdash E, E'[x := v]$$

      so we may still conclude, by [EXP VAR],

      $$P; (E, E'[x := v]) \vdash y : d \cdot \mathrm{AF}$$

      Since $y \neq x$ and AF is a combined (basic) effect, we have

      $$P; (E, E'[x := v]) \vdash y[x := v] : d \cdot \mathrm{AF}[x := v]$$

      Hence we obtain our desired result

      $$P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v]$$

    - $(d\,y) = (c\,x)$. By assumption, we have

      $$P; E \vdash v : c \cdot \mathrm{AF}$$

      By Lemma 12, we can strengthen the environment, so we have

      $$P; (E, E'[x := v]) \vdash v : c \cdot \mathrm{AF}$$

      Since $v = x[x := v]$ and $\mathrm{AF} = \mathrm{AF}[x := v]$, by substituting we have

      $$P; (E, E'[x := v)] \vdash x[x := v] : c \cdot \mathrm{AF}[x := v]$$

      Hence we obtain our desired result

      $$P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v]$$

    - $(d\,y) \in E'$. By IH, we have

      $$P \vdash E, E'[x := v]$$

      Furthermore, $(d\ y) \in E'[x := v]$, since $y \neq x$ and type d does not capture x. By rule [EXP VAR], we may conclude

      $$P; (E, E'[x := v]) \vdash y : d \cdot \mathrm{AF}$$

      Since $y = y[x := v]$ and $\mathrm{AF} = \mathrm{AF}[x := v]$, we have

      $$P; (E, E'[x := v]) \vdash y[x := v] : d \cdot \mathrm{AF}[x := v]$$

      Hence we obtain our desired result

      $$P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v]$$

  - $e \equiv \texttt{null}$. Then our assumption is

    $$P; (E, c\,x, E') \vdash \texttt{null} : d \cdot a$$

    This assumption is derivable only with rule [EXP NULL]; hence we know

    $$\begin{array}{l} P \vdash (E, c\,x, E') \\ P \vdash d \\ a = \mathrm{AF} \end{array}$$

    By IH, we have

    $$P \vdash (E, E'[x := v])$$

    By rule [EXP NULL], we may conclude

    $$P; (E, E'[x := v]) \vdash \texttt{null} : d \cdot \mathrm{AF}$$

    Since $\texttt{null} = \texttt{null}[x := v]$ and $\mathrm{AF} = \mathrm{AF}[x := v]$, we have

    $$P; (E, E'[x := v]) \vdash \texttt{null}[x := v] : d \cdot \mathrm{AF}[x := v]$$

    Hence we obtain our desired result

    $$P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v]$$

  - $e \equiv \rho$. Then our assumption is

    $$P; (E, c\,x, E') \vdash \rho : d \cdot a$$

    This assumption is derivable only with rule [EXP ADDR]; hence we know

    $$\begin{array}{l} P \vdash (E', c\,x, E'') \\ (E, c\,x, E') = (E'', d\,\rho, E''') \\ \rho \in \mathit{Addr}_d \\ a = \mathrm{AF} \end{array}$$

    Since ρ is an address, and x is a variable, it must be, by the structure of environments (where for some state σ, E_σ is the environment initially passed), that $\rho \in E$. Hence we have

    $$(E, c\,x, E') = (E_1, d\,\rho, E_2, c\,x, E_3)$$

    By IH, we have

    $$P \vdash (E_1, d\,\rho, E_2, E_3[x := v])$$

    By rule [EXP ADDR], we may conclude

    $$P; (E_1, d\,\rho, E_2, E_3[x := v]) \vdash \rho : d \cdot \mathrm{AF}$$

    Since $\rho = \rho[x := v]$ and $\mathrm{AF} = \mathrm{AF}[x := v]$, we have

    $$P; (E_1, d\,\rho, E_2, E_3[x := v]) \vdash \rho[x := v] : d \cdot \mathrm{AF}[x := v]$$

    Hence we obtain our desired result

    $$P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v]$$

  - $e \equiv e'{}_\gamma f$. Then our assumption is

    $$P; (E, c\,x, E') \vdash e'{}_\gamma f : d \cdot a$$

    This assumption is derivable only through rules [EXP REF] and [EXP REF RACE]; hence we know

    $$\begin{array}{l} P; (E, c\,x, E') \vdash e' : d_o \cdot a_o \\ \texttt{class}\ d_o\ \texttt{\{}\ \ldots d\,f \ldots \texttt{\}} \in P \\ a = (a_o; a_f) \end{array}$$

where a_f is one of four cases

$$\begin{array}{ll}
\text{AM} & \text{if } f \in \textit{Normal} \text{ and } \gamma = \,. \\
\text{AF} & \text{if } f \in \textit{Final} \text{ and } \gamma = \,. \\
\text{AN} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = \,. \\
\text{CL} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = \,..
\end{array}$$

By IH, we have

$$P; (E, E'[x := v]) \vdash e'[x := v] \texttt{ : } d_o \cdot a_o[x := v]$$

By rule [EXP REF]/[EXP REF RACE], we may conclude

$$\begin{array}{l}
P; (E, E'[x := v]) \vdash e'[x := v]_\gamma f \texttt{ : } d \cdot a' \\
a' = (a_o[x := v]; a_f)
\end{array}$$

Since $f = f[x := v]$ (f is not a variable), and $a_f = a_f[x := v]$ (a_f is a combined (basic) effect for all cases), we have

$$\begin{array}{l}
P; (E, E'[x := v]) \vdash \\
\quad (e'{}_\gamma f)[x := v] \texttt{ : } d \cdot (a_o; a_f)[x := v]
\end{array}$$

Hence we obtain our desired result

$$P; (E, E'[x := v]) \vdash e[x := v] \texttt{ : } d \cdot a[x := v]$$

- $e \equiv e'{}_\gamma f \texttt{ = } e''$. Then our assumption is

$$P; (E, c\,x, E') \vdash e'{}_\gamma f \texttt{ = } e'' \texttt{ : } d \cdot a$$

This assumption is derivable only through rules [EXP ASSIGN] and [EXP ASSIGN RACE]; hence we know

$$\begin{array}{l}
P; (E, c\,x, E') \vdash e' \texttt{ : } d_o \cdot a_o \\
P; (E, c\,x, E') \vdash e'' \texttt{ : } d \cdot a_v \\
\texttt{class } d_o \texttt{ \{ } \ldots d\, f \ldots \texttt{ \} } \in P \\
a = (a_o; a_v; a_f)
\end{array}$$

where a_f is one of three cases

$$\begin{array}{ll}
\text{AM} & \text{if } f \in \textit{Normal} \text{ and } \gamma = \,. \\
\text{AN} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = \,. \\
\text{CL} & \text{if } f \in \textit{Volatile} \text{ and } \gamma = \,..
\end{array}$$

By IH, we have

$$\begin{array}{l}
P; (E, E'[x := v]) \vdash e'[x := v] \texttt{ : } d_o \cdot a_o[x := v] \\
P; (E, E'[x := v]) \vdash e''[x := v] \texttt{ : } d \cdot a_v[x := v]
\end{array}$$

By rule [EXP ASSIGN]/[EXP ASSIGN RACE], we may conclude

$$\begin{array}{l}
P; (E, E'[x := v]) \vdash e'[x := v]_\gamma f \texttt{ = } e''[x := v] \texttt{ : } d \cdot a' \\
a' = (a_o[x := v]; a_v[x := v]; a_f)
\end{array}$$

Since $f = f[x := v]$ (f is not a variable), and $a_f = a_f[x := v]$ (a_f is a combined (basic) effect for all cases), we have

$$\begin{array}{l}
P; (E, E'[x := v]) \vdash \\
\quad (e'{}_\gamma f \texttt{ = } e'')[x := v] \texttt{ : } d \cdot (a_o; a_v; a_f)[x := v]
\end{array}$$

Hence we obtain our desired result

$$P; (E, E'[x := v]) \vdash e[x := v] \texttt{ : } d \cdot a[x := v]$$

- $e \equiv e'{}_\gamma m\texttt{(}e_{1..n}\texttt{)}$. Then our assumption is

$$P; (E, c\,x, E') \vdash e'{}_\gamma m\texttt{(}e_{1..n}\texttt{)} \texttt{ : } d \cdot a$$

This assumption is derivable only through rule [EXP INVOKE]; hence we know

$$\begin{array}{l}
P; (E, c\,x, E') \vdash e' \texttt{ : } d_o \cdot a_o \\
\texttt{class } d_o \texttt{ \{ } \ldots \textit{meth} \ldots \texttt{\}} \\
\textit{meth} = a''\, d\, m\texttt{(}d_i\; x_i{}^{\,i \in 1..n}\texttt{)\{ } e'' \texttt{ \}} \\
P; (E, c\,x, E') \vdash e_i \texttt{ : } d_i \cdot a_i \qquad \forall i \in 1..n \\
P; (E, c\,x, E') \vdash a''[\texttt{this} := e', x_i := e_i{}^{\,i \in 1..n}] \uparrow a''' \\
a''' \sqsubseteq \text{AN} \\
a = a_o; a_1; \ldots; a_n; [\![\gamma]\!]; a'''
\end{array}$$

By IH, we have

$$\begin{array}{l}
P; (E, E'[x := v]) \vdash e'[x := v] \texttt{ : } d_o \cdot a_o[x := v] \\
P; (E, E'[x := v]) \vdash e_i[x := v] \texttt{ : } d_i \cdot a_i[x := v] \quad \forall i \in 1..n \\
P; (E, E'[x := v]) \vdash \\
\quad a''[\texttt{this} := e', x_i := e_i{}^{\,i \in 1..n}][x := v] \uparrow a'''[x := v]
\end{array}$$

Also from Lemma 1 and $a''' \sqsubseteq \text{AN}$, we have

$$(a'''[x := v]) \sqsubseteq (\text{AN}[x := v]) = \text{AN}$$

By rule [EXP INVOKE], we may conclude

$$\begin{array}{l}
P; (E, E'[x := v]) \vdash \\
\quad e'[x := v]_\gamma m\texttt{(}e_i[x := v]{}^{\,i \in 1..n}\texttt{)} \texttt{ : } d \cdot a' \\
a' = (a_o[x := v]; a_1[x := v]; \ldots; a_n[x := v]; a'''[x := v])
\end{array}$$

Hence we obtain our desired result

$$P; (E, E'[x := v]) \vdash e[x := v] \texttt{ : } d \cdot a[x := v]$$

- $e \equiv e'{}_\gamma m\texttt{\#(}e_{1..n}\texttt{)}$. Similar to previous case.
- $e \equiv \texttt{new } d \texttt{ (}e_{1..n}\texttt{)}$. Then our assumption is

$$P; (E, c\,x, E') \vdash \texttt{new } d \texttt{ (}e_{1..n}\texttt{)} \texttt{ : } d \cdot a$$

This assumption is derivable only through rule [EXP NEW]; hence we know

$$\begin{array}{l}
\texttt{class } d \texttt{ \{ } d_i\; x_i{}^{\,i \in 1..n} \ldots \texttt{\}} \in P \\
P; (E, c\,x, E') \vdash e_i \texttt{ : } d_i \cdot a_i \qquad \forall i \in 1..n \\
a = (a_1; \ldots; a_n; \text{AM})
\end{array}$$

By IH, we have

$$P; (E, E'[x := v]) \vdash e_i[x := v] \texttt{ : } d_i \cdot a_i[x := v] \quad \forall i \in 1..n$$

By rule [EXP NEW], we may conclude

$$\begin{array}{l}
P; (E, E'[x := v]) \vdash \texttt{new } d \texttt{ (}e_i[x := v]{}^{\,i \in 1..n}\texttt{)} \texttt{ : } d \cdot a' \\
a' = (a_1[x := v]; \ldots; a_n[x := v]; \text{AM})
\end{array}$$

Since we have the following equalities,

$$\begin{array}{l}
\text{AM} = \text{AM}[x := v] \\
\texttt{new } d \texttt{ (}e_i[x := v]{}^{\,i \in 1..n}\texttt{)} \\
\quad = (\texttt{new } d \texttt{ (}e_i{}^{\,i \in 1..n}\texttt{)})[x := v] \\
\quad = e[x := v] \\
(a_1[x := v]; \ldots; a_n[x := v]; \text{AM}[x := v]) \\
\quad = (a_1; \ldots; a_n; \text{AM})[x := v] \\
\quad = a[x := v]
\end{array}$$

we obtain our desired result

$$P; (E, E'[x := v]) \vdash e[x := v] \texttt{ : } d \cdot a[x := v]$$

- $e \equiv \ell_\gamma \texttt{sync } e'$. Then our assumption is

$$P; (E, c\,x, E') \vdash \ell_\gamma \texttt{sync } e' \texttt{ : } d \cdot a$$

This assumption is derivable only through rule [EXP SYNC]; hence we know

$$\begin{array}{l}
P; (E, c\,x, E') \vdash_{\text{lock}} \ell \\
P; (E, c\,x, E') \vdash e' \texttt{ : } d \cdot a_b \\
a = \mathcal{S}(\ell, \gamma, a_b)
\end{array}$$

By IH, we have

$$\begin{array}{l}
P; (E, E'[x := v]]) \vdash_{\text{lock}} \ell[x := v] \\
P; (E, E'[x := v]) \vdash e'[x := v] \texttt{ : } d \cdot a_b[x := v]
\end{array}$$

By rule [EXP SYNC], we may conclude

$$\begin{array}{l}
P; E, E'[x := v] \vdash \ell[x := v]_\gamma \texttt{sync } e'[x := v] \texttt{ : } d \cdot a' \\
a' = \mathcal{S}(\ell[x := v], \gamma, a_b[x := v])
\end{array}$$

We proceed by case analysis on the structure of effect a_b.

- $a_b = b$. In this case, since $b' = b'[x := v]$ for any combined (basic) effect, including a_b, AR, and AL, we have

$$\begin{aligned} a' &= \mathcal{S}(\ell[x := v], \gamma, a_b[x := v]) \\ &= \ell[x := v] \mathbin{?} a_b[x := v] : ([\![\gamma]\!]; \textsc{ar}; a_b[x := v]; \textsc{al}) \\ &= (\ell \mathbin{?} a_b : ([\![\gamma]\!]; \textsc{ar}; a_b; \textsc{al}))[x := v] \\ &= a[x := v] \end{aligned}$$

- $a_b = \ell \mathbin{?} a_1 : a_2$. Then applying the substitution, we get

$$a_b[x := v] = \ell[x := v] \mathbin{?} a_1[x := v] : a_2[x := v]$$

Proceeding with a',

$$\begin{aligned} & a' \\ &= \mathcal{S}(\ell[x := v], \gamma, a_b[x := v]) \\ &= \mathcal{S}(\ell[x := v], \gamma, \ell[x := v] \mathbin{?} a_1[x := v] : a_2[x := v]) \\ &= \mathcal{S}(\ell[x := v], \gamma, a_1[x := v]) \\ &= \mathcal{S}(\ell, \gamma, a_1)[x := v] \\ &= \mathcal{S}(\ell, \gamma, a_b)[x := v] \\ &= a[x := v] \end{aligned}$$

- $a_b = \ell' \mathbin{?} a_1 : a_2$, where $\ell' \neq \ell$. Then applying the substitution, we get

$$a_b[x := v] = \ell'[x := v] \mathbin{?} a_1[x := v] : a_2[x := v]$$

Let $\theta(E) = E[x := v]$, for any expression or effect E. Proceeding with a',

$$\begin{aligned} & a' \\ &= \mathcal{S}(\theta(\ell), \gamma, \theta(a_b)) \\ &= \mathcal{S}(\theta(\ell), \gamma, \theta(\ell') \mathbin{?} \theta(a_1) : \theta(a_2)) \\ &= \theta(\ell') \mathbin{?} \mathcal{S}(\theta(\ell), \gamma, \theta(a_1)) : \mathcal{S}(\theta(\ell), \gamma, \theta(a_2)) \\ &= \theta(\ell') \mathbin{?} \theta(\mathcal{S}(\ell, \gamma, a_1)) : \theta(\mathcal{S}(\ell, \gamma, a_2)) \\ &= \theta(\ell' \mathbin{?} \mathcal{S}(\ell, \gamma, a_1) : \mathcal{S}(\ell, \gamma, a_2)) \\ &= \theta(\mathcal{S}(\ell, \gamma, \ell' \mathbin{?} a_1 : a_2)) \\ &= a[x := v] \end{aligned}$$

Hence we obtain our desired result

$$P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v]$$

- $e \equiv \texttt{in-sync}\ \rho\ e'$. Then our assumption is

$$P; (E, c\,x, E') \vdash \texttt{in-sync}\ \rho\ e' : d \cdot a$$

This assumption is derivable only through rule [EXP INSYNC]; hence we know

$$\begin{aligned} & P; (E, c\,x, E') \vdash_{\text{lock}} \rho \\ & P; (E, c\,x, E') \vdash e' : d \cdot a_b \\ & a = \mathcal{SI}(\rho, a_b) \end{aligned}$$

By IH, we have

$$\begin{aligned} & P; (E, E'[x := v]) \vdash_{\text{lock}} \rho[x := v] \\ & P; (E, E'[x := v]) \vdash e'[x := v] : d \cdot a_b[x := v] \end{aligned}$$

Then by rule [EXP INSYNC] we may conclude

$$\begin{aligned} & P; (E, E'[x := v]) \vdash \texttt{in-sync}\ \rho[x := v]\ e'[x := v] : d \cdot a' \\ & a' = \mathcal{SI}(\rho[x := v], a_b[x := v]) \end{aligned}$$

We proceed by case analysis on the structure of effect a_b.

- $a_b = b$. In this case, since $b' = b'[x := v]$ for any combined (basic) effect, including a_b, ρ, and AL, we have

$$\begin{aligned} a' &= \mathcal{SI}(\rho[x := v], a_b[x := v]) \\ &= a_b[x := v]; \textsc{al} \\ &= (a_b; \textsc{al})[x := v] \\ &= a[x := v] \end{aligned}$$

- $a_b = \rho \mathbin{?} a_1 : a_2$. Then applying the substitution, we get

$$a_b[x := v] = \rho[x := v] \mathbin{?} a_1[x := v] : a_2[x := v]$$

Proceeding with a',

$$\begin{aligned} & a' \\ &= \mathcal{SI}(\rho[x := v], a_b[x := v]) \\ &= \mathcal{SI}(\rho[x := v], \rho[x := v] \mathbin{?} a_1[x := v] : a_2[x := v]) \\ &= \mathcal{SI}(\rho[x := v], a_1[x := v]) \\ &= \mathcal{SI}(\rho, a_1)[x := v] \\ &= \mathcal{SI}(\rho, a_b)[x := v] \\ &= a[x := v] \end{aligned}$$

- $a_b = \rho' \mathbin{?} a_1 : a_2$, where $\rho' \neq \rho$. Then applying the substitution, we get

$$a_b[x := v] = \rho'[x := v] \mathbin{?} a_1[x := v] : a_2[x := v]$$

Let $\theta(E) = E[x := v]$, for any expression or effect E. Proceeding with a',

$$\begin{aligned} & a' \\ &= \mathcal{SI}(\theta(\rho), \theta(a_b)) \\ &= \mathcal{SI}(\theta(\rho), \theta(\rho') \mathbin{?} \theta(a_1) : \theta(a_2)) \\ &= \theta(\rho') \mathbin{?} \mathcal{SI}(\theta(\rho), \theta(a_1)) : \mathcal{SI}(\theta(\rho), \theta(a_2)) \\ &= \theta(\rho') \mathbin{?} \theta(\mathcal{SI}(\rho, a_1)) : \theta(\mathcal{SI}(\rho, a_2)) \\ &= \theta(\rho' \mathbin{?} \mathcal{SI}(\rho, a_1) : \mathcal{SI}(\rho, a_2)) \\ &= \theta(\mathcal{SI}(\rho, \rho' \mathbin{?} a_1 : a_2)) \\ &= a[x := v] \end{aligned}$$

Hence we obtain our desired result

$$P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v]$$

- $e \equiv \texttt{fork}\ e'$. Then our assumption is

$$P; (E, c\,x, E') \vdash \texttt{fork}\ e' : d \cdot a$$

This assumption is derivable only through rule [EXP FORK]; hence we know

$$\begin{aligned} & P; (E, c\,x, E') \vdash e' : d_b \cdot a_b \\ & d = \texttt{Unit} \\ & a = \textsc{al} \end{aligned}$$

By IH, we have

$$P; (E, E'[x := v]) \vdash e'[x := v] : d_b \cdot a_b[x := v]$$

Then by rule [EXP FORK] we have

$$\begin{aligned} & P; (E, E'[x := v]) \vdash \texttt{fork}\ e'[x := v] : d \cdot a' \\ & a' = \textsc{al} \end{aligned}$$

Since $\textsc{al} = \textsc{al}[x := v]$ and $\texttt{fork}\ (e'[x := v]) = (\texttt{fork}\ e')[x := v]$, we obtain our desired result

$$P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v]$$

- $e \equiv \texttt{let}\ x_1\ e_1\ \texttt{in}\ e_2$. Then our assumption is

$$P; (E, c\,x, E') \vdash \texttt{let}\ x_1\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : d \cdot a$$

This assumption is derivable only through rule [EXP LET]; hence we know

$$\begin{aligned} & P; (E, c\,x, E') \vdash e_1 : c_1 \cdot a_1 \\ & P; (E, c\,x, E', c_1\,x_1) \vdash e_2 : d \cdot a_2 \\ & P; (E, c\,x, E') \vdash a_2[x_1 := e_1] \uparrow a_2' \\ & a = (a_1; a_2') \end{aligned}$$

By well-formedness of the environment, and because types do not capture variables, we have

$$(c_1\ x_1)[x := v] = (c_1\ x_1)$$

By IH, we have

$$\begin{array}{l} P; (E, E'[x := v]) \vdash \\ \quad e_1[x := v] : c_1 \cdot a_1[x := v] \\ P; (E, E'[x := v], (c_1\ x_1)[x := v]) \vdash \\ \quad e_2[x := v] : d \cdot a_2[x := v] \\ P; (E, E'[x := v]) \vdash a_2[x_1 := e_1][x := v] \uparrow a_2'[x := v] \end{array}$$

In the last judgment, since $x \neq x_1$, we have

$$\begin{array}{l} P; (E, E'[x := v]) \vdash \\ \quad (a_2[x := v])[x_1 := e_1[x := v]] \uparrow a_2'[x := v] \end{array}$$

By rule [EXP LET], we may conclude

$$\begin{array}{l} P; (E, E'[x := v]) \vdash \\ \quad \texttt{let}\ x_1\ \texttt{=}\ (e_1[x := v])\ \texttt{in}\ (e_2[x := v]) : d \cdot a' \\ a' = (a_1[x := v]; a_2'[x := v]) = a[x := v] \end{array}$$

Hence we obtain our desired result

$$P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v]$$

- $e \equiv \texttt{if}\ e_1\ e_2\ e_3$. Then our assumption is

$$P; (E, c\,x, E') \vdash \texttt{if}\ e_1\ e_2\ e_3 : d \cdot a$$

This assumption is derivable only through rule [EXP IF]; hence we know

$$\begin{array}{l} P; (E, c\,x, E') \vdash e_1 : d' \cdot a_1 \\ P; (E, c\,x, E') \vdash e_2 : d \cdot a_2 \\ P; (E, c\,x, E') \vdash e_3 : d \cdot a_3 \\ a = (a_1; (a_2 \sqcup a_3)) \end{array}$$

By IH, we have

$$\begin{array}{l} P; (E, E'[x := v]) \vdash e_1[x := v] : d' \cdot a_1[x := v] \\ P; (E, E'[x := v]) \vdash e_2[x := v] : d \cdot a_2[x := v] \\ P; (E, E'[x := v]) \vdash e_3[x := v] : d \cdot a_3[x := v] \end{array}$$

By rule [EXP IF], we may conclude

$$\begin{array}{l} P; (E, E'[x := v]) \vdash \\ \quad \texttt{if}\ e_1[x := v]\ e_2[x := v]\ e_3[x := v] : d \cdot a' \\ a' = (a_1[x := v]; (a_2[x := v] \sqcup a_3[x := v])) \end{array}$$

Since the following equalities hold

$$\begin{array}{l} (\texttt{if}\ e_1[x := v]\ e_2[x := v]\ e_3[x := v]) \\ \quad = (\texttt{if}\ e_1\ e_2\ e_3)[x := v] \\ \quad = e[x := v] \\ a' = (a_1[x := v]; (a_2[x := v] \sqcup a_3[x := v])) \\ \quad = (a_1; (a_2 \sqcup a_3)[x := v] \\ \quad = a[x := v] \end{array}$$

by substituting, we obtain our desired result

$$P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v]$$

- $e \equiv \texttt{while}\ e_1\ e_2$. Then our assumption is

$$P; (E, c\,x, E') \vdash \texttt{while}\ e_1\ e_2 : d \cdot a$$

This assumption is derivable only with rule [EXP WHILE]; hence we know

$$\begin{array}{l} P; (E, c\,x, E') \vdash e_1 : c_1 \cdot a_1 \\ P; (E, c\,x, E') \vdash e_2 : c_2 \cdot a_2 \\ a = (a_1; (a_2; a_1)^*) \end{array}$$

By IH, we have

$$\begin{array}{l} P; (E, E'[x := v]) \vdash e_1[x := v] : c_1 \cdot a_1[x := v] \\ P; (E, E'[x := v]) \vdash e_2[x := v] : c_2 \cdot a_2[x := v] \end{array}$$

By rule [EXP WHILE], we may conclude

$$\begin{array}{l} P; (E, E'[x := v]) \vdash \texttt{while}\ e_1[x := v]\ e_2[x := v] : d \cdot a' \\ a' = (a_1[x := v]; (a_2[x := v]; a_1[x := v])^*) \end{array}$$

Since the following equalities hold

$$\begin{array}{l} \texttt{while}\ e_1[x := v]\ e_2[x := v] \\ \quad = (\texttt{while}\ e_1\ e_2)[x := v] \\ \quad = e[x := v] \\ a' = (a_1[x := v]; (a_2[x := v]; a_1[x := v])^*) \\ \quad = (a_1; (a_2; a_1)^*)[x := v] \\ \quad = a[x := v] \end{array}$$

by substituting, we obtain our desired result

$$P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v]$$

LEMMA 7 (Well-Typed Effects). *If $P; E \vdash e : c \cdot a$ then $P; E \vdash a$.*

Proof By induction on derivation of $P; E \vdash e : c \cdot a$. For cases involving the following rules:

- [EXP VAR]: $e \equiv x$
- [EXP NULL]: $e \equiv \texttt{null}$
- [EXP ADDR]: $e \equiv \rho$
- [EXP REF]: $e \equiv e'.f$
- [EXP REF RACE]: $e \equiv e'{}_\gamma f$
- [EXP ASSIGN]: $e \equiv e'.f\ \texttt{=}\ e''$
- [EXP ASSIGN RACE]: $e \equiv e'{}_\gamma f\ \texttt{=}\ e''$
- [EXP NEW]: $e \equiv \texttt{new}\ c\ (e_{1..n})$
- [EXP FORK]: $e \equiv \texttt{fork}\ e'$
- [EXP IF]: $e \equiv \texttt{if}\ e_1\ e_2\ e_3$
- [EXP WHILE]: $e \equiv \texttt{while}\ e_1\ e_2$

In these cases, a is either a combined (basic) effect or the product of sequentially composition, iterative closure, or join of inductively well-typed sub-effects. We proceed by case analysis on the structure of a.

- $a = b$. In this case, the rule [AT BASE] applies, and we may conclude

$$P; E \vdash a$$

- $a = \ell\,?\,a_1 : a_2$. In this case, all lock expressions contained in a, including ℓ, are constant, by IH. Sequential composition, iterative closure, and join operations do not change or add lock expressions. Hence, by rule [AT COND], we may conclude

$$P; E \vdash a$$

There are five remaining cases to consider.

- [EXP SYNC] As an assumption, we have

$$P; E \vdash \ell_\gamma\texttt{sync}\ e'\ : c \cdot a$$

This assumption may be derived only through rule [EXP SYNC]; hence we know

$$\begin{array}{l} P; E \vdash_{\text{lock}} \ell \\ P; E \vdash e' : c \cdot a' \end{array}$$

By IH, we have $P; E \vdash a'$. We proceed by case analysis on the structure of effect a'.

  - $a' = b$. In this case, by the first case of function $\mathcal{S}$, we know

$$a = \ell\,?\,a' : ([\![\gamma]\!]; \text{AR}; a'; \text{AL})$$

Since ℓ, the only lock expression in a, is constant, and since both branches in a are well-typed effects, by rule [AT COND] we may conclude

$$P; E \vdash a$$

  - $a' = \ell\,?\,a_1 : a_2$. In this case, by the second case of function $\mathcal{S}$, we know

    $$a = \mathcal{S}(\ell, \gamma, a_1)$$

    Since by IH a' is a well-typed effect, we know that a_1 is also well-typed. Hence all locks in a_1 are constant. Furthermore, ℓ is a constant lock expression. Thus any lock expression in $\mathcal{S}(\ell, \gamma, a_1)$ is constant. By rule [AT COND], we may conclude

    $$P; E \vdash a$$

  - $a' = \ell'\,?\,a_1 : a_2$ where $\ell \neq \ell'$. In this case, by the third case of function $\mathcal{S}$, we know

    $$a = \ell'\,?\,\mathcal{S}(\ell, \gamma, a_1) : \mathcal{S}(\ell, \gamma, a_2)$$

    Since by IH a' is a well-typed effect, we know that ℓ' is constant, and that a_1 and a_2 are well-typed. Thus any lock expression in $\mathcal{S}(\ell, \gamma, a_1)$ and $\mathcal{S}(\ell, \gamma, a_2)$ is constant. By rule [AT COND], we may conclude

    $$P; E \vdash a$$

- [EXP INSYNC] As an assumption, we have

  $$P; E \vdash \texttt{in-sync}\ \rho\, e' \mathbf{:}\ c \cdot a$$

  This assumption may be derived only through rule [EXP INSYNC]; hence we know

  $$\begin{array}{l} P; E \vdash_{\text{lock}} \rho \\ P; E \vdash e' \mathbf{:}\ c \cdot a' \end{array}$$

  By IH, we have

  $$P; E \vdash a'$$

  We proceed by case analysis on the structure of effect a'.

  - $a' = b$. In this case, by the first case of function $\mathcal{SI}$, we know

    $$a = a'; \text{AL}$$

    Since a' is a combined (basic) effect, $a = a'; \text{AL}$ is also a combined effect. Then by rule [AT BASE] we may conclude

    $$P; E \vdash a$$

  - $a' = \rho\,?\,a_1 : a_2$. In this case, by the second case of function $\mathcal{SI}$, we know

    $$a = \mathcal{SI}(\rho, a_1)$$

    Since a' is well-typed, inductively a_1 is also well-typed. Thus we know any lock expression contained in $\mathcal{SI}(\rho, a_1)$ is constant. Thus by rule [AT COND], we may conclude

    $$P; E \vdash a$$

    If $\mathcal{SI}(\rho, a_1)$ contains no lock expression, then by rule [AT BASE], we may similarly conclude

    $$P; E \vdash a$$

  - $a' = \rho'\,?\,a_1 : a_2$ where $\rho \neq \rho'$. In this case, by the third case of function $\mathcal{SI}$, we know

    $$a = \rho'\,?\,\mathcal{SI}(\rho, a_1) : \mathcal{SI}(\rho, a_2)$$

    By IH, a' is well-typed; hence inductively ρ' is a constant lock expression and a_1 and a_2 are well-typed. Since $\mathcal{SI}$ may at most introduce ρ' as a lock expression and potentially reduce the number of lock expressions, we may conclude by rule [AT COND] that

    $$P; E \vdash a$$

- [EXP INVOKE] As an assumption, we have

  $$P; E \vdash e'{}_{\gamma} m\texttt{(}e_{1..n}\texttt{)}\ \mathbf{:}\ c \cdot a$$

  This assumption is derivable only through rule [EXP INVOKE]; hence we know

  $$\begin{array}{l} P; E \vdash e' \mathbf{:}\ c' \cdot a' \\ \texttt{class}\ c'\ \texttt{\{}\ \ldots \mathit{meth} \ldots \texttt{\}} \in P \\ \mathit{meth} = a''\ c\ m\texttt{(}c_i\ x_i\ {}^{i \in 1..n}\texttt{)}\ \texttt{\{}\ e''\texttt{\}} \\ P; E \vdash e_i \mathbf{:}\ c_i \cdot a_i \qquad \forall i \in 1..n \\ P; E \vdash a''[\texttt{this} := e', x_i := e_i\ {}^{i \in 1..n}] \uparrow a''' \\ a''' \sqsubseteq \text{AN} \end{array}$$

  By IH, we have

  $$\begin{array}{l} P; E \vdash a' \\ P; E \vdash a_i \quad \forall i \in 1.n \\ P; E \vdash a''' \end{array}$$

  In particular, for the last judgment, if any lock expression ℓ' in $a''[\texttt{this} := e', x_i := e_i\ {}^{i \in 1..n}]$ is not constant, the lift will eliminate ℓ'. Then by rule [EXP INVOKE], we have

  $$a = (a'; a_1; \ldots; a_n; [\![\gamma]\!]; a''')$$

  Since sequential composition does not change lock expressions or add new lock expressions, we may conclude all lock expressions in a (if any) are constant. Hence by rules [AT BASE] and [AT COND], we may conclude

  $$P; E \vdash a$$

- [EXP INVOKE COMPOUND] As an assumption, we have

  $$P; E \vdash e'{}_{\gamma} m\texttt{\#(}e_{1..n}\texttt{)}\ \mathbf{:}\ c \cdot a$$

  The argument proceeds in a similar manner to the previous case.

- [EXP LET] As an assumption, we have

  $$P; E \vdash \texttt{let}\ x\ \texttt{=}\ e'\ \texttt{in}\ e'' \mathbf{:}\ c \cdot a$$

  This assumption is derivable only through rule [EXP LET]; hence we know

  $$\begin{array}{l} P; E \vdash e' \mathbf{:}\ c' \cdot a' \\ P; (E, c'\ x) \vdash e'' \mathbf{:}\ c \cdot a'' \\ P; E \vdash a''[x := e'] \uparrow a''' \end{array}$$

  By IH, we know

  $$\begin{array}{l} P; E \vdash a' \\ P; E \vdash a''' \end{array}$$

  In particular, for the last judgment, if any lock expression ℓ' in $a''[x := e']$ is not constant, the lift will eliminate ℓ'. By rule [EXP LET], we may conclude

  $$a = (a'; a''')$$

  Since sequential composition does not change lock expressions or add new lock expressions, we may conclude al lock expressions in a (if any) are constant. Hence by rules [AT BASE] and [AT COND], we may conclude

  $$P; E \vdash a$$

LEMMA 8 (Well-Typed Lift). *If $P; E \vdash a \uparrow a'$ then $P; E \vdash a'$.*

Proof By induction on the structure of effect a.

- $a = b$. In this case, by rule [LIFT BASE], we know

  $$\begin{array}{c} P \vdash E \\ a' = a \end{array}$$

  Then by rule [AT BASE], we may obtain our desired result

  $$P; E \vdash a'$$

- $a = \ell \,?\, a_1 : a_2$. We proceed by case analysis on the constancy of ℓ.
  - $P; E \vdash_{\text{lock}} \ell$. Then $P; E \vdash a \uparrow a'$ is only derivable with rule [LIFT GOOD LOCK]; hence we know

    $$\begin{gathered} P; E \vdash a_1 \uparrow a_1' \\ P; E \vdash a_2 \uparrow a_2' \\ a' = (\ell \,?\, a_1' : a_2') \end{gathered}$$

    By IH, we have

    $$\begin{gathered} P; E \vdash a_1' \\ P; E \vdash a_2' \end{gathered}$$

    By rule [AT COND], we may conclude

    $$P; E \vdash a'$$

  - $P; E \not\vdash_{\text{lock}} \ell$. Then $P; E \vdash a \uparrow a'$ is only derivable with rule [LIFT BAD LOCK]; hence we know

    $$\begin{gathered} P; E \vdash a_1 \uparrow a_1' \\ P; E \vdash a_2 \uparrow a_2' \\ a' = (a_1' \sqcup a_2') \end{gathered}$$

    By IH, we have

    $$\begin{gathered} P; E \vdash a_1' \\ P; E \vdash a_2' \end{gathered}$$

    Since all locks in a_1' and a_2' are constant, and joining does not change or add lock expressions, by rules [AT BASE] and [AT COND] we may conclude

    $$P; E \vdash a'$$

LEMMA 9 (Substitution with well-formed lock). *If we have*

$$\begin{gathered} P; E, c\, x \vdash a \\ P; E \vdash_{\text{lock}} \ell \end{gathered}$$

then we can conclude $P; E \vdash a[x := \ell] \uparrow a[x := \ell]$.

Proof The judgment $P; E \vdash_{\text{lock}} \ell$ is derivable only through rule [LOCK EXP]; hence we know

$$P; E \vdash \ell : c \cdot \text{AF}$$

We proceed by induction on the structure of effect a.

- $a = b$. In this case, our assumption of $P; (E, c\ x) \vdash a$ is derivable only through rule [AT BASE]; hence we know

  $$P \vdash (E, c\, x)$$

  We may further weaken the environment by rule [ENV X]:

  $$P \vdash E$$

  Then by rule [LIFT BASE] we may conclude

  $$P; E \vdash a \uparrow a$$

  Since a does not mention x, we have the equality

  $$a = a[x := \ell]$$

  Substituting, we obtain our desired result

  $$P; E \vdash a[x := \ell] \uparrow a[x := \ell]$$

- $a = \ell' \,?\, a_1 : a_2$. In this case, our assumption of $P; (E, c\, x) \vdash a$ is derivable only through rule [AT COND]; hence we know

  $$\begin{gathered} P; (E, c\, x) \vdash_{\text{lock}} \ell' \\ P; (E, c\, x) \vdash a_1 \\ P; (E, c\, x) \vdash a_2 \end{gathered}$$

  By IH, we have

  $$\begin{gathered} P; E \vdash a_1[x := \ell] \uparrow a_1[x := \ell] \\ P; E \vdash a_2[x := \ell] \uparrow a_2[x := \ell] \end{gathered}$$

  By Lemma 6 and $P; (E, c\, x) \vdash_{\text{lock}} \ell'$, we may conclude

  $$P; E \vdash_{\text{lock}} \ell'[x := \ell]$$

  By rule [LIFT COND], we may conclude

  $$\begin{aligned} &P; E \vdash \\ &\quad \ell'[x := \ell] \,?\, a_1[x := \ell] : a_2[x := \ell] \uparrow \\ &\quad \ell'[x := \ell] \,?\, a_1[x := \ell] : a_2[x := \ell] \end{aligned}$$

  Simplifying, we obtain our desired result

  $$P; E \vdash a[x := \ell] \uparrow a[x := \ell]$$

LEMMA 10 (While equivalence). *If we have a judgment* $P; E \vdash \texttt{while}\, e_1\, e_2 : c{\cdot}a$ *then we can also conclude* $P; E \vdash \texttt{if}\ e_1\ (e_2; \texttt{while}\ e_1\ e_2)\ \texttt{null}$ a

Proof We start with an alternative, but equivalent, definition of iterative closure. For effect a,

$$a^* \overset{\text{def}}{=} (\text{AF} \sqcup (a) \sqcup (a; a) \sqcup (a; a; a) \sqcup \ldots)$$

It is easy to verify that this alternative definition is equivalent to the original definition.

The `while` judgment assumption is derivable only through rule [EXP WHILE]; hence we know

$$\begin{gathered} P; E \vdash e_1 : c_1 \cdot a_1 \\ P; E \vdash e_2 : c_2 \cdot a_2 \\ a = (a_1; (a_2; a_1)^*) \end{gathered}$$

Then by expanding the iterative closure, we get the equality

$$a = a_1; (\text{AF} \sqcup (a_2; a_1) \sqcup (a_2; a_1; a_2; a_1) \sqcup \ldots)$$

Using rule [EXP IF], we may conclude

$$\begin{gathered} P; E \vdash \texttt{if}\ e_1\ (e_2; \texttt{while}\ e_1\ e_2)\ \texttt{null} : c \cdot a' \\ a' = a_1; ((a_2; a) \sqcup \text{AF}) \end{gathered}$$

Expanding a and distributing $a_2; a_1$ across the joins, we get the equality

$$\begin{aligned} a' &= a_1; (a_2; a_1; (\text{AF} \sqcup (a_2; a_1) \sqcup (a_2; a_1; a_2; a_1) \sqcup \ldots) \sqcup \text{AF}) \\ &= a_1; (\text{AF} \sqcup (a_2; a_1) \sqcup (a_2; a_1; a_2; a_1) \sqcup \ldots) \\ &= a \end{aligned}$$

Thus we obtain our desired result.

LEMMA 11 (Lifting After Substitution). *Suppose the following:*

$$\begin{gathered} P; E \vdash a[x := e'] \uparrow a' \\ P; E \vdash a[x := e''] \uparrow a'' \\ e' \textit{ not a value} \end{gathered}$$

Then we can conclude $a'' \sqsubseteq a'$.

Proof By induction on structure of effect a.

- $a = b$. In this case, the following holds:

  $$a'' = (b[x := e'']) = b = (b[x := e']) = a'$$

  Hence we may conclude

  $$a'' \sqsubseteq a'$$

- $a = \ell\,?\,a_1 : a_2$. The assumptions $P; E \vdash a[x := e'] \uparrow a'$ and $P; E \vdash a[x := e''] \uparrow a''$ are derivable only through rules [LIFT GOOD LOCK] and [LIFT BAD LOCK]. Hence, we know the following must hold:

  $$\begin{array}{l} P; E \vdash a_1[x := e'] \uparrow a_1' \\ P; E \vdash a_1[x := e''] \uparrow a_1'' \end{array}$$

  and

  $$\begin{array}{l} P; E \vdash a_2[x := e'] \uparrow a_2' \\ P; E \vdash a_2[x := e''] \uparrow a_2'' \end{array}$$

  By IH, we know

  $$\begin{array}{l} a_1'' \sqsubseteq a_1' \\ a_2'' \sqsubseteq a_2' \end{array}$$

  We proceed by case analysis on whether x is free in ℓ.

  - x is free in ℓ. In this case, $\ell[x := e']$ is not a value, since e' is not a value. The lock judgment does not hold:

    $$P; E \not\vdash_{\text{lock}} \ell[x := e']$$

    Hence, by rule [LIFT BAD LOCK] we get

    $$a' = a_1' \sqcup a_2'$$

    Case analysis on lock judgment for $\ell[x := e'']$:

    – $P; E \vdash_{\text{lock}} \ell[x := e'']$. In this case,

    $$a'' = (\ell[x := e'']\,?\,a_1'' : a_2'') \sqsubseteq (a_1' \sqcup a_2') = a'$$

    – $P; E \not\vdash_{\text{lock}} \ell[x := e'']$. In this case,

    $$a'' = (a_1'' \sqcup a_2'') \sqsubseteq (a_1' \sqcup a_2') = a'$$

  - x is not free in ℓ. In this case, the following holds:

    $$\ell[x := e'] = \ell = \ell[x := e'']$$

    Case analysis on lock judgment for ℓ.

    – $P; E \vdash_{\text{lock}} \ell$. In this case, we may conclude

    $$a'' = (\ell\,?\,a_1'' : a_2'') \sqsubseteq (\ell\,?\,a_1' : a_2') = a'$$

    – $P; E \not\vdash_{\text{lock}} \ell$. In this case, we may conclude

    $$a'' = (a_1'' \sqcup a_2'') \sqsubseteq (a_1' \sqcup a_2') = a'$$

LEMMA 12 (Environment Strengthening). *Suppose $E = E', c\,x, E''$. If $P \vdash E$, then the following hold:*

1. *If $P \vdash (E', E'')$ then $P \vdash E$.*
2. *If $P; (E', E'') \vdash meth$ then $P; E \vdash meth$.*
3. *If $P; (E', E'') \vdash a$ then $P; E \vdash a$.*
4. *If $P; (E', E'') \vdash_{\text{lock}} \ell$ then $P; E \vdash_{\text{lock}} \ell$.*
5. *If $P; (E', E'') \vdash a \uparrow a'$ then $P; E \vdash a \uparrow a'$.*
6. *If $P; (E', E'') \vdash e : c' \cdot a$ then $P; E \vdash e : c' \cdot a$.*

Proof By simultaneous induction on all parts of the lemma.

LEMMA 13 (Object Map). *Let e an expression, and both σ and σ' well-formed states. If we have $\alpha(\sigma, e)$ and $\alpha(\sigma', e)$ both well-defined, and the object maps in both σ and σ' have equivalent domains, then*

$$\alpha(\sigma, e) = \alpha(\sigma', e)$$

Proof Since $\alpha(\sigma, e)$ and $\alpha(\sigma', e)$ are both well-defined, we have the judgments

$$\begin{array}{l} P; E_\sigma \vdash e : c \cdot a \\ P; E_{\sigma'} \vdash e : c \cdot a' \end{array}$$

Because both σ and σ' have equivalent domains, every address ρ has the same type d in both σ and σ'. The second judgment we have, then, is equivalent to

$$P; E_\sigma \vdash e : c \cdot a'$$

We conclude with our desired result:

$$a = a'$$

LEMMA 14 (Sequentiality for Sync). *Assume for some state σ the following*

$$\begin{array}{l} P; E_\sigma \vdash_{\text{lock}} \rho \\ P; E_\sigma \vdash a \\ \textit{disjoint locksets } h \textit{ and } n \\ \rho \in n \end{array}$$

Then we may conclude

$$[\![\gamma]\!]; \mathcal{SI}(\rho, a) \sqsubseteq_n^h \mathcal{S}(\rho, \gamma, a)$$

Proof By induction over structure of effect a.

- $a = \kappa$. In this case,

  $$\begin{array}{r} \mathcal{SI}(\rho, a) = \kappa; \text{AL} \\ \mathcal{S}(\rho, \gamma, a) = \rho\,?\,a : ([\![\gamma]\!]; \text{AR}; \kappa; \text{AL}) \end{array}$$

  With $\rho \notin h$, we can show for any κ:

  $$[\![\gamma]\!]; \kappa; \text{AL} \sqsubseteq_n^h [\![\gamma]\!]; \text{AR}; \kappa; \text{AL}$$

  Hence we obtain our desired result

  $$[\![\gamma]\!]; \mathcal{SI}(\rho, a) \sqsubseteq_n^h \mathcal{S}(\rho, \gamma, a)$$

- $a = \rho\,?\,a_1 : a_2$. In this case,

  $$\begin{array}{r} \mathcal{SI}(\rho, a) = \mathcal{SI}(\rho, a_1) \\ \mathcal{S}(\rho, \gamma, a) = \mathcal{S}(\rho, \gamma, a_1) \end{array}$$

  By IH,

  $$[\![\gamma]\!]; \mathcal{SI}(\rho, a_1) \sqsubseteq_n^h \mathcal{S}(\rho, \gamma, a_1)$$

  Hence we obtain our desired result

  $$[\![\gamma]\!]; \mathcal{SI}(\rho, a) \sqsubseteq_n^h \mathcal{S}(\rho, \gamma, a)$$

- $a = \rho'\,?\,a_1 : a_2$ where $\rho' \neq \rho$. In this case,

  $$\begin{array}{r} \mathcal{SI}(\rho, a) = \rho'\,?\,\mathcal{SI}(\rho, a_1) : \mathcal{SI}(\rho, a_2) \\ \mathcal{S}(\rho, \gamma, a) = \rho'\,?\,\mathcal{S}(\rho, \gamma, a_1) : \mathcal{S}(\rho, \gamma, a_2) \end{array}$$

  Assume that $\rho' \notin n$. Then by IH,

  $$[\![\gamma]\!]; \mathcal{SI}(\rho, a_1) \sqsubseteq_n^{h \cup \{\rho'\}} \mathcal{S}(\rho, \gamma, a_1)$$

  Assume that $\rho' \notin h$. Then by IH,

  $$[\![\gamma]\!]; \mathcal{SI}(\rho, a_2) \sqsubseteq_{n \cup \{\rho'\}}^h \mathcal{S}(\rho, \gamma, a_2)$$

  Combining these, we may conclude

  $$\begin{array}{r} \rho'\,?\,([\![\gamma]\!]; \mathcal{SI}(\rho, a_1)) : ([\![\gamma]\!]; \mathcal{SI}(\rho, a_2)) \\ \sqsubseteq_n^h \rho'\,?\,\mathcal{S}(\rho, \gamma, a_1) : \mathcal{S}(\rho, \gamma, a_2) \end{array}$$

  By definition of conditional effects, we extract $[\![\gamma]\!]$:

  $$\begin{array}{r} [\![\gamma]\!]; \rho'\,?\,\mathcal{SI}(\rho, a_1) : \mathcal{SI}(\rho, a_2) \\ \sqsubseteq_n^h \rho'\,?\,\mathcal{S}(\rho, \gamma, a_1) : \mathcal{S}(\rho, \gamma, a_2) \end{array}$$

  Substituting, we obtain our desired result

  $$[\![\gamma]\!]; \mathcal{SI}(\rho, a) \sqsubseteq_n^h \mathcal{S}(\rho, \gamma, a)$$

Lemma 15 (Subeffect Relation for Sync). *Assume the following:*

$$\begin{array}{l} P; E \vdash_{\text{lock}} \rho \\ \exists\, h, n \,.\, h \not\pitchfork n \not\pitchfork \{\rho\} \\ P; E \vdash a \end{array}$$

Then we may conclude

$$a \sqsubseteq_n^{h \cup \{\rho\}} \mathcal{S}(\rho, \gamma, a)$$

Proof By induction over structure of effect a.

- $a = \kappa$. In this case,

  $$\mathcal{S}(\rho, \gamma, a) = \rho \,?\, \kappa : (\llbracket \gamma \rrbracket; \textsc{ar}; \kappa; \textsc{al})$$

  We can conclude the following

  $$\kappa \sqsubseteq_n^{h \cup \{\rho\}} \rho \,?\, \kappa : (\llbracket \gamma \rrbracket; \textsc{ar}; \kappa; \textsc{al})$$

  only if the following are true:

  $$\begin{array}{c} \rho \notin n \Rightarrow \kappa \sqsubseteq_n^{h \cup \{\rho\}} \kappa \\ \rho \notin h \cup \{\rho\} \Rightarrow \kappa \sqsubseteq_{n \cup \{\rho\}}^{h \cup \{\rho\}} (\llbracket \gamma \rrbracket; \textsc{ar}; \kappa; \textsc{al}) \end{array}$$

  The first statement is true since it relates two combined (basic) effects. The second statement is true vacuously. Hence we obtain our desired result

  $$a \sqsubseteq_n^{h \cup \{\rho\}} \mathcal{S}(\rho, \gamma, a)$$

- $a = \rho \,?\, a_1 : a_2$. In this case,

  $$\mathcal{S}(\rho, \gamma, a) = \mathcal{S}(\rho, \gamma, \rho \,?\, a_1 : a_2) = \mathcal{S}(\rho, \gamma, a_1)$$

  By IH, we have

  $$a_1 \sqsubseteq_n^{h \cup \{\rho\}} \mathcal{S}(\rho, \gamma, a_1)$$

  We can conclude the following

  $$\rho \,?\, a_1 : a_2 \sqsubseteq_n^{h \cup \{\rho\}} \mathcal{S}(\rho, \gamma, \rho \,?\, a_1 : a_2)$$

  only if the following are true:

  $$\begin{array}{c} \rho \notin n \Rightarrow a_1 \sqsubseteq_n^{h \cup \{\rho\}} \mathcal{S}(\rho, \gamma, \rho \,?\, a_1 : a_2) \\ \rho \notin h \cup \{\rho\} \Rightarrow a_2 \sqsubseteq_{n \cup \{\rho\}}^{h \cup \{\rho\}} \mathcal{S}(\rho, \gamma, \rho \,?\, a_1 : a_2) \end{array}$$

  The first statement is true due to IH. The second statement is true vacuously. Hence we obtain our desired result

  $$a \sqsubseteq_n^{h \cup \{\rho\}} \mathcal{S}(\rho, \gamma, a)$$

- $a = \rho' \,?\, a_1 : a_2$, where $\rho' \neq \rho$. In this case,

  $$\mathcal{S}(\rho, \gamma, a) = \mathcal{S}(\rho, \gamma, \rho' \,?\, a_1 : a_2) = \rho' \,?\, \mathcal{S}(\rho, \gamma, a_1) : \mathcal{S}(\rho, \gamma, a_2)$$

  We can conclude the following

  $$\rho' \,?\, a_1 : a_2 \sqsubseteq_n^{h \cup \{\rho\}} \mathcal{S}(\rho, \gamma, \rho' \,?\, a_1 : a_2)$$

  only if the following are true:

  $$\begin{array}{c} \rho' \notin n \Rightarrow a_1 \sqsubseteq_n^{h \cup \{\rho, \rho'\}} \mathcal{S}(\rho, \gamma, a_1) \\ \rho' \notin h \cup \{\rho\} \Rightarrow a_2 \sqsubseteq_{n \cup \{\rho'\}}^{h \cup \{\rho\}} \mathcal{S}(\rho, \gamma, a_2) \end{array}$$

  Assume $\rho' \notin n$. Then by IH,

  $$a_1 \sqsubseteq_n^{h \cup \{\rho', \rho\}} \mathcal{S}(\rho, \gamma, a_1)$$

  Assume $\rho' \notin h \cup \{\rho\}$. Then by IH,

  $$a_2 \sqsubseteq_{n \cup \{\rho'\}}^{h \cup \{\rho\}} \mathcal{S}(\rho, \gamma, a_2)$$

  Hence we obtain our desired result

  $$a \sqsubseteq_n^{h \cup \{\rho\}} \mathcal{S}(\rho, \gamma, a)$$

Lemma 16 (Sync Not a Left Mover). *Let σ a state, $\mathcal{E}$ an evaluation context, ρ a lock, and a an effect. Assume the following:*

$$\begin{array}{l} P; E_\sigma \vdash a \\ P; E_\sigma \vdash_{\text{lock}} \rho \\ \rho \notin \mathit{locks}(\mathcal{E}) \end{array}$$

Then we may conclude

$$Y(\mathcal{E}, \mathcal{S}(\rho, ., a)) \not\sqsubseteq \textsc{cl}$$

Proof By induction on structure of effect a.

- $a = \kappa$. In this case,

  $$\mathcal{S}(\rho, ., a) = \rho \,?\, \kappa : (\textsc{ar}; \kappa; \textsc{al})$$

  Substituting and simplifying, we have

  $$\begin{array}{l} Y(\mathcal{E}, \mathcal{S}(\rho, ., a)) \\ = Y(\mathcal{E}, \rho \,?\, \kappa : (\textsc{ar}; \kappa; \textsc{al})) \\ = \rho \,?\, \kappa : (\textsc{ar}; \kappa; \textsc{al}) \end{array}$$

  We have $Y(\mathcal{E}, \mathcal{S}(\rho, ., a)) \sqsubseteq \textsc{cl}$ only if this holds under any lockset h. However, if $\rho \notin h$, then we get

  $$\textsc{ar}; \kappa; \textsc{al} = \textsc{an} \not\sqsubseteq \textsc{cl}$$

  Hence we obtain our desired result

  $$Y(\mathcal{E}, \mathcal{S}(\rho, ., a)) \not\sqsubseteq \textsc{cl}$$

- $a = \rho \,?\, a_1 : a_2$. In this case,

  $$\mathcal{S}(\rho, ., a) = \mathcal{S}(\rho, ., \rho \,?\, a_1 : a_2) = \mathcal{S}(\rho, ., a_1)$$

  By IH, we have

  $$Y(\mathcal{E}, \mathcal{S}(\rho, ., a_1)) \not\sqsubseteq \textsc{cl}$$

  Hence we obtain our desired result

  $$Y(\mathcal{E}, \mathcal{S}(\rho, ., a)) \not\sqsubseteq \textsc{cl}$$

- $a = \rho' \,?\, a_1 : a_2$, where $\rho' \neq \rho$. In this case,

  $$\mathcal{S}(\rho, ., a) = \mathcal{S}(\rho, ., \rho' \,?\, a_1 : a_2) = \rho' \,?\, \mathcal{S}(\rho, ., a_1) : \mathcal{S}(\rho, ., a_2)$$

  By IH, we have

  $$\begin{array}{l} Y(\mathcal{E}, \mathcal{S}(\rho, ., a_1)) \not\sqsubseteq \textsc{cl} \\ Y(\mathcal{E}, \mathcal{S}(\rho, ., a_2)) \not\sqsubseteq \textsc{cl} \end{array}$$

  Then we may conclude with

  $$\rho' \,?\, Y(\mathcal{E}, \mathcal{S}(\rho, ., a_1)) : Y(\mathcal{E}, \mathcal{S}(\rho, ., a_2)) \not\sqsubseteq \textsc{cl}$$

  By Lemma 17, we can simplify this to be:

  $$\begin{array}{l} Y(\mathcal{E}, \mathcal{S}(\rho, ., \rho' \,?\, a_1 : a_2)) \\ = Y(\mathcal{E}, \rho' \,?\, \mathcal{S}(\rho, ., a_1) : \mathcal{S}(\rho, ., a_2)) \\ = Y(\mathcal{E}, \rho' \,?\, Y(\mathcal{E}, \mathcal{S}(\rho, ., a_1)) : Y(\mathcal{E}, \mathcal{S}(\rho, ., a_2))) \\ \sqsubseteq \rho' \,?\, Y(\mathcal{E}, \mathcal{S}(\rho, ., a_1)) : Y(\mathcal{E}, \mathcal{S}(\rho, ., a_2)) \\ \not\sqsubseteq \textsc{cl} \end{array}$$

  Hence we obtain our desired result

  $$Y(\mathcal{E}, \mathcal{S}(\rho, ., a)) \not\sqsubseteq \textsc{cl}$$

Lemma 17 (Y Subeffect). *Let a an effect and $\mathcal{E}$ an evaluation context. If $P; E \vdash a$, then $Y(\mathcal{E}, a) \sqsubseteq a$.*

Proof By induction over the structure of effect a.

- $a = \kappa$. In this case, since we cannot simplify κ any further, we have

  $$Y(\mathcal{E}, \kappa) = \kappa$$

Hence we obtain our desired result

$$Y(\mathcal{E}, a) \sqsubseteq a$$

- $a = \rho \,?\, a_1 : a_2$. We proceed by case analysis on ρ:
  - $\rho \in \mathit{locks}(\mathcal{E})$. In this case,

    $$Y(\mathcal{E}, \rho \,?\, a_1 : a_2) = Y(\mathcal{E}, a_1)$$

    By IH, we have

    $$Y(\mathcal{E}, a_1) \sqsubseteq a_1$$

    Also, may conclude the following

    $$a_1 \sqsubseteq \rho \,?\, a_1 : a_2$$

    only if the following hold:

    $$\begin{array}{l} \rho \notin n \Rightarrow a_1 \sqsubseteq^{\{\rho\}} a_1 \\ \rho \notin h \Rightarrow a_1 \sqsubseteq_{\{\rho\}} a_2 \end{array}$$

    The first statement is trivially true. The second statement is vacuously true, due to a contradiction: ρ cannot be in the not-held lockset, since we know ρ is held in $\mathcal{E}$.
    Hence we obtain our desired result

    $$Y(\mathcal{E}, a) \sqsubseteq a$$

  - $\rho \notin \mathit{locks}(\mathcal{E})$. In this case,

    $$Y(\mathcal{E}, \rho \,?\, a_1 : a_2) = \rho \,?\, Y(\mathcal{E}, a_1) : Y(\mathcal{E}, a_2)$$

    By IH, we have

    $$\begin{array}{l} Y(\mathcal{E}, a_1) \sqsubseteq a_1 \\ Y(\mathcal{E}, a_2) \sqsubseteq a_2 \end{array}$$

    We may conclude the following:

    $$\rho \,?\, Y(\mathcal{E}, a_1) : Y(\mathcal{E}, a_2) \sqsubseteq \rho \,?\, a_1 : a_2$$

    only if the following hold:

    $$\begin{array}{l} \rho \notin n \Rightarrow Y(\mathcal{E}, a_1) \sqsubseteq^{\{\rho\}} a_1 \\ \rho \notin h \Rightarrow Y(\mathcal{E}, a_2) \sqsubseteq_{\{\rho\}} a_2 \end{array}$$

    Both these statements are true due to IH. Since $\rho \notin \mathit{locks}(\mathcal{E})$, we have the equality

    $$Y(\mathcal{E}, \rho \,?\, a_1 : a_2) = \rho \,?\, Y(\mathcal{E}, a_1) : Y(\mathcal{E}, a_2)$$

    Hence we obtain our desired result

    $$Y(\mathcal{E}, a) \sqsubseteq a$$

LEMMA 18 (Lock Removal). *Let h and n be lock sets, and ρ a lock. Let a_1 and a_2 be effects. Assume the following:*

$$\begin{array}{l} P; E \vdash_{\text{lock}} \rho \\ P; E \vdash a_1 \\ P; E \vdash a_2 \\ \exists\, h, n \,.\, h \not\cap n \not\cap \{\rho\} \\ a_1 \textit{ and } a_2 \textit{ are not conditional on } \rho \\ a_1 \sqsubseteq_n^{h \cup \{\rho\}} a_2 \end{array}$$

Then we may conclude

$$a_1 \sqsubseteq_n^h a_2$$

Proof By induction on the structure of effects a_1 and a_2.

- $(a_1, a_2) = (\kappa_1, \kappa_2)$. In this case, we assume

  $$\kappa_1 \sqsubseteq_n^{h \cup \{\rho\}} \kappa_2$$

  By inversion on this derivation, we know

  $$\kappa_1 \sqsubseteq \kappa_2$$

  Then by definition of effect ordering on conditionals, we may conclude

  $$\kappa_1 \sqsubseteq_n^h \kappa_2$$

- $(a_1, a_2) = (\kappa, \rho' \,?\, a_{2L} : a_{2R})$, where $\rho' \neq \rho$. In this case, we assume

  $$\kappa \sqsubseteq_n^{h \cup \{\rho\}} \rho' \,?\, a_{2L} : a_{2R}$$

  By inversion on this derivation, we know

  $$\begin{array}{l} (\rho' \notin n) \Rightarrow \kappa \sqsubseteq_n^{h \cup \{\rho, \rho'\}} a_{2L} \\ (\rho' \notin h \cup \{\rho\}) \Rightarrow \kappa \sqsubseteq_{n \cup \{\rho'\}}^{h \cup \{\rho\}} a_{2R} \end{array}$$

  By IH, we have

  $$\begin{array}{l} (\rho' \notin n) \Rightarrow \kappa \sqsubseteq_n^{h \cup \{\rho'\}} a_{2L} \\ (\rho' \notin h) \Rightarrow \kappa \sqsubseteq_{n \cup \{\rho'\}}^h a_{2R} \end{array}$$

  By definition of effect ordering on conditionals, we may conclude

  $$\kappa \sqsubseteq_n^h \rho' \,?\, a_{2L} : a_{2R}$$

- $a_1 = \rho' \,?\, a_{1L} : a_{1R}$, where $\rho' \neq \rho$. In this case, we assume

  $$\rho' \,?\, a_{1L} : a_{1R} \sqsubseteq_n^{h \cup \{\rho\}} a_2$$

  By inversion on this derivation, we know

  $$\begin{array}{l} (\rho' \notin n) \Rightarrow a_{1L} \sqsubseteq_n^{h \cup \{\rho, \rho'\}} a_2 \\ (\rho' \notin h \cup \{\rho\}) \Rightarrow a_{1R} \sqsubseteq_{n \cup \{\rho'\}}^{h \cup \{\rho\}} a_2 \end{array}$$

  By IH, we have

  $$\begin{array}{l} (\rho' \notin n) \Rightarrow a_{1L} \sqsubseteq_n^{h \cup \{\rho'\}} a_2 \\ (\rho' \notin h) \Rightarrow a_{1R} \sqsubseteq_{n \cup \{\rho'\}}^h a_2 \end{array}$$

  By definition of effect ordering on conditionals, we may conclude

  $$\rho' \,?\, a_{1L} : a_{1R} \sqsubseteq_n^h a_2$$

LEMMA 19 (Yet Another In-Sync Subeffect Relation). *Let a be an effect, and ρ a lock. Assume the following:*

$$\begin{array}{l} P; E \vdash_{\text{lock}} \rho \\ P; E \vdash a \\ \exists\, h, n \,.\, h \not\cap n \ \wedge\ \rho \in h \end{array}$$

Then we may conclude

$$a; \text{AL} \sqsubseteq_n^h \mathcal{SI}(\rho, a)$$

Proof By induction over structure of effect a.

- $a = \kappa$. In this case, by definition of $\sqsubseteq_n^h$,

  $$\kappa \sqsubseteq_n^h \kappa$$

  By Lemma 1,

  $$\kappa; \text{AL} \sqsubseteq_n^h \kappa; \text{AL}$$

  By definition of $\mathcal{SI}$,

  $$\kappa; \text{AL} \sqsubseteq_n^h \mathcal{SI}(\rho, \kappa)$$

  We obtain our desired result

  $$a; \text{AL} \sqsubseteq_n^h \mathcal{SI}(\rho, a)$$

- $a = \rho \mathbin{?} a_1 : a_2$. In this case, our conclusion, which is

$$\begin{array}{rl} & \rho \mathbin{?} a_1 : a_2; \textsc{al} \sqsubseteq_n^h \mathcal{SI}(\rho, \rho \mathbin{?} a_1 : a_2) \\ \Leftrightarrow & \rho \mathbin{?} (a_1; \textsc{al}) : (a_2; \textsc{al}) \sqsubseteq_n^h \mathcal{SI}(\rho, a_1) \end{array}$$

can only be true if the antecedents are true, by inversion on the derivation:

$$\begin{array}{l} (\rho \notin n) \Rightarrow a_1; \textsc{al} \sqsubseteq_n^h \mathcal{SI}(\rho, a_1) \\ (\rho \notin h) \Rightarrow a_2; \textsc{al} \sqsubseteq_{n \cup \{\rho\}}^h \mathcal{SI}(\rho, a_1) \end{array}$$

The first statement is true by IH. The second statement is vacuously true, since $\rho \in h$. Hence we may conclude our desired result

$$a; \textsc{al} \sqsubseteq_n^h \mathcal{SI}(\rho, a)$$

- $a = \rho' \mathbin{?} a_1 : a_2$ where $\rho' \neq \rho$. In this case, our conclusion, which is

$$\begin{array}{rl} & \rho' \mathbin{?} a_1 : a_2; \textsc{al} \sqsubseteq_n^h \mathcal{SI}(\rho, \rho' \mathbin{?} a_1 : a_2) \\ \Leftrightarrow & \rho' \mathbin{?} (a_1; \textsc{al}) : (a_2; \textsc{al}) \sqsubseteq_n^h \rho' \mathbin{?} \mathcal{SI}(\rho, a_1) : \mathcal{SI}(\rho, a_2) \end{array}$$

can only be true if the antecedents are true, by inversion on the derivation:

$$\begin{array}{l} (\rho' \notin n) \Rightarrow a_1; \textsc{al} \sqsubseteq_n^{h \cup \{\rho'\}} \mathcal{SI}(\rho, a_1) \\ (\rho' \notin h) \Rightarrow a_2; \textsc{al} \sqsubseteq_{n \cup \{\rho'\}}^h \mathcal{SI}(\rho, a_2) \end{array}$$

Both statements are true by IH. Hence we may conclude our desired result

$$a; \textsc{al} \sqsubseteq_n^h \mathcal{SI}(\rho, a)$$

LEMMA 20 (Subeffect Relation for In-Sync). *Let a_y, a and a' be effects, and ρ be a lock. Assume the following:*

$$\begin{array}{l}
P; E \vdash_{\text{lock}} \rho \\
P; E \vdash a_y \quad \textit{and} \quad P; E \vdash a \quad \textit{and} \quad P; E \vdash a' \\
\exists\, h, n \,.\, h \not\pitchfork n \ \ \wedge \ \ \rho \in h \\
a_y; a' \sqsubseteq^h_n a
\end{array}$$

Then we may conclude

$$a_y; \mathcal{SI}(\rho, a') \sqsubseteq^h_n \mathcal{SI}(\rho, a)$$

Proof By induction over structure of effect a'.

- $a' = \kappa$. In this case, we have

$$\begin{array}{rll}
 & a_y; \kappa \sqsubseteq^h_n a & \text{by assumption} \\
\Leftrightarrow & a_y; \kappa; \text{AL} \sqsubseteq^h_n a; \text{AL} & \text{by Lemma 1} \\
\Rightarrow & a_y; \kappa; \text{AL} \sqsubseteq^h_n \mathcal{SI}(\rho, a) & \text{by Lemma 19} \\
\Leftrightarrow & a_y; \mathcal{SI}(\rho, \kappa) \sqsubseteq^h_n \mathcal{SI}(\rho, a) & \text{by definition of } \mathcal{SI} \\
\Leftrightarrow & a_y; \mathcal{SI}(\rho, a') \sqsubseteq^h_n \mathcal{SI}(\rho, a) & \text{our conclusion}
\end{array}$$

- $a' = \rho \,?\, a_1 : a_2$.

$$\begin{array}{rll}
 & a_y; (\rho \,?\, a_1 : a_2) \sqsubseteq^h_n a & \text{by assumption} \\
\Leftrightarrow & \rho \,?\, (a_y; a_1) : (a_y; a_1) \sqsubseteq^h_n a & \text{by sequential composition} \\
\Rightarrow & (\rho \notin n) \Rightarrow a_y; a_1 \sqsubseteq^h_n a & \text{by inversion on derivation} \\
\Rightarrow & a_y; \mathcal{SI}(\rho, a_1) \sqsubseteq^h_n \mathcal{SI}(\rho, a) & \text{by IH} \\
\Leftrightarrow & a_y; \mathcal{SI}(\rho, \rho \,?\, a_1 : a_2) \sqsubseteq^h_n \mathcal{SI}(\rho, a) & \text{by definition of } \mathcal{SI} \\
\Leftrightarrow & a_y; \mathcal{SI}(\rho, a') \sqsubseteq^h_n \mathcal{SI}(\rho, a) & \text{our conclusion}
\end{array}$$

- $a' = \rho' \,?\, a_1 : a_2$ where $\rho' \neq \rho$.

$$\begin{array}{rll}
 & a_y; (\rho' \,?\, a_1 : a_2) \sqsubseteq^h_n a & \text{by assumption} \\
\Leftrightarrow & \rho' \,?\, (a_y; a_1) : (a_y; a_1) \sqsubseteq^h_n a & \text{by sequential composition} \\
\Rightarrow & (\rho' \notin n) \Rightarrow a_y; a_1 \sqsubseteq^{h \cup \{\rho'\}}_n a \quad \wedge & \\
 & (\rho' \notin h) \Rightarrow a_y; a_2 \sqsubseteq^h_{n \cup \{\rho'\}} a & \text{by inversion on derivation} \\
\Rightarrow & (\rho' \notin n) \Rightarrow a_y; \mathcal{SI}(\rho, a_1) \sqsubseteq^{h \cup \{\rho'\}}_n \mathcal{SI}(\rho, a) \quad \wedge & \\
 & (\rho' \notin h) \Rightarrow a_y; \mathcal{SI}(\rho, a_2) \sqsubseteq^h_{n \cup \{\rho'\}} \mathcal{SI}(\rho, a) & \text{by IH} \\
\Rightarrow & \rho' \,?\, (a_y; \mathcal{SI}(\rho, a_1)) : (a_y; \mathcal{SI}(\rho, a_2)) \sqsubseteq^h_n \mathcal{SI}(\rho, a) & \text{by definition of } \sqsubseteq^h_n \\
\Leftrightarrow & a_y; \rho' \,?\, \mathcal{SI}(\rho, a_1) : \mathcal{SI}(\rho, a_2) \sqsubseteq^h_n \mathcal{SI}(\rho, a) & \text{by sequential composition} \\
\Leftrightarrow & a_y; \mathcal{SI}(\rho, \rho' \,?\, a_1 : a_2) \sqsubseteq^h_n \mathcal{SI}(\rho, a) & \text{by definition of } \mathcal{SI} \\
\Leftrightarrow & a_y; \mathcal{SI}(\rho, a') \sqsubseteq^h_n \mathcal{SI}(\rho, a) & \text{our conclusion}
\end{array}$$

LEMMA 21 (CS Context Subexpression). *Suppose there is a deduction that concludes $\mathit{ls} \vdash_{cs} \mathcal{E}[e]$. Then that deduction contains, at a position corresponding to the hole in $\mathcal{E}$, a subdeduction that concludes $\mathit{ls}' \vdash_{cs} e$, where $\mathit{ls}' \subseteq \mathit{ls}$.*

Proof Induction over the derivation of $\mathit{ls} \vdash_{cs} \mathcal{E}[e]$.

LEMMA 22 (CS Context Replacement). *Suppose the following:*

$$\begin{array}{l} \mathit{ls} \vdash_{cs} \mathcal{E}[e], \textit{ and} \\ \mathit{ls}_e \vdash_{cs} e, \textit{ and} \\ \mathit{ls}'_e \vdash_{cs} e', \textit{ and} \\ \mathit{ls}'_e \not\pitchfork (\mathit{ls} \setminus \mathit{ls}_e) \end{array}$$

Then $\mathit{ls}' \vdash_{cs} \mathcal{E}[e']$ and $\mathit{ls}' = (\mathit{ls} \setminus \mathit{ls}_e) \cup \mathit{ls}'_e$.

Proof By induction over the structure of $\mathcal{E}$.

- $\mathcal{E} \equiv [\,]$: In this case, we assume

  $$\begin{array}{l} \mathit{ls} \vdash_{cs} [e] \\ \mathit{ls}_e \vdash_{cs} e \\ \mathit{ls}'_e \vdash_{cs} e' \\ \mathit{ls}'_e \not\pitchfork (\mathit{ls} \setminus \mathit{ls}_e) \end{array}$$

  and may immediately conclude our desired result

  $$\begin{array}{l} \mathit{ls}' \vdash_{cs} [e'] \\ \mathit{ls}' = \mathit{ls}'_e = (\mathit{ls} \setminus \mathit{ls}_e) \cup \mathit{ls}'_e \end{array}$$

- $\mathcal{E} \equiv \texttt{new}\, c\texttt{(}\overline{v},\ \mathcal{E}',\ \overline{e}\texttt{)}$: In this case, we assume

  $$\begin{array}{l} \mathit{ls} \vdash_{cs} \texttt{new}\, c\ \texttt{(}\overline{v},\ \mathcal{E}'[e],\ \overline{e}\texttt{)} \\ \mathit{ls}_e \vdash_{cs} e \\ \mathit{ls}'_e \vdash_{cs} e' \\ \mathit{ls}'_e \not\pitchfork (\mathit{ls} \setminus \mathit{ls}_e) \end{array}$$

  In the first assumption, this judgment may be concluded only by rule [CS NOT IN-SYNC]; hence we know

  $$\mathit{ls} \vdash_{cs} \mathcal{E}'[e]$$

  By IH, we may conclude

  $$\begin{array}{l} \mathit{ls}' \vdash_{cs} \mathcal{E}'[e'] \\ \mathit{ls}' = (\mathit{ls} \setminus \mathit{ls}_e) \cup \mathit{ls}'_e \end{array}$$

  Since $\texttt{in-sync} \notin \texttt{new}\, c\ \texttt{(}\overline{v},\ [\,],\ \overline{e}\texttt{)}$, by rule [CS NOT IN-SYNC] we obtain our desired result

  $$\begin{array}{l} \mathit{ls}' \vdash_{cs} \texttt{new}\, c\ \texttt{(}\overline{v},\ \mathcal{E}'[e'],\ \overline{e}\texttt{)} \\ \mathit{ls}' = (\mathit{ls} \setminus \mathit{ls}_e) \cup \mathit{ls}'_e \end{array}$$

- $\mathcal{E} \equiv \mathcal{E}'{}_{\gamma} f$: Similar to previous argument.
- $\mathcal{E} \equiv \mathcal{E}'{}_{\gamma} f\ \texttt{=}\ e'$: Similar to previous argument.
- $\mathcal{E} \equiv v'{}_{\gamma} f\ \texttt{=}\ \mathcal{E}'$: Similar to previous argument.
- $\mathcal{E} \equiv \mathcal{E}'{}_{\gamma} m\texttt{(}\overline{e}\texttt{)}$: Similar to previous argument.
- $\mathcal{E} \equiv \mathcal{E}'{}_{\gamma} m\texttt{\#(}\overline{e}\texttt{)}$: Similar to previous argument.
- $\mathcal{E} \equiv \rho_{\gamma} m\texttt{(}\overline{v},\ \mathcal{E}', \overline{e}\texttt{)}$: Similar to previous argument.
- $\mathcal{E} \equiv \rho_{\gamma} m\texttt{\#(}\overline{v},\ \mathcal{E}', \overline{e}\texttt{)}$: Similar to previous argument.
- $\mathcal{E} \equiv \texttt{let}\ x\ \texttt{=}\ \mathcal{E}'\ \texttt{in}\ e'$: Similar to previous argument.
- $\mathcal{E} \equiv \texttt{if}\ \mathcal{E}'\ e_1\ e_2$: Similar to previous argument.
- $\mathcal{E} \equiv \mathcal{E}'{}_{\gamma}\texttt{sync}\ e'$: Similar to previous argument.
- $\mathcal{E} \equiv \texttt{in-sync}\ \rho\, \mathcal{E}'$: In this case, we assume

  $$\begin{array}{l} \mathit{ls} \vdash_{cs} \texttt{in-sync}\ \rho\, \mathcal{E}'[e] \\ \mathit{ls}_e \vdash_{cs} e \\ \mathit{ls}'_e \vdash_{cs} e' \\ \mathit{ls}'_e \not\pitchfork (\mathit{ls} \setminus \mathit{ls}_e) \end{array}$$

  In the first assumption, this judgment may be concluded only by rule [CS IN-SYNC]; hence we know

  $$\mathit{ls} \setminus \{\rho\} \vdash_{cs} \mathcal{E}'[e]$$

  From this judgment and Lemma 21, we know that $\rho \notin \mathit{ls}_e$. By IH, we may conclude

  $$\begin{array}{l} \mathit{ls}' \vdash_{cs} \mathcal{E}'[e'] \\ \mathit{ls}' = (\mathit{ls} \setminus \{\rho\} \setminus \mathit{ls}_e) \cup \mathit{ls}'_e \end{array}$$

  Since $\rho \notin \mathit{ls}'$, by rule [CS IN-SYNC] we obtain our desired result

  $$\begin{array}{l} \mathit{ls}' \cup \{\rho\} \vdash_{cs} \texttt{in-sync}\ \rho\, \mathcal{E}'[e'] \\ \mathit{ls}' \cup \{\rho\} = (\mathit{ls} \setminus \{\rho\} \setminus \mathit{ls}_e) \cup \mathit{ls}'_e \cup \{\rho\} \\ \qquad\qquad = (\mathit{ls} \setminus \mathit{ls}_e) \cup \mathit{ls}'_e \end{array}$$