Types for Precise Thread Interference
University of California at Santa Cruz Technical Report UCSC-SOE-11-22

Jaeheon Yi  Tim Disney  Stephen N. Freund  Cormac Flanagan
Computer Science Department  Computer Science Department  Computer Science Department
University of California, Santa Cruz  Williams College  University of California, Santa Cruz
Santa Cruz, CA 95064  Williamstown, MA 01267  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.

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

![Figure 1: Traveling Salesperson Algorithm](image-url)

```java
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  }
```
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 points. 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 ICC (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 ICC 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.

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:

```java
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:

```
Figure 2: Traveling Salesperson Algorithm with atomic blocks
```
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.

```java
... 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:

```java
... 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.

```java
... 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:

```java
... 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 ICC 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 $\alpha$ can be commuted out so that $\alpha$ executes serially, without interfering operations of other threads. In this case, we consider $\alpha$ a serializable transaction, or simply a transaction.

4. Effects for Cooperability

Our effect system characterize the behavior of each program sub-expression using two kinds of effects: mover effects and atomicity effects.

4.1 Mover Effects

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

$$\mu ::=_F \mid _Y \mid _M \mid _R \mid _L \mid _N$$

$F$: 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 ($\mu^*$) 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 $\sqsubseteq$ shown via the lattice below.
4.2 Atomicity Effects

Each program expression also has an atomicity effect \( \tau \) that summarizes whether the expression performs a yield operation.

\[
\tau ::= A \mid C
\]

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

\[
A \sqsubseteq C
\]

\[
\tau^+ \equiv \tau
\]

\[
\tau_1 ; \tau_2 \equiv \tau_1 \sqcup \tau_2
\]

4.3 Combined Effects

A combined effect \( \kappa \) is a pair of a mover effect \( \mu \) and an atomicity effect \( \tau \):

\[
\kappa ::= \tau \mu
\]

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:

\[
\tau_1 \mu_1 \sqsubseteq \tau_2 \mu_2 \iff \tau_1 \subseteq \tau_2 \text{ and } \mu_1 \sqsubseteq \mu_2
\]

\[
\tau_1 \mu_1 \sqcup \tau_2 \mu_2 \equiv \tau_1 \mu_3 \text{ where } \tau_3 = \tau_1 \sqcup \tau_2 \text{ and } \mu_3 = \mu_1 \sqcup \mu_2
\]

\[
(\tau \mu)^* \equiv \tau^* \mu^*
\]

The following diagram summarizes the resulting lattice of combined effects:

![Diagram 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: \textsc{yieldjava} Syntax

\[
P \in \text{Program} = \overline{\text{defn}}
\]

\[
\text{defn} \in \text{Definition} ::= \text{class } c \{ \text{ field } \text{ meth} \}
\]

\[
\text{field} \in \text{Field} ::= c \ f
\]

\[
\text{meth} \in \text{Method} ::= a \ e \ m(\tau_2) \{ e \}
\]

\[
e, \ell \in \text{Expr} ::= x \ | \ \text{null}
\]

\[
e, f \in \text{Field} ::= c, d \in \text{Name}
\]

\[
f \in \text{Name} = \text{Normal} \cup \text{Final} \cup \text{Volatile}
\]

\[
m \in \text{Name} \text{ or MethodName} \Rightarrow \text{Effect}
\]

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 \( \ell \) 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 \( \kappa \) or an effect conditional on whether a lock \( \ell \) is held:

\[
a ::= \kappa \mid \ell ? a_1 : a_2
\]

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

\[
(\ell ? a_1 : a_2)^* = \ell ? a_1^* : a_2^*
\]

\[
(\ell ? a_1 : a_2) \cdot a = \ell ? (a_1 : a) = \ell ? (a_1 : a) : (a_2 : a)
\]

\[
\ell ? a_1 : a_2 \sqcup a = \ell ? (a_1 \sqcup a) : (a_2 \sqcup a)
\]

\[
a \sqcup (\ell ? a_1 : a_2) = \ell ? (a \sqcup a_1) : (a \sqcup a_2)
\]

We also extend the effect ordering to conditional effects. To decide \( a_1 \sqsubseteq a_2 \), we use an auxiliary relation \( \sqsubseteq_n \), 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 \( a_1 \sqsubseteq_n a_2 \) and check \( a_1 \sqsubseteq_n a_2 \) recursively as follows:

\[
\kappa_1 \sqsubseteq \kappa_2 \iff \ell \notin n \Rightarrow a_1 \sqsubseteq_n \kappa_1 \sqsubseteq_n \ell = a_1 \sqsubseteq_n \kappa_1 \sqsubseteq_n a_2
\]

\[
\kappa_1 \sqsubseteq \kappa_2 \iff \ell \notin h \Rightarrow a_2 \sqsubseteq_n \kappa_2 \sqsubseteq_n \ell = a_2 \sqsubseteq_n \kappa_2 \sqsubseteq_n a_2
\]

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 \textsc{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 \textsc{yieldjava} syntax. A program \( P \) consists of a sequence of class definitions \( \text{defn} \). Each class definition \( \text{defn} \)

\[
\begin{array}{l}
P \in \text{Program} = \overline{\text{defn}} \\
\text{defn} \in \text{Definition} ::= \text{class } c \{ \text{ field } \text{ meth} \} \\
\text{field} \in \text{Field} ::= c \ f \\
\text{meth} \in \text{Method} ::= a \ e \ m(\tau_2) \{ e \} \\
e, \ell \in \text{Expr} ::= x \ | \ \text{null} \\
e, f \in \text{Field} ::= c, d \in \text{Name} \\
f \in \text{Name} = \text{Normal} \cup \text{Final} \cup \text{Volatile} \\
m \in \text{Name} \\
a \in \text{Effect} \\
\end{array}
\]
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(\tau \pi) \{ e \}
```

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

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

```
\textbf{synchronized} \ (e_1) \ \{ \ e_2 \ }
```

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

<table>
<thead>
<tr>
<th>Expression Form</th>
<th>Non-Yielding</th>
<th>Yielding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field Read</td>
<td>( e.f )</td>
<td>( e.f )</td>
</tr>
<tr>
<td>Field Write</td>
<td>( e.f = e )</td>
<td>( e.f = e )</td>
</tr>
<tr>
<td>Atomic Method Call</td>
<td>( e.m(\tau) )</td>
<td>( e.m(\tau) )</td>
</tr>
<tr>
<td>Non-Atomic Method Call</td>
<td>( e.m#(\tau) )</td>
<td>( e.m#(\tau) )</td>
</tr>
<tr>
<td>Lock Synchronization</td>
<td>( e.sync e )</td>
<td>( e.sync e )</td>
</tr>
</tbody>
</table>

### 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:

\[
E ::= e \mid E, e x
\]

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 \((P; E \vdash meth)\), class declarations \((P \vdash defn)\), and programs \((P \vdash OK)\). 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 truth expression sequentially composed with the join of the \texttt{then} and \texttt{else} branches. Similarly, the effect of a while loop \texttt{while} \( e_1 \ e_2 \) is the effect of the \texttt{test} \( e_1 \) composed with the iterative closure \((e_2; e_1)^\ast\) 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 \( f \) 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 RACE]** A racy field read is a non-mover operation, since it may conflict with concurrent accesses by other threads. A racy read \( e.f \) may be annotated with a yield point (if \( \gamma = \text{"\texttt{''}}\) or not (if \( \gamma = \text{"\texttt{''}}\)). We use the auxiliary function \( \llbracket \gamma \rrbracket \) to map \( \gamma \) to the corresponding effect:

\[
\begin{align*}
  1 : & \text{OptYield} \to \text{Effect} \\
  \overline{..} & = \text{AF} \\
  .. & = \text{CY}
\end{align*}
\]

Thus, if the expression \( e \) has effect \( a \), then the non-yielding racy access \( e.f \) has effect \( (a; AF; AN) \), whereas the yielding racy access \( e.f \) has effect \( (a; CY; AN) \).
- **[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 \( \ell, \text{sync} \ e \) first checks that \( \ell \) is a valid lock expression \((P; E \vdash \text{lock} \ \ell)\), meaning that \( \ell \) must have effect \( AF \) to guarantee that it always denotes the same lock at run time.

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

<table>
<thead>
<tr>
<th>( a )</th>
<th>( S(\ell, \gamma, a) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \kappa )</td>
<td>( \ell ? \kappa : \llbracket \gamma \rrbracket ; \text{AR}; \kappa; \text{AL} )</td>
</tr>
<tr>
<td>( \ell \ ? \ \ell_1 : a_1 )</td>
<td>( S(\ell, \gamma, a_1) )</td>
</tr>
</tbody>
</table>
| \( \ell' \ ? \ \ell_1 : a_2 \) | \( \ell' ? S(\ell, \gamma, a_1) ; S(\ell, \gamma, a_2) \) if \( \ell \neq \ell' \)

If the body of the synchronized statement has a basic effect \( \kappa \) and the lock \( \ell \) is already held, then the synchronized statement also has effect \( \kappa \), 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 \( \kappa \) and the lock is not already held, then the synchronized statement has effect \( \llbracket \gamma \rrbracket ; \text{AR}; \kappa; \text{AL} \), since the execution consists of a potential yield point, followed by a right-mover (the acquire), followed by \( \kappa \) (the body), followed by a left-mover (the release).
Figure 4: YIELDJAVA Type Rules

\[ P; E \vdash c : a \]

**EXP VAR**
\[
P; E \vdash E
\]
\[
P; E \vdash c
\]
\[
E = E_1 \cdot E_2
\]
\[
E = c \cdot E_2
\]
\[
P; E \vdash x : c \cdot AF
\]

**EXP NULL**
\[
P; E \vdash E
\]
\[
P; E \vdash c
\]
\[
P; E \vdash \text{null} : c \cdot AF
\]

**EXP IF**
\[
P; E \vdash E_1 : d \cdot a_1
\]
\[
P; E \vdash E_2 : c \cdot a_i
\]
\[
\forall i \in \{2, 3\}
\]
\[
P; E \vdash \text{if } E_1 \cdot E_2 \cdot c \cdot (a_1 \cup a_3)
\]
\[
P; E \vdash \text{while } E_1 \cdot \text{E2} : \text{Unit} \cdot (a_1 \cup (a_2 \cup a_3))
\]

**EXP REF**
\[
P; E \vdash e \cdot c 
\]
\[
\text{class } c \{ \ldots d f \ldots \} \in P
\]
\[
f \in \text{Normal} \quad \Rightarrow \quad a' = \text{AM}
\]
\[
f \in \text{Final} \quad \Rightarrow \quad a' = \text{AF}
\]
\[
P; E \vdash e \cdot f : d \cdot (a; a')
\]

**EXP ASSIGN**
\[
P; E \vdash c : a
\]
\[
P; E \vdash e' : d \cdot a'
\]
\[
\text{class } c \{ \ldots d f \ldots \} \in P
\]
\[
f \in \text{Final} \quad \Rightarrow \quad a' = \text{AF}
\]
\[
P; E \vdash (e, f = e') : d \cdot (a; a'; \text{AM})
\]

**EXP REF RACE**
\[
P; E \vdash e \cdot c 
\]
\[
\text{class } c \{ \ldots d f \ldots \} \in P
\]
\[
f \in \text{Volatilize}
\]
\[
P; E \vdash e \cdot f : d \cdot (a; [\gamma]; \text{AN})
\]

**EXP FORK**
\[
P; E \vdash c : a
\]
\[
P; E \vdash e : \text{Unit} \cdot \text{AL}
\]
\[
P; E \vdash \text{fork } e : c 
\]

**ENV EMPTY**
\[
P; E \vdash c
\]
\[
\text{x} \not\in \text{dom}(E)
\]
\[
P; E \vdash (E, c \cdot x)
\]

**ENV X**
\[
P; E \vdash c
\]
\[
x \not\in \text{dom}(E)
\]
\[
P; E \vdash (E, c \cdot x)
\]

**CLASS NAME**
\[
P; E \vdash E
\]
\[
\text{class } c \{ \ldots \} \in P
\]
\[
P; E \vdash c
\]

**AT BASE**
\[
P; E \vdash E
\]
\[
P; E \vdash a
\]
\[
\forall i \in \{1, 2\}
\]
\[
P; E \vdash a \uparrow a'
\]

**AT COND**
\[
P; E \vdash E
\]
\[
P; E \vdash \text{lock } \ell
\]
\[
\forall i \in \{1, 2\}
\]
\[
P; E \vdash a \uparrow a'
\]

**LOCK EXP**
\[
P; E \vdash e : c \cdot AF
\]
\[
P; E \vdash \text{lock } e
\]
\[
P; E \vdash c
\]
\[
P; E \vdash \text{lock } c
\]
\[
P; E \vdash \ell \cdot a_1 \uparrow a_2
\]

**LIFT BASE**
\[
P; E \vdash E
\]
\[
P; E \vdash a \uparrow a'
\]
\[
P; E \vdash \text{lock } \ell
\]
\[
P; E \vdash (E, c) \cdot a_1 \uparrow a_2
\]

**LIFT GOOD LOCK**
\[
P; E \vdash E
\]
\[
P; E \vdash a \uparrow a'
\]
\[
\forall i \in \{1, 2\}
\]
\[
P; E \vdash (\ell \cdot a_1 \uparrow a_2) \uparrow ((\ell \cdot a_1 \uparrow a_2)
\]

**LIFT BAD LOCK**
\[
P; E \vdash E
\]
\[
P; E \vdash a \uparrow a'
\]
\[
\forall i \in \{1, 2\}
\]
\[
P; E \vdash (\ell \cdot a_1 \uparrow a_2) \uparrow ((\ell \cdot a_1 \uparrow a_2)
\]

**METHOD**
\[
P; E \vdash d \cdot x \cdot c : a
\]
\[
P; E \vdash d \cdot x \cdot a
\]
\[
a \subseteq a
\]
\[
P; E \vdash a \cdot c \cdot m \cdot (d \cdot x) \{ a \}
\]

**CLASs**
\[
P; E \vdash c \cdot \text{field}_1 \cdot \text{meth}_1
\]
\[
\forall i \in \{1, 2\}
\]
\[
P; E \vdash \text{field}_1 \cdot \text{meth}_1
\]

**DEFIN**
\[
P; E \vdash \text{defn}
\]
\[
P; E \vdash \text{defn}
\]
\[
\forall i \in \{1, 2\}
\]
\[
P; E \vdash \text{defn}
\]

**OK**
\[
P; E \vdash \text{OK}
\]
\[
P; E \vdash \text{OK}
\]

**program**
\[
P; E \vdash \text{program}
\]
\[
P; E \vdash \text{program}
\]
\[
\forall i \in \{1, 2\}
\]
\[
P; E \vdash \text{program}
\]

**ClassesOnce**
\[
P; E \vdash \text{ClassesOnce}
\]
\[
P; E \vdash \text{ClassesOnce}
\]
\[
\forall i \in \{1, 2\}
\]
\[
P; E \vdash \text{ClassesOnce}
\]

**FieldsOnce**
\[
P; E \vdash \text{FieldsOnce}
\]
\[
P; E \vdash \text{FieldsOnce}
\]
\[
\forall i \in \{1, 2\}
\]
\[
P; E \vdash \text{FieldsOnce}
\]

**MethodsOnce**
\[
P; E \vdash \text{MethodsOnce}
\]
\[
P; E \vdash \text{MethodsOnce}
\]
\[
\forall i \in \{1, 2\}
\]
\[
P; E \vdash \text{MethodsOnce}
\]
If the body has conditional effect $\ell' \cdot a_1 : a_2$, where $\ell$ is the lock being acquired by this synchronized statement, then we ignore $a_2$ and recursively apply $S$ to $a_1$, since $\ell$ is held within the synchronized body.

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

[EXP LET] This rule for let $x = e_1 \text{ in } e_2$ 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 = e_1 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:

[EXP LET 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 $a'[\text{this} := e, x_i := e_{i \in 1..n}]$, 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 FORK] A fork expression $\text{fork } c$ creates a new thread to evaluate $c$. 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 $\sigma$ 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 $\sigma$ 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 $\text{final}$ and $\text{volatile}$ to classify fields as either $\text{Final}$ or $\text{Volatile}$; all other fields are considered $\text{Normal}$. We also introduce one new modifier, $\text{racy}$, to capture intentionally racy $\text{Normal}$ fields. Our implementation assumes that correct field annotations are provided for the input program. Such annotations could be generated using RccJava [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 (\text{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.
24. The fields (such as the Write-Guarded Fields. similarly. The rules for computing the effects of these operations, races, which are avoided when possible, large programs often have some intentional non-moving memory access: consider the consequences of Java’s relaxed memory model [37]). The racy nature of the read (and that the programmer needs to dots as usual indicate a yield point, and the trailing # identifies the racy array write (lock held) e1 e2 = e3 a1; a2; CL 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 arrays1.

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 ⊆ a in the following class definitions:

```java
class C {
    a t m() { ... }
}
class D extends C {
    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: (CY: AN) = 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. 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 programs 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 a 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.
### Table 6. Interference Points and Unintended Yields

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (lines)</th>
<th>Annot. Time (min.)</th>
<th>Annot. Count</th>
<th>Interference Points</th>
<th>Unintended Yields</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util.zip.Inflater</td>
<td>317</td>
<td>9</td>
<td>4</td>
<td>Preemptive: 36 Race: 12 Atomic: 0 AtomRace: 0 Cooperative: 0</td>
<td>0</td>
</tr>
<tr>
<td>java.util.zip.Deflater</td>
<td>381</td>
<td>7</td>
<td>8</td>
<td>Preemptive: 49 Race: 13 Atomic: 0 Cooper: 0</td>
<td>0</td>
</tr>
<tr>
<td>elevator</td>
<td>1,447</td>
<td>30</td>
<td>64</td>
<td>Preemptive: 454 Race: 146 Atomic: 241 AtomRace: 60 Cooperative: 25</td>
<td>0</td>
</tr>
<tr>
<td>moldyn-fixed</td>
<td>1,352</td>
<td>10</td>
<td>39</td>
<td>Preemptive: 983 Race: 130 Atomic: 657 AtomRace: 37 Cooperative: 30</td>
<td>0</td>
</tr>
</tbody>
</table>

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 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.getChildren() 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
ArrayIndexOutOfBoundsException can occur if the size of the data array is not properly initialized before it is copied.

Vector data, potentially resulting in an incorrectly initialized current thread in between requesting the size and copying the data. Yield annotations highlight that the accesses to the data array may not properly coordinate because yield annotations must be added by a concurrent thread in between requesting the size and copying the data. This bug leads to potential races on all data accesses.

Single thread, we could remove the yield on the synchronized methods. However, each thread creates its own Vector object, and thus acquiring a Vector object, 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 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 ArrayIndexOutOfBoundsException. Similar pitfalls for the methods removeAll(c) and retainAll(c) are also caught by ICC.

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. This problem is obvious when using ICC 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.

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

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

Moldyn. The moldyn benchmark uses a barrier object to synchronize the actions of multiple computation threads. The barrier object maintains an array 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 of yield annotations were necessary to document all such cases. Again, we report on the corrected version moldyn-fixed in Figure 6.
8. Related Work

Cooperability. Cooperative multithreading is a thread execution model in which context switching between threads may occur only at \texttt{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]: \texttt{yield} statements demarcate where thread interference is permitted. A critical difference is that these \texttt{yield} statements are enforced at run time to provide serializability via transactional memory techniques; in contrast, the ICC checker guarantees 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 ICC 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 [6, 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 scheduling poses.

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 difficult 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


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 $\rho$ and the in-sync construct. We assume that addresses are divided into distinct sets, one for each type of object. The membership relation $\rho \in Addr_c$ indicates that the object at address $\rho$ is of type $c$. The expression in-sync $\rho e$ describes an expression $e$ that is executing while holding the lock $\rho$.

$$
e \in \text{Expr} ::= \cdots \mid \text{in-sync} \rho e$$
$$v \in \text{Value} ::= \rho \mid \text{null}$$
$$\rho \in \text{Addr} = \bigcup_{c \in \text{ClassName}} \text{Addr}_c$$

$$T \in \text{ThreadState} ::= \{ e \mid \text{ready} e \mid \text{wrong} \}$$
$$t \in \text{Thread} = \{ 1, 2, 3, \ldots \}$$
$$\text{obj} \in \text{Object} ::= \{ T = v \}^o$$
$$o \in \text{Lock} ::= \bot \mid \text{Thread}$$
$$\sigma \in \text{State} = \text{Program}$$
$$\cup (\text{Addr} \mapsto \text{Object})$$
$$\cup (\text{Thread} \mapsto \text{ThreadState})$$

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

A thread state $T$ is either:

- an expression $e$,
- ready $e$, which describes a newly created thread ready to execute an expression $e$, or
- 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 implicitly associated with it at run time).

Finally, the run-time state $\sigma$ 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 $E$, an expression with a “hole” $[]$, to indicate the next subexpression to evaluate. We use the form $\sigma[t \rightarrow T]$ to indicate a state that agrees with $\sigma$ at all threads except $t$, where it maps to $T$. Similarly, we use $\sigma[\rho \rightarrow \{ T = v \}^o]$ and $\sigma[\rho.f \rightarrow v]$ to indicate updates to addresses and object fields. Evaluation finishes when each thread state is either a value or wrong.

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

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

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 $\sigma$ (or simply yielding if the context is clear) if for $\sigma(t) = T$,

1. $T = \text{read}y e$
2. $T \in \text{Value}$,
3. $T = \text{wrong}$,
4. $T = E[\rho.f]$,
5. $T = E[\rho..f = v]$,
6. $T = E[\rho..\text{sync} e]$,
7. $T = E[\rho..\text{m}(\tau)]$, or
8. $T = E[\rho..\text{m#}(\tau)]$.

The thread $t$ is cooperatively enabled in a state $\sigma$ if for all $t'$ in $\text{Dom}(\sigma) \setminus \{ t \}$, we have $t'$ is yielding in $\sigma$. 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 wrong), or has not started yet. A state is yielding if all threads are yielding in that state.
Figure 11: YieldJava Semantics

**Evaluation contexts**

\[ E ::= [ \ldots ] | \text{new} c(\tau, E, v) \ | E.f \ | E.g \ | v.g = e \ | v.g \ = E \]
\[ E \cdot m(\tau) \ | E \cdot m(\tau) \ | v.m(\tau, E, v) \ | v.m(\tau, E, \tau) \ | \text{let } x = E \text{ in e} \ | \text{if } E \ e \ | E \cdot \text{sync} \ e \ | \text{in-sync } E \]

**Transition rules**

\[ \sigma[t \mapsto E[\rho, f]] \rightarrow_t \ \sigma[t \mapsto E[v]] \quad \text{if } \sigma(\rho) = \{ \ldots, f = v, \ldots \}^o \quad \text{[RED READ]} \]
\[ \sigma[t \mapsto E[\rho, f = v]] \rightarrow_t \ \sigma[t \mapsto E[v], \rho, f \mapsto v] \quad \text{[RED WRITE]} \]
\[ \sigma[t \mapsto E[\rho, m(v_{1 \ldots n})]] \rightarrow_t \ \sigma[t \mapsto E[e[\text{this} := \rho, x_i := v_i^{E \cdot 1 \ldots n}]]] \quad \text{if } \rho \in \text{Addr}_c \text{ and class } c \{ \ldots \text{a } d' m(\bar{d} e) \{ e \} \ldots \} \in P \quad \text{[RED INVOKE]} \]
\[ \sigma[t \mapsto E[\rho, m(\#(v_{1 \ldots n})] \rightarrow_t \ \sigma[t \mapsto E[e[\text{this} := \rho, x_i := v_i^{E \cdot 1 \ldots n}]]] \quad \text{if } \rho \in \text{Addr}_c \text{ and class } c \{ \ldots \text{a } d' m(\bar{d} e) \{ e \} \ldots \} \in P \quad \text{[RED INVOKE COMPOUND]} \]
\[ \sigma[t \mapsto E[\text{new} c(v_{1 \ldots n})]] \rightarrow_t \ \sigma[t \mapsto E[\rho], \rho \mapsto \{ f_i = v_i^{E \cdot 1 \ldots n} \}^+ \quad \text{if } \rho \not\in \text{dom}(\sigma) \text{ and } \rho \in \text{Addr}_c \text{ and class } c \{ \bar{t} f \ldots \} \in P \quad \text{[RED NEW]} \]
\[ \sigma[t \mapsto E[\rho, \text{sync } e]], \rho \mapsto \{ f = v \}^+ \rightarrow_t \ \sigma[t \mapsto E[\text{in-sync } \rho e], \rho \mapsto \{ f = v \}^+] \quad \text{[RED SYNC]} \]
\[ \sigma[t \mapsto E[\rho, \text{sync } e]], \rho \mapsto \{ f = v \}^+ \rightarrow_t \ \sigma[t \mapsto E[e], \rho \mapsto \{ f = v \}^+] \quad \text{[RED SYNC REENTRANT]} \]
\[ \sigma[t \mapsto E[\text{in-sync } \rho v]], \rho \mapsto \{ f = v \}^+ \rightarrow_t \ \sigma[t \mapsto E[v], \rho \mapsto \{ f = v \}^+] \quad \text{[RED IN-SYNC]} \]
\[ \sigma[t \mapsto E[\text{fork } e]] \rightarrow_t \ \sigma[t \mapsto E[\text{null}], t' \mapsto \text{ready } e] \quad \text{if } t' \not\in \text{Dom}(\sigma) \quad \text{[RED FORK]} \]
\[ \sigma[t \mapsto \text{ready } e] \rightarrow_t \ \sigma[t \mapsto [e]] \quad \text{[RED READY]} \]
\[ \sigma[t \mapsto E[\text{let } x = v \text{ in } e]] \rightarrow_t \ \sigma[t \mapsto E[e[x := v]]] \quad \text{[RED LET]} \]
\[ \sigma[t \mapsto E[\text{if } v \text{ e}_2 \text{ e}_3]] \rightarrow_t \ \sigma[t \mapsto E[e_2]] \quad \text{if } v \not= \text{null} \quad \text{[RED IF-NULL]} \]
\[ \sigma[t \mapsto E[\text{if } \text{null } e_2 \text{ e}_3]] \rightarrow_t \ \sigma[t \mapsto E[e_3]] \quad \text{[RED IF-NULL]} \]
\[ \sigma[t \mapsto E[\text{while } e_1 \text{ e}_2]] \rightarrow_t \ \sigma[t \mapsto E[e_1 \text{ (e}_2 \text{ while } e_1 \text{ e}_2) \text{null}]] \quad \text{[RED WHILE]} \]
\[ \sigma[t \mapsto E[e]] \rightarrow_t \ \sigma[t \mapsto \text{wrong]} \quad \text{if } e \in \{ \text{null, f, null, f = v, null, m(\tau), null, m(\tau), null, sync } e' \} \quad \text{[RED WRONG]} \]

**Transition relations**

(preemptive semantics) \( \sigma \rightarrow \sigma' \) if \( \sigma \rightarrow_t \sigma' \)

(cooperative semantics) \( \sigma \rightarrow_c \sigma' \) if \( \sigma \rightarrow_t \sigma' \) and \( \sigma(t) \) is cooperatively enabled
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:

\[ E ::= e | E, c x | E, c \rho \]

- Addresses have effect AF, the constant effect, and have the appropriate type c for the object address \( \rho \).

- The rule [EXP INSYNC] first checks that the lock expression \( \rho \) 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 \( SI(\rho, a) \).

\[
\begin{array}{c|c}
\text{a} & \text{SI(\rho, a)} \\
\hline
\kappa & \kappa; \text{AL} \\
\rho \? \alpha_1 : a_2 & \rho \? \alpha_1 : a_2 \\
\rho' \? \alpha_1 : a_2 & \text{SI}(\rho, a_1); \text{SI}(\rho, a_2) \text{ if } \rho \neq \rho'
\end{array}
\]

Briefly, if \( a \) is a combined effect \( \kappa \), then the overall effect is \( \kappa \) sequentially followed by AL, the effect for a lock release operation. If \( a \) is an effect conditional on holding lock \( \rho \), and the in-sync expression is also holding \( \rho \), then we simplify \( a \) by having \( SI \) recurse down the holding branch. However, if \( a \) is a conditional effect on some different lock \( \rho' \), then we recurse \( SI \) 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 \( \text{ready} e \), the rule checks if \( e \) is well-formed in its antecedent. Otherwise, the thread state is \text{wrong}, which is always well-formed.

- The judgment \( P; E \vdash \text{obj} : c \) 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.

\textbf{THEOREM 1 (Preservation).} If \( \vdash \sigma \) and \( \sigma \rightarrow \sigma' \) then \( \vdash \sigma' \).

\textbf{Proof} Suppose \( \sigma \rightarrow^* \sigma' \). Let:

\[ \sigma = P \cup \{ \rho_i \mapsto \text{obj}_i \in c_1..n \} \cup \{ t_j \mapsto T_{j} \in c_1..m \} \]

From the rule [STATE] we have:

\[ P \vdash \text{OK} \]
\[ E = c_1 \rho_1, ..., c_n \rho_n \]
\[ \rho_i \in \text{Addr}_{c_i}, \forall i \in 1..n \]
\[ P; E \vdash \text{obj}_i : c_i, \forall i \in 1..n \]
\[ P; E \vdash T_j, \forall j \in 1..m \]

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

\[ P; E' \vdash \text{obj}_i : c_i, \forall i \in 1..n' \]
\[ P; E' \vdash T_j, \forall j \in 1..m' \]

in the environment \( E' \) capturing all allocated objects in \( \sigma' \):

\[ E' = c_1 \rho_1, ..., c_n \rho_n' \]

such that \( \forall i \in 1..n', \rho_i \in \text{Addr}_{c_i} \).

Proof is by case analysis on the reduction rule used for \( \sigma \rightarrow^* \sigma' \).

- [READ] In this case we have:

\[ \sigma(t) = \mathcal{E}[\rho, f] \]
\[ \sigma'(t) = \mathcal{E}[v] \]
\[ \sigma(\rho) = \{ \ldots f = v \ldots \}^{\sigma} \]

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:

\[ P; E \vdash \mathcal{E}[\rho, f] : c \cdot a \]

Figure 12: Additional \textsc{YieldJava} Type Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[EXP ADDR]</td>
<td>( P \vdash E )</td>
</tr>
<tr>
<td>( E = E_1, c \rho, E_2 )</td>
<td>( \rho \in \text{Addr}_c )</td>
</tr>
<tr>
<td>( P; E \vdash c \cdot AF )</td>
<td></td>
</tr>
</tbody>
</table>
| [EXP INSYNC] | \( P; E \vdash \text{lock} \rho \)
| \( P; E \vdash \text{in-sync} \rho e : c \cdot SI(\rho, a) \) | |
| [ENV ADDR] | \( P \vdash E \)
| \( P \vdash (E, c \rho) \) | |
| [THREAD OK] | \( P \vdash E \)
| \( P \vdash \text{e} : c \cdot a \) | |
| [THREAD READY] | \( P \vdash \text{e} : c \cdot a \)
| \( P; E \vdash \text{ready} e \) | |
| [THREAD WRONG] | \( P \vdash \text{e} : c \cdot a \)
| \( P \vdash \text{e} \) | |
| [OBJECT] | \( c \) \{ \( d_i, f_i \in c_i \ldots \} \in P \)
| \( P; E \vdash v_i : d_i \cdot c_i \) \( \forall i \in 1..n \) | |
| \( P; E \vdash f_i = v_i \in c_i \ldots \}^{\sigma} : c \) | |

\[ [\text{STATE}] \]
\[ P \vdash \text{OK} \]
\[ E = c_1 \rho_1, ..., c_n \rho_n \]
\[ \rho_i \in \text{Addr}_{c_i}, \forall i \in 1..n \]
\[ P; E \vdash \text{obj}_i : c_i, \forall i \in 1..n \]
\[ P; E \vdash T_j, \forall j \in 1..m \]

\[ \vdash P \cup \{ \rho_i \mapsto \text{obj}_i \in c_1..n \} \cup \{ t_j \mapsto T_{j} \in c_1..m \} \]
By Lemma 2 we have
\[ P; E \vdash \rho \cdot f : c_{\text{rd}} \cdot a_{\text{rd}} \]
Since this can only be concluded by \([\text{EXP REF}]\) or \([\text{EXP REF RACE}]\)
we also know
\[ P; E \vdash \rho \cdot d \cdot \text{AF} \]
By rules \([\text{EXP REF}]\) and \([\text{EXP REF RACE}]\), we know that \(a_{\text{rd}}\) must be
one of \(\text{AF}, \text{AM}, \text{AN},\) or \(\text{CL}\). Since \(\text{AF}\) is a subeffect of any one
of these effects, \(a_{\text{rd}} \subseteq a_{\text{rd}}\) and we can conclude by Lemma 4
\[ P; E \vdash E[t] : c \cdot a' \]
• [RED WRITE] For this case we have:
\[ \sigma(t) = \mathcal{E}[\rho \cdot f = v] \]
\[ \sigma'(t) = \mathcal{E}[v] \]
\[ \sigma'(\rho) = \{ \dots \cdot f = v \ldots \}^\sigma \]
Since the object map in \(\sigma'\) 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:
\[ P; E \vdash \sigma'(t) \]
\[ P; E \vdash \sigma'(\rho) \]
From [THREAD OK] we know:
\[ P; E \vdash E[\rho \cdot f = v] : c \cdot a \]
By Lemma 2 we have:
\[ P; E \vdash \rho \cdot f = v : c_{\text{wr}} \cdot a_{\text{wr}} \]
This judgment can only be concluded with \([\text{EXP ASSIGN}]\) or
\([\text{EXP ASSIGN RACE}]\), so we also know:
\[ P; E \vdash \rho \cdot d \cdot \text{AF} \]
\[ P; E \vdash v : c_{\text{wr}} \cdot \text{AF} \]
By rules \([\text{EXP ASSIGN}]\) and \([\text{EXP ASSIGN RACE}]\), we know \(a_{\text{wr}}\) must be
one of \(\text{AM}, \text{AN},\) or \(\text{CL}\). Since we know \(P; E \vdash v : c_{\text{wr}} \cdot \text{AF}\)
and \(\text{AF} \subseteq a_{\text{wr}}\), by Lemma 4 we can conclude:
\[ P; E \vdash E[t] : d \cdot a' \]
Thus we have shown \(P; E \vdash \sigma'(t)\). To show \(P; E \vdash \sigma'(\rho)\)
we note that \(P; E \vdash \sigma(\rho) : \text{equiv}\) and \(\sigma'(\rho) = \{ \dots \cdot f = v \ldots \}^\sigma\).
Since we know by \([\text{EXP ASSIGN}]\) and \([\text{EXP ASSIGN RACE}]\) that
\(P; E \vdash v : d_2 \cdot \text{AF}\) we can conclude by \([\text{OBJECT}]\):
\[ P; E \vdash \sigma'(\rho) : c_a \]
• [RED INVOKE] In this case, let \(\theta = \mathcal{E}[\text{this} := \rho \cdot x_i := v_{i \in 1..n}]\),
where
\[ \sigma(t) = \mathcal{E}[\rho \cdot m(v_{1..n})] \]
\[ \sigma'(t) = \mathcal{E}[\theta(t)] \]
\[ \rho \in \text{Addr}_c \]
\[ \text{class } c \{ \ldots d_{m} \cdot m(d_i \cdot x_i \in 1..n) \{ e \} \ldots \} \subseteq P \]
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:
\[ P; E \vdash E[\rho \cdot m(v_{1..n})] : d \cdot a \]
By Lemma 2 we have:
\[ P; E \vdash \rho \cdot m(v_{1..n}) : d_m \cdot a_{\text{iv}} \]
This can only be concluded by the rules \([\text{EXP INVOKE}]\) so we also have:
\[ P; E \vdash \rho \cdot c \cdot \text{AF} \]
\[ P; E \vdash v_i : d_i \cdot a_i \quad \forall i \in 1..n \]
\[ P; E \vdash \theta(a_m) \vdash a'' \]
\[ a_{iv} = (\text{AF}; a_1; \ldots; a_n; a'') \]
\[ \text{class } c \{ \ldots M \ldots \} \subseteq P \]
\[ M = a_m \cdot \text{ld}_{m} \cdot m(d_i \cdot x_i \in 1..n) \{ e \} \]
From \([\text{CLASS}]\) we know \(P; E \vdash \text{this} := \rho \cdot x_i := v_{i \in 1..n}\)
\[ P; d_i \cdot x_i \in 1..n \vdash a_m \]
\[ a_m' \subseteq a_m \]
By Lemma 6:
\[ P; E \vdash \theta(c) : d_m \cdot \theta(a_{\text{m}}) \]
Since \(a_{\text{m}}' \subseteq a_{\text{m}}\), by Lemma 1 we have \(\theta(a_{\text{m}}') \subseteq \theta(a_{\text{m}})\) and
therefore \(\theta(a_{\text{m}}') \subseteq a_{\text{m}}\). Thus by Lemma 4:
\[ P; E \vdash E[\text{this} := \rho \cdot x_i := v_{i \in 1..n}] : d \cdot a' \]
• [RED INVOKE COMPOUND] Similar to previous case.
• [RED NEW] In this case we have:
\[ \sigma(t) = \mathcal{E}[\text{new } c(v_{1..n})] \]
\[ \sigma'(t) = \mathcal{E}[\rho] \]
\[ \sigma'(\rho) = \{ f_i = v_{i \in 1..n} \}^{\rho} \]
\[ \rho \notin \text{Dom}(\sigma) \]
\[ \rho \in \text{Addr}_c \]
\[ \text{class } c \{ d_i \cdot f_i \in 1..n \ldots \} \subseteq P \]
Since \(\rho\) is the only new address in the environment, we have
\(E' = (E, c \cdot \rho)\). Since all other threads do not change it is sufficient
to show:
\[ P; E' \vdash \sigma'(t) \]
\[ P; E' \vdash \sigma'(\rho) : c \]
From [THREAD OK] we have:
\[ P; E \vdash E[\text{new } c(v_{1..n})] : d \cdot a \]
By Lemma 2 we know:
\[ P; E \vdash \text{new } c(v_{1..n}) : c \cdot a_{\text{new}} \]
This can only be concluded by \([\text{EXP NEW}]\) so we also know:
\[ P; E \vdash v_i : d_i \cdot \text{AF} \quad \forall i \in 1..n \]
\[ a_{\text{new}} = (\text{AF}; \text{AM}) \]
From the transition rules we know that \(\rho \in \text{Addr}_c\), so we can conclude,
via rule \([\text{EXP ADDR}]\),
\[ P; E' \vdash \rho : c : a_{\rho} \]
\[ a_{\rho} \subseteq \text{AF} \]
Since \(a_{\rho} \subseteq a_{\text{new}}\) we can conclude from Lemma 4:
\[ P; E' \vdash \sigma'(\rho) : c \cdot a'_\rho \]
So we have shown \(P; E' \vdash \sigma'(t)\). To show \(P; E' \vdash \sigma'(\rho) : c \cdot a'_\rho\)
we note that from the transition rules we know \(\rho \in \text{Addr}_c\) and
\[ \text{class } c \{ d \cdot f \ldots \} \subseteq P. \]
Also from \([\text{EXP NEW}]\) we know
\[ P; E \vdash v_i : d_i \cdot a_i \quad \forall i \in 1..n \]
Therefore by \([\text{OBJECT}]\) we can conclude
\[ P; E' \vdash \{ \bar{f} = v \}^{\rho} : c \]
So we have shown:
\[ P; E' \vdash \sigma'(\rho) : c \]
• [RED SYNC] In this case we have:
\[ \sigma(t) = \mathcal{E}[\rho \cdot \text{sync } e] \]
\[ \sigma'(\rho) = \{ \bar{f} = \bar{v} \}^{\rho} \]
\[ \sigma(t) = \mathcal{E}[\text{in-sync } \rho \cdot e] \]
\[ \sigma'(\rho) = \{ \bar{f} = \bar{v} \}^{\rho} \]
Since only the lock state of the object at $\rho$ changes, we have $E' = E$. Since all other threads and objects besides $\rho$ do not change it is sufficient to show:

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

From [THREAD OK] we know:

$$P; E \vdash E[\rho, \text{sync} \ e] : c \cdot a$$

By Lemma 2 we have:

$$P; E \vdash \rho, \text{sync} \ e : c_{\text{sync}} \cdot a_{\text{sync}}$$

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

$$P; E \vdash \text{lock} \ \rho$$
$$P; E \vdash e : c_{\text{sync}} \cdot a_e$$
$$a_{\text{sync}} = S(\rho, \gamma, a_e)$$

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

$$P; E \vdash \text{in-sync} \ \rho \ e : c_{\text{sync}} \cdot a_{\text{ins}}$$
$$a_{\text{sync}} = ST(\rho, a_e)$$

Observe that $\rho$ is not held by thread $t$ in state $\sigma$. Then since the following satisfies the preconditions of Lemma 14,

$$P; E \vdash \text{lock} \ \rho$$
$$P; E \vdash a_e$$
$$\rho \in n$$

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

$$[\gamma]; ST(\rho, a_e) \subseteq_n S(\rho, \gamma, a_e)$$

Since the following satisfies the preconditions of Lemma 5,

$$P; E \vdash E[\rho, \text{sync} \ e] : c \cdot a$$
$$P; E \vdash \rho, \text{sync} \ e : c_{\text{sync}} \cdot S(\rho, \gamma, a_e)$$
$$\rho, \text{sync} \ e \text{ is not a value}$$
$$P; E \vdash \text{in-sync} \ \rho \ e : c_{\text{sync}} \cdot ST(\rho, a_e)$$
$$\exists n \ . [\gamma]; ST(\rho, a_e) \subseteq_n S(\rho, \gamma, a_e)$$

we obtain our desired result

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

and hence we have shown

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

From [OBJECT] we know

$$P; E \vdash \{ \overline{\tau} \}^t : c_o$$
$$\text{class } c_o \{ \overline{\tau} \}^t \chi \in P$$
$$P; E \vdash \tau : \overline{\pi}$$

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

• **[RED SYNC REENTRANT]** In this case we have

$$\sigma(t) = E[\rho, \text{sync} \ e]$$
$$\sigma(\rho) = \{ \overline{\tau} \}^t$$
$$\sigma'(t) = E[e]$$
$$\sigma'(\rho) = \{ \overline{\tau} \}^t$$

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

$$P; E \vdash E[\rho, \text{sync} \ e] : c \cdot a$$

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

$$P; E \vdash \rho, \text{sync} \ e : c_{\text{sync}} \cdot a_{\text{sync}}$$

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

$$P; E \vdash \text{lock} \ \rho$$
$$P; E \vdash e : c_{\text{sync}} \cdot a_e$$
$$a_{\text{sync}} = S(\rho, \gamma, a_e)$$

By Lemma 15, we have

$$a_e \subseteq_\rho S(\rho, \gamma, a_e)$$

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

$$AF; a_e \vdash \text{lock} \ \rho$$
$$S(\rho, \gamma, a_e)$$

By Lemma 5, we may conclude

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

and hence we have shown our desired result

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

From [THREAD OK] we know:

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

By Lemma 2 we have:

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

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

$$P; E \vdash \text{lock} \ \rho$$
$$P; E \vdash v : c_{\text{ins}} \cdot AF$$
$$a_{\text{ins}} = ST(\rho, AF) = AF; AL$$

Since $AF \subseteq AF; AL$, by Lemma 4 we have $P; E \vdash E[v] : c \cdot a'$. Thus we have shown $P; E \vdash \sigma'(t)$.

From [OBJECT] we know

$$P; E \vdash \{ \overline{\tau} \}^t : c_o$$
$$\text{class } c_o \{ \overline{\tau} \}^t \chi \in P$$
$$P; E \vdash \tau : \overline{\pi}$$

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

• **[RED FORK]** In this case we have:

$$\sigma(t) = E[\text{fork} \ e]$$
$$\sigma(\rho) = E[\text{null}]$$
$$\sigma'(t') = \text{ready} \ e$$

where $t' \not\in \text{dom}(\sigma)$.

The object map remains constant, hence $E = E'$. Since all other threads other than $t$ and $t'$ and the object map do not change it is sufficient to show:

$$P; E \vdash \sigma'(t)$$
$$P; E \vdash \sigma'(t')$$
From [thread ok] we know
\[ P; E \vdash E[\text{fork } e] : c \cdot a \]
By Lemma 2 we have
\[ P; E \vdash \text{fork } e : c_f \cdot a_f \]
Since this can only be concluded by [exp fork] we also know:
\[ P; E \vdash e : d \cdot a_e \]
\[ c_f = \text{Unit} \]
\[ a_f = AL \]
From [exp null] we can conclude \[ P; E \vdash \text{null}; \text{Unit} \cdot AF \]
Since AF \subseteq AL, we can conclude by Lemma 4:
\[ P; E \vdash E[\text{null} \cdot AF] \]
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
\[ \sigma(t) = \text{ready } e \]
\[ \sigma'(t) = [e] \]
Since the object map does not change, hence \[ E = E' \]. 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 Lemma 2 we may conclude
\[ P; E \vdash e \]
Thus we have shown \[ P; E \vdash \sigma'(t) \].

- [red let] In this case we have:
\[ \sigma(t) = E[\text{let } x = v \in e] \]
\[ \sigma'(t) = E[e[x := v]] \]
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
\[ P; E \vdash E[\text{let } x = v \in e] : c \cdot a \]
By Lemma 2 we have:
\[ P; E \vdash \text{let } x = v \in c : c_{let} \cdot a_{let} \]
Since this judgment can only be concluded from [exp let] we also have:
\[ P; E \vdash v : c_{o} \cdot AF \]
\[ P; E, c_{o} x \vdash e : c_{let} \cdot a_{e} \]
\[ P; E \vdash a_{e}[x := v] \uparrow a'_{e} \]
\[ a_{let} = (\text{AF}; a_{e}) \]
From Lemma 6 we have:
\[ P; E \vdash e[x := v] : c_{let} \cdot a_{e}[x := v] \]
By Lemma 7 we know:
\[ P; E, c_{o} x \vdash a_{e} \]
This allows us to conclude via Lemma 9 and \[ P; E \vdash v : c_{o} \cdot AF \] that:
\[ P; E \vdash a_{e}[x := v] \uparrow a'_{e} \]
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 \( a_{e}[x := v] \subseteq a_{let} \) and thus by Lemma 4 we have:
\[ P; E \vdash E[e[x := v]] : c \cdot a' \]

- [red if-null] In this case we have
\[ \sigma(t) = E[\text{if null } e_2 e_3] \]
\[ \sigma'(t) = E[e_3] \]
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:
\[ P; E \vdash E[\text{if null } e_2 e_3] : c \cdot a \]
By Lemma 2 we know that:
\[ P; E \vdash e_2 e_3 : c_{if} \cdot a_{if} \]
which can only be concluded by [exp if] so we also have:
\[ P; E \vdash \text{null}; d \cdot AF \]
\[ P; E \vdash e_2 : c_{if} \cdot a_{if} \]
\[ P; E \vdash e_3 : c_{if} \cdot a_{if} \]
\[ a_{if} = (\text{AF}; a_{if}) \]
By Lemma 1 we know that \( a_{if} \subseteq a_{if} \) so by Lemma 4 we can conclude that
\[ P; E \vdash E[e_3] : c \cdot a' \]

- [red if-nonnull] In this case we have
\[ \sigma(t) = E[\text{if } v e_2 e_3] \]
\[ \sigma'(t) = E[e_3] \]
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:
\[ P; E \vdash E[\text{if } v e_2 e_3] : c \cdot a \]
By Lemma 2 we know that:
\[ P; E \vdash e_2 e_3 : c_{if} \cdot a_{if} \]
which can only be concluded by [exp if] so we also have:
\[ P; E \vdash v : d \cdot AF \]
\[ P; E \vdash e_2 : c_{if} \cdot a_{if} \]
\[ P; E \vdash e_3 : c_{if} \cdot a_{if} \]
\[ a_{if} = (\text{AF}; a_{if}) \]
By Lemma 1 we know that \( a_{if} \subseteq a_{if} \) so by Lemma 4 we can conclude that
\[ P; E \vdash E[e_3] : c \cdot a' \]

- [red while] In this case we have:
\[ \sigma(t) = E[\text{while } e_1 e_2] \]
\[ \sigma'(t) = E[\text{if } e_1(e_2; \text{while } e_1 e_2) \text{ null}] \]
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:
\[ P; E \vdash E[\text{while } e_1 e_2] : c \cdot a \]
By Lemma 2 we know that:
\[ P; E \vdash e_1 e_2 : \text{Unit} \cdot a_{w} \]
and from [exp while] we have:
\[ P; E \vdash e_1 : d \cdot a_1 \]
\[ P; E \vdash e_2 : d' \cdot a_2 \]
\[ a_{w} = (a_1; (a_2; a_1)^{e}) \]
By Lemma 10 we know:
\[
P; E \vdash \text{if } e_1(e_2); \text{while } e_1 \ e_2 \text{ null : Unit : } a_w
\]
So by Lemma 4 we can conclude:
\[
P; E \vdash \mathcal{E}[\text{if } e_1(e_2); \text{while } e_1 \ e_2 \text{ null }] : c : a'
\]

- **[RED WRONG]** In this case we have:
\[
\sigma(t) = \mathcal{E}[e] \\
\sigma'(t) = \text{wrong}
\]
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 \( \sigma'(t) = \text{wrong} \) we can conclude \( P; E \vdash \text{wrong} \) 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 \( \text{ls } \vdash \_ \ T \) relates locksets to thread states \( T \): if \( T \) contains an expression \( e \), the judgment holds if \( \text{ls} \) contains all locks \( \rho \) for which \( e \equiv \mathcal{E}[\text{in-sync \ } \rho \ e'] \). We abuse notation slightly to say in-sync \( \in \mathcal{E} \) and in-sync \( \in \mathcal{E} \) if \( e \equiv \mathcal{E}[\text{in-sync \ } \rho \ e'] \) or \( e \equiv \mathcal{E}[\text{in-sync \ } \rho \ e''] \), respectively, for some \( e, e', \rho, \rho' \), and \( \mathcal{E}'' \).

The judgment \( \text{ls } \vdash \_ \ T \) checks the well-formedness of states: the judgment holds if for every thread \( t \), when we obtain the set of locks \( \text{ls} \) held by \( t \) in the state, then for the thread state \( T \) corresponding to \( t \), the judgment \( \text{ls } \vdash \_ \ T \) 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

- **[CS EXP]**
  \[
  \text{ls } \vdash \_ \ T \\
  \begin{align*}
  \text{in-sync} \notin \mathcal{E} & \quad \emptyset \vdash_{cs} e \\
  \text{ls } \cup \{ \rho \} \vdash_{cs} \text{in-sync} \rho e & \quad \text{ls } \vdash_{cs} e \quad \rho \notin \text{ls} \\
  \text{ls not in-sync} & \quad \text{ls } \vdash_{cs} e \quad \text{in-sync} \notin \mathcal{E} \\
  \text{ls } \vdash_{cs} \mathcal{E}[e] & \quad \emptyset \vdash_{cs} e \\
  \text{cs wrong} & \quad \text{ls } \vdash_{cs} \text{wrong}
  \end{align*}
  \]

- **[CS READY]**
  \[
  \emptyset \vdash_{cs} e \\
  \emptyset \vdash_{cs} \text{ready } e \\
  \text{cs wrong}
  \]

- **[CS STATE]**
  \[
  \text{ls } \vdash_{cs} T_k \\
  \text{ls}_k \vdash_{cs} \{ \rho \ | \ \sigma(\rho) = \{ f \in \mathcal{E} \} \} \quad \forall k \in 1..n \\
  \text{ls } \vdash_{cs} P \cup [ \rho_j \mapsto \text{obj} \ j \in 1..m ] \cup [ t_k \mapsto T_k \ k \in 1..n ]
  \]

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 \( \sigma \Rightarrow_1 \sigma' \). Let
\[
\sigma = P \cup [ \rho_j \mapsto \text{obj} \ j \in 1..m ] \cup [ t_k \mapsto T_k \ k \in 1..n ]
\]
From the rule [CS STATE] we have
\[
\text{ls}_k \vdash_{cs} T_k \\
\text{ls}_k \vdash_{cs} \{ \rho \ | \ \sigma(\rho) = \{ f \in \mathcal{E} \} \} \quad \forall k \in 1..n
\]
We need to show
\[
\text{ls}_k \vdash_{cs} T_k \\
\forall k \in 1..n'
\]
for some \( n' \), where
\[
\text{ls}_k \vdash_{cs} \{ \rho \ | \ \sigma(\rho) = \{ f \in \mathcal{E} \} \} \quad \forall k \in 1..n'
\]
in the state
\[
\sigma' = P \cup [ \rho_j \mapsto \text{obj} \ j \in 1..m' ] \cup [ t_k \mapsto T_k \ k \in 1..n' ]
\]
Proof is by case analysis on the reduction rule used for \( \sigma \Rightarrow_1 \sigma' \).

- **[RED READ]** In this case, we have
\[
\sigma(t) = \mathcal{E}[\rho, f] \\
\sigma'(t) = \mathcal{E}[v] \\
\sigma(\rho) = \{ \ldots f = v \ldots \}
\]
All objects and all threads other than \( t \) do not change, hence it is sufficient to show
\[
\text{ls}_k \vdash_{cs} \mathcal{E}[v] \\
\text{ls}_k \vdash_{cs} \mathcal{E}[\rho, f]
\]
By Lemma 21, the derivation for this judgment includes the following judgment
\[
\text{ls}_k \vdash_{cs} \rho, f
\]
where, by rule [CS EXP], we have \( \text{ls}_k = \emptyset \).

Also, we know the following, by rule [CS EXP]:
\[
\text{ls}_v \vdash_{cs} t_v \\
\text{ls}_v = \emptyset
\]
Then by Lemma 22, we may conclude with our desired result
\[
\text{ls}_v \vdash_{cs} \mathcal{E}[v] \\
\text{ls}_v = \text{ls}_k
\]

- **[RED WRITE]** In this case, we have
\[
\sigma(t) = \mathcal{E}[\rho, f = v] \\
\sigma'(t) = \mathcal{E}[v] \\
\sigma(\rho) = \{ \ldots f = v \ldots \}
\]
No object has a lock changed, and all threads other than \( t \) do not change, hence it is sufficient to show
\[
\text{ls}_k \vdash_{cs} \mathcal{E}[v] \\
\text{ls}_k \vdash_{cs} \mathcal{E}[\rho, f = v]
\]
By Lemma 21, the derivation for this judgment includes the following judgment
\[
\text{ls}_w \vdash_{cs} \rho, f = v
\]
where, by rule [cs exp], we have \( ls_\omega = \emptyset \).

Also, we know the following, by rule [cs exp]:

\[
l_{s,v} \vdash_{cs} v \\
l_{s,v} = \emptyset
\]

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

\[
l_{s'} \vdash_{cs} E[e] \\
l_{s'} = ls
\]

**[red invoke]**. In this case, let \( \theta = [\text{this} := \rho, x_i := v_{i \in 1..n}] \),

where

\[
\sigma(t) = \mathcal{E}[\rho, m(v_{1..n})] \\
\sigma'(t) = \mathcal{E}[\theta(e)]
\]

\( \rho \in \text{Addr}_e \)

\( \text{class } c \{ \ldots a_m \, d_m \, m(d_i, x_i \, i \in 1..n) \{ \, e \} \ldots \} \in \mathcal{P} \)

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

\[
l_s \vdash_{cs} \mathcal{E}[^{\theta(e)}]
\]

where \( ls = \{ \rho \mid \sigma(\rho) = \{ \mathcal{T} = \mathcal{E}[\mathcal{P}] \} \} \).

We have, by assumption,

\[
l_{s,v} \vdash_{cs} \mathcal{E}[\rho, m(v_{1..n})]
\]

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

\[
l_{s,v} \vdash_{cs} \rho, m(v_{1..n})
\]

where, by rule [cs exp], we have \( ls_{s,v} = \emptyset \).

Also, the method body \( e \) cannot contain an \( \text{in-sync} \) run-time expression (due to well-formedness), and the \( \theta \) substitution does not replace any part of \( e \) with an \( \text{in-sync} \) expression.

Hence we know the following, by rule [cs exp]:

\[
l_{s,v} \vdash_{cs} \theta(e) \\
l_{s,v} = \emptyset
\]

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

\[
l_{s'} \vdash_{cs} \mathcal{E}[^\theta(e)] \\
l_{s'} = ls
\]

**[red invoke compound]**. Similar to previous argument.

**[red new]**. In this case, we have

\[
\sigma(t) = \mathcal{E}[\text{new } c(v_{1..n})] \\
\sigma'(t) = \mathcal{E}[p] \\
\sigma'(\rho) = \{ f_i = v_{i \in 1..n} \}^+ \\
\rho \notin \text{Dom}(\sigma) \\
\rho \in \text{Addr}_e \\
\text{class } c \{ \ldots d_i \, f_i \, \{ e \} \ldots \} \in \mathcal{P}
\]

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

\[
l_s \vdash_{cs} \mathcal{E}[p]
\]

where \( ls = \{ \rho \mid \sigma(\rho) = \{ \mathcal{T} = \mathcal{E}[\mathcal{P}] \} \} \).

We have, by assumption,

\[
l_s \vdash_{cs} \mathcal{E}[\text{new } c(v_{1..n})]
\]

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

\[
l_{s,v} \vdash_{cs} \text{new } c(v_{1..n})
\]

where, by rule [cs exp], we have \( ls_{s,v} = \emptyset \).

Also, we know the following, by rule [cs exp]:

\[
l_{s,v} \vdash_{cs} \rho \\
l_{s,v} = \emptyset
\]

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

\[
l_{s'} \vdash_{cs} \mathcal{E}[\rho] \\
l_{s'} = ls
\]

**[red sync]**. In this case we have:

\[
\sigma(t) = \mathcal{E}[\rho, \text{sync } e] \\
\sigma(\rho) = \{ \mathcal{T} = \mathcal{E}[\mathcal{P}] \}^+ \\
\sigma'(t) = \mathcal{E}[\text{in-sync } \rho, e] \\
\sigma'(\rho) = \{ \mathcal{T} = \mathcal{E}[\mathcal{P}] \}^+
\]

The object at \( \rho \) 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

\[
l_s \cup \{ \rho \} \vdash_{cs} \mathcal{E}[\text{in-sync } \rho, e]
\]

where \( ls = \{ \rho \mid \sigma(\rho) = \{ \mathcal{T} = \mathcal{E}[\mathcal{P}] \} \} \).

We have, by assumption,

\[
l_s \vdash_{cs} \mathcal{E}[\rho, \text{sync } e]
\]

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

\[
l_{s,v} \vdash_{cs} \rho, \text{sync } e
\]

where, necessarily, we have \( ls_{s,v} = \emptyset \) by rule [cs exp]; by inspection of the reduction rules, \( e \) cannot contain an \( \text{in-sync} \) expression.

By rule [cs in-sync], we have

\[
\{ \rho \} \vdash_{cs} \text{in-sync } \rho, e
\]

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

\[
l_{s'} \vdash_{cs} \mathcal{E}[\rho, \text{sync } e] \\
l_{s'} = ls \cup \{ \rho \}
\]

**[red sync reentrant]**. In this case we have

\[
\sigma(t) = \mathcal{E}[\rho, \text{sync } e] \\
\sigma(\rho) = \{ \mathcal{T} = \mathcal{E}[\mathcal{P}] \}^+ \\
\sigma'(t) = \mathcal{E}[\text{in-sync } \rho, e] \\
\sigma'(\rho) = \{ \mathcal{T} = \mathcal{E}[\mathcal{P}] \}^+
\]

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

\[
l_s \vdash_{cs} \mathcal{E}[e]
\]

where \( ls = \{ \rho \mid \sigma(\rho) = \{ \mathcal{T} = \mathcal{E}[\mathcal{P}] \} \} \).

We have, by assumption,

\[
l_s \vdash_{cs} \mathcal{E}[\rho, \text{sync } e]
\]

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

\[
l_{s,v} \vdash_{cs} \rho, \text{sync } e
\]

where, necessarily, we have \( ls_{s,v} = \emptyset \) by rule [cs exp]; by inspection of the reduction rules, \( e \) cannot contain an \( \text{in-sync} \) expression.

Again by rule [cs exp], we have

\[
l_{s,e} \vdash_{cs} \mathcal{E}[e]
\]

where \( ls_{s,e} = \emptyset \).

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

\[
l_{s'} \vdash_{cs} \mathcal{E}[e] \\
l_{s'} = ls
\]
• [red in-sync]. In this case we have
\[\sigma(t) = E[\text{in-sync } \rho \ v]\]
\[\sigma'(t) = E[v]\]
\[\sigma'(\rho) = \{T = \tau\}^\flat\]

The object at \(\rho\) 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
\[ls \setminus \{\rho\} \vdash_{cs} E[e]\]

where \(ls = \{\rho | \sigma(\rho) = \{T = \tau\}^\flat\}\).

We have, by assumption,
\[ls \vdash_{cs} E[\text{in-sync } \rho \ v]\]

By Lemma 21, the derivation for this judgment includes the following judgment
\[ls_s \vdash_{cs} \text{in-sync } \rho \ v\]

where, by rule [cs in-sync], we have \(ls_s = \{\rho\}\).

Also, we know the following, by rule [cs exp]:
\[ls_v \vdash_{cs} v\]
\[ls_v = \emptyset\]

Then by Lemma 22, we may conclude with our desired result
\[ls' \vdash_{cs} E[v]\]
\[ls' = ls \setminus \{\rho\}\]

• [red fork]. In this case we have:
\[\sigma(t) = E[\text{fork } e]\]
\[\sigma'(t) = E[\text{null}]\]
\[\sigma'(\rho) = \text{ready } e\]

where \(t' \not\in \text{Dom}(\sigma)\).

Here, a new thread \(t'\) is introduced into the state \(\sigma'\). All objects and all threads other than \(t\) and \(t'\) do not change; hence it is sufficient to show
\[ls \vdash_{cs} E[\text{null}]\]
\[\emptyset \vdash_{cs} \text{ready } e\]

where \(ls = \{\rho | \sigma(\rho) = \{T = \tau\}^\flat\}\).

We have, by assumption,
\[ls \vdash_{cs} E[\text{fork } e]\]

By Lemma 21, the derivation for this judgment includes the following judgment
\[ls_f \vdash_{cs} \text{fork } e\]

where, necessarily, we have \(ls_f = \emptyset\) 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]:
\[ls_v \vdash_{cs} \text{null}\]
\[ls_v = \emptyset\]

Then by Lemma 22, and by rule [cs ready], we may conclude with our desired result
\[ls' \vdash_{cs} E[\text{null}]\]
\[ls' = ls\]
\[\emptyset \vdash_{cs} \text{ready } e\]

• [red ready]. In this case we have
\[\sigma(t) = \text{ready } e\]
\[\sigma'(t) = [e]\]

All objects and all threads other than \(t\) do not change; hence it is sufficient to show
\[ls \vdash_{cs} [e]\]

where \(ls = \{\rho | \sigma(\rho) = \{T = \tau\}^\flat\}\).

By inspection of the reduction rules, we observe that no lock is held by \(t\) in \(\sigma\): 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,
\[\emptyset \vdash_{cs} \text{ready } e\]

This judgment may be derived only through rule [cs ready]; hence we obtain our desired result:
\[ls \vdash_{cs} \text{ready } e\]

• [red let]. In this case we have:
\[\sigma(t) = E[\text{let } x = v \ \text{in } e]\]
\[\sigma'(t) = E[e[x := v]]\]

All objects and all threads other than \(t\) do not change; hence it is sufficient to show
\[ls \vdash_{cs} E[e[x := v]]\]

where \(ls = \{\rho | \sigma(\rho) = \{T = \tau\}^\flat\}\).

We have, by assumption,
\[ls \vdash_{cs} E[\text{let } x = v \ \text{in } e]\]

By Lemma 21, the derivation for this judgment includes the following judgment
\[ls_f \vdash_{cs} \text{let } x = v \ \text{in } e\]

where, necessarily, we have \(ls_f = \emptyset\) 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
\[ls' \vdash_{cs} E[e[x := v]]\]
\[ls' = ls\]

• [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
\[\sigma(t) = E[e]\]
\[\sigma'(t) = \text{wrong}\]

All objects and all threads other than \(t\) do not change; hence it is sufficient to show
\[ls \vdash_{cs} \text{wrong}\]

where \(ls = \{\rho | \sigma(\rho) = \{T = \tau\}^\flat\}\).

Immediately by rule [cs wrong], we obtain our desired result:
\[ls \vdash_{cs} \text{wrong}\]

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 $\sigma$ under $\rightarrow$, then it can reach $\sigma$ under $\rightarrow_e$ 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 $S \subseteq \text{State}$ and transition relation $T \subseteq \text{State} \times \text{State}$, the left restriction of $T$ to $S$, or $T[S/T]$, is the transition relation $T$ with each pair’s first component contained in $S$. Similarly, the right restriction of $T$ to $S$, or $T\setminus S$, 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:

- $N_i$ is the set of states where the thread $i$ is not in any transaction;
- $R_i$ is the set of states where the thread $i$ is in the right-mover part of some transaction;
- $L_i$ is the set of states where the thread $i$ is in the left-mover part of some transaction; and
- $W_i$ is the set of states where the thread $i$ has gone wrong.

The reduction theorem relates the following three transition relations:

- $\rightarrow_i$ is the transition relation that describes the behavior of the thread $i$;
- $\leftarrow_i$ is the transition relation where, at each state, one may choose a thread $i$ and use the transition $\rightarrow_i$;
- $\leftarrow_e$ 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 $R_i$, $L_i$, and $W_i$ be sets of states, and $\rightarrow_i$ be a transition relation. Suppose for all $i$:

1. $R_i$, $L_i$, and $W_i$ are pairwise disjoint,
2. $(L_i/\rightarrow_i \setminus R_i)$ is false,
3. $W_i/\rightarrow_i$ is false,

and for all $j \neq i$:

4. $\rightarrow_i$ and $\rightarrow_j$ are disjoint,
5. $(\rightarrow_i \setminus R_i)$ right-commutes with $\rightarrow_j$,
6. $(L_i/\rightarrow_i)$ left-commutes with $\rightarrow_j$, and
7. if $p \rightarrow_i q$, then $R_i(p) \Rightarrow R_j(q)$, $L_i(p) \Rightarrow L_j(q)$, and $W_i(p) \Rightarrow W_j(q)$.

We define $N_i$, $N$, $W$, $\leftarrow_i$, and $\leftarrow_e$ as follows:

- $N_i = \neg(R_i \lor L_i)$
- $N = \forall i . N_i$
- $W = \exists i . W_i$
- $\leftarrow_i = \exists i . \leftarrow_i$
- $\leftarrow_e = \exists \sigma . (\forall j \neq i . N_j)/\leftarrow_i$

Now suppose $p \in N$ and $p \leftarrow_e q$. Then the following statements are true.

1. If $q \in N$, then $p \leftarrow_e^* q$.
2. If $q \in W$ and $\forall i . q \notin L_i$, then $p \leftarrow_e^* q'$ and $q' \in W$.

**Proof** Refer to [23].

We now turn our attention to the specific case of YieldJava. Consider a fixed program $P$. Any well-typed state $\sigma$ with an object map $[p_i \rightarrow \text{obj}_j \in \text{CL}]$ has a corresponding object environment $E_\sigma = c_1, p_1, \ldots, c_m, p_m$ where $p_i \in \text{Addr}_c$.

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

\[
\alpha(\sigma, e) = a
\]

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:

$N_i = WT \cap \{ \sigma \mid \forall i . \sigma(i) \neq \text{wrong} \}$

$W_i = WT \cap \{ \sigma \mid \sigma(i) \equiv \text{wrong} \}$

$R_i = WT \cap \{ \sigma \mid \alpha(\sigma, \sigma(i)) \subseteq \text{CL} \} \setminus N_i$

$L_i = WT \cap \{ \sigma \mid \alpha(\sigma, \sigma(i)) \subseteq \text{CL} \} \setminus N_i$

Then we define the auxiliary states $N$ and $W$:

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

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

$Y(\xi, \kappa) = \kappa$

$Y(\xi, \rho \cdot a_1 : a_2) = Y(\xi, a_1)$ if $\rho \in \text{locks}(\xi)$

$Y(\xi, \rho \cdot a_1 : a_2) = Y(\xi, a_1) : Y(\xi, a_2)$ otherwise

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

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

1. If $\sigma' \in N$, then $\sigma \rightarrow_e^* \sigma'$.
2. If $\sigma' \in W$ and $\forall i . \sigma' \notin L_i$, then there exists a state $\sigma''$ such that $\sigma \rightarrow_e^* \sigma''$ 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. $L_i/\rightarrow_i \setminus R_i$ is false,
3. $W_i/\rightarrow_i$ is false,
4. $\rightarrow_i$ and $\rightarrow_j$ are disjoint for all $j \neq i$,
5. $(\rightarrow_i \setminus R_i)$ 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 $R_i(p) \Rightarrow R_j(q)$, $L_i(p) \Rightarrow L_j(q)$, and $W_i(p) \Rightarrow W_j(q)$ for all $j \neq i$.

We apply the Reduction Theorem by substituting the following:
the set $W_i$ for $W_i$, the set $R_i$ for $R_i$, the set $L_i$ for $L_i$, the relation $→_1$ for $←_1$, the relation $→_2$ for $←_2$, the state $σ$ for $p$, and the state $σ'$ for $q$.

1. $R_i, L_i, \text{and } W_i$ are pairwise disjoint.

**Proof:** Inspection of definitions.

2. $L_i / →_1 \setminus R_i$ is false.

**Proof:** Case analysis on $σ → σ'$.

(a) **Cases** [red new], [red read], [red write], [red fork], and [red in-sync]: We assume that $σ'(i) \in R_i$ and thus $α(σ', σ'(i)) ⊑ CL$. For some well-formed state $σ_0$,

- **[red new]:** We have
  
  $σ = σ_0[i → E[\text{new } c(v)]]$  
  
  $σ' = α(σ_0[i → E[ρ, f]]$  

  where $ρ \not∈ \text{Dom}(σ)$ and $ρ \in \text{Addr}_c$, and $class \ c ∈ \{d \vec{F} \ldots \} ∈ P$.

- **[red read]:** We have
  
  $σ = σ_0[i → E[ρ, f]]$  
  
  $σ' = σ_0[i → E[v]]$  

  where $σ(ρ, f) = v$.

- **[red write]:** We have
  
  $σ = σ_0[i → E[ρ, f]]$  
  
  $σ' = σ_0[i → E[v]]$  

  $σ' = σ_0[i → E[null], i' → ready e]$  

  where $i' \not∈ \text{Dom}(σ)$.

- **[red in-sync]:** We have
  
  $σ = σ_0[i → E[\text{in-sync } v], ρ → \{ \vec{F} = τ \}]$  
  
  $σ' = σ_0[i → E[null], i′ → ready e]$  

  For cases [red read] and [red write], $γ = ✓$, since $σ ∈ L_i$, which excludes any yield by thread $i$. Let $σ(i) = E[e]$ for each case, where $e$ is some red $c$, and $σ'(i) = E[v]$. Then we have $α(σ, E[e]) ⊑ CL$ and $α(σ', E[v]) ⊑ CL$. Since $α(σ', v) ⊑ α(σ, e)$, we obtain, by Lemma 4,

  $α(σ', E[e]) ⊑ α(σ, E[e]) ⊑ CL$  

  which contradicts our assumption of $σ'(i) ∈ R_i$.

(b) **[red invoke]:** For some well-formed state $σ_0$, we have

$σ = σ_0[i → E[ρ, m(τ)]]$  

$σ' = σ_0[i → E[c \text{this} := ρ, τ := τ]]$  

where $ρ \in \text{Addr}_d$, and $class \ d′ ⇓ a \ m(\vec{A}) \{ e \} \ldots \} ∈ P$.

Here, $γ = ✓$, since $L_i$ excludes yielding thread states for $i$. We show that

$α(σ', E[c \text{this} := ρ, τ := τ]) ⊑ α(σ, ρ, m(τ))$  

and use Lemma 4 to obtain a contradiction. From $α(σ, ρ, m(τ))$ being well-defined, we know the following judgment must hold for the well-formed environment $E_0$:

$P; E_0 ∨ ρ, m(τ) : c \cdot a$  

and from the antecedents of this judgment we know that

$P; E_0 \vdash a' \text{[this} := ρ, τ := τ] \uparrow a$  

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

$a' \text{[this} := ρ, τ := τ] ⊑ a$  

(1)

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

$P; c \text{this} ≡ a' \ c m(\vec{A}) \{ e \} \ldots \} ∈ P$  

and by inversion on this judgment, we know

$P; c \text{this} = \vec{F} \uparrow e : c \cdot a''$  

such that

$a'' ⊑ a'$  

(2)

By the substitution lemma, we have

$P; c \text{this} ≡ (a'' \text{[this} := ρ, τ := τ] \uparrow a$  

Thus we have the equality

$α(σ', c \text{this} := ρ, τ := τ) = a'' \text{[this} := ρ, τ := τ]\uparrow a$  

By Lemma 1 and Formula 2,

$a'' \text{[this} := ρ, τ := τ] ∈ a'$  

(3)

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

$a'' \text{[this} := ρ, τ := τ] ⊑ a$  

(4)

Applying Lemma 4, we get

$α(σ', E[e \text{this} := ρ, τ := τ]) ⊑ α(σ, E[ρ, m(τ)])$  

(5)

We have obtained a contradiction: by assumption of $σ' ∈ R_i$,

$α(σ', E[e \text{this} := ρ, τ := τ]) ⊑ CL$  

but by Formulas 4 and 5, we have

$α(σ', E[e \text{this} := ρ, τ := τ]) ⊑ CL$  

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

$σ = σ_0[i → E[ρ, sync c]], ρ → \{ \vec{F} = τ \}]$  

$σ' = σ_0[i → E[\text{in-sync } ρ, ρ, e, τ := τ]]$  

We have $γ = ✓$, since $σ ∈ L_i$, excludes yielding thread states for $i$. By $S$ and $SI$, we have

$α(σ, \text{sync } e) = SI(ρ, α(σ, e))$  

$α(σ', \text{in-sync } ρ, e) = SI(ρ, α(σ, e))$  

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

$α(σ', \text{in-sync } ρ, e) ⊑ α(σ, \text{sync } e)$  

We use Lemma 5 to obtain the subeffect relation

$α(σ', E[\text{in-sync } ρ, e]) \subseteq α(σ, E[\text{sync } e])$  

But this is a contradiction: By assumption of $σ ∈ L_i$, we know

$α(σ, E[\text{sync } e]) ⊑ CL$  

and so

$α(σ', E[\text{in-sync } ρ, e]) \subseteq (\rho) CL$  

23
But also \( \sigma' \in R_i \) by assumption, and we have
\[
\alpha(\sigma', \mathcal{E}[\text{in-sync } \rho \varepsilon]) \not\subseteq \mathcal{CL}
\]

Hence this case is trivially true.

(e) [Red Sync Reentrant]: For some well-formed state \( \sigma_0 \), we have
\[
\begin{align*}
\sigma &= \sigma_0[i \mapsto \mathcal{E}[\rho, \text{sync } e], \rho \mapsto \{ \tau = \top \}] \\
\sigma' &= \sigma_0[i \mapsto \mathcal{E}[e], \rho \mapsto \{ \tau = \top \}]
\end{align*}
\]
We have \( \gamma = \) since \( \gamma \in L_i \) and thus non-yielding. Thread \( i \) may transition from \( \mathcal{E}[\rho, \text{sync } e] \) to \( \mathcal{E}[e] \) only if \( \rho \) is held by \( i \). Hence we have \( \rho \in \text{locks}(\mathcal{E}) \). By Lemma 15, we have
\[
\alpha(\sigma', e) \subseteq (\rho, \alpha(\sigma, e))
\]
This implies
\[
\mathcal{AF}; \alpha(\sigma', e) \subseteq \emptyset \cup \text{locks}(\mathcal{E}) \subseteq (\rho, \alpha(\sigma, e))
\]
Applying Lemma 5, we obtain
\[
\mathcal{AF}; \alpha(\sigma', \mathcal{E}[e]) \subseteq \emptyset \cup \alpha(\sigma, \mathcal{E}[\rho, \text{sync } e])
\]
which is equivalent to
\[
\alpha(\sigma', \mathcal{E}[e]) \subseteq \alpha(\sigma, \mathcal{E}[\rho, \text{sync } e])
\]
This is a contradiction: By assumption of \( \sigma \in L_i \), we know
\[
\alpha(\sigma, \mathcal{E}[\rho, \text{sync } e]) \subseteq \mathcal{CL}
\]
and so
\[
\alpha(\sigma', \mathcal{E}[e]) \not\subseteq \mathcal{CL}
\]
But also \( \sigma' \in R_i \) by assumption, and we have
\[
\alpha(\sigma', \mathcal{E}[e]) \not\subseteq \mathcal{CL}
\]
Hence this case is trivially true.

(f) [Red Ready]: \( \sigma(i) = \mathcal{E}[\text{ready } e] \) and is yielding, thus we ignore this case.

(g) [Red Let]: For some well-formed state \( \sigma_0 \), we have
\[
\begin{align*}
\sigma &= \sigma_0[i \mapsto \mathcal{E}[\text{let } x = v \text{ in } e]] \\
\sigma' &= \sigma_0[i \mapsto \mathcal{E}[e[x := v]]]
\end{align*}
\]
We show that
\[
\alpha(\sigma', e[x := v]) \subseteq \alpha(\sigma, \text{let } x = v \text{ in } e)
\]
and use Lemma 4 to obtain a contradiction. Because \( \alpha(\sigma, \text{let } x = v \text{ in } e) \) is well-defined, we know the following judgment must hold for the well-formed environment \( E_o \):
\[
P; E_o \vdash \text{let } x = v \text{ in } e : c \cdot a
\]
and the following antecedents in this judgment must also be true:
\[
P; E_o \vdash v : \mathcal{AF} \cdot c' \\
P; E_o, c' \vdash x : c \cdot a' \\
P; E_o \vdash a'[x := v] : \top \cdot a
\]
Applying the substitution lemma on the second antecedent, we get
\[
P; E_o \vdash e[x := v] : c \cdot a'[x := v]
\]
By definition of \( \top \), we have
\[
a'[x := v] \subseteq a
\]
(h) [Red If-Nonnull] and [Red If-Null]: For some well-formed state \( \sigma_0 \), we have
\[
\begin{align*}
\sigma &= \sigma_0[i \mapsto \mathcal{E}[\text{if } v \ v e_2 e_3]] \\
\sigma' &= \sigma_0[i \mapsto \mathcal{E}[e_j]] \quad \text{for } j = 2..3
\end{align*}
\]
and by assumption, \( \sigma \in L_i \) and \( \sigma' \in R_i \). Then by definition of \( \sqcup \), for \( j = 2..3, \)
\[
\alpha(\sigma', e_j) \subseteq \alpha(\sigma, \text{if } v e_2 e_3) = (\alpha(\sigma, e_2) \sqcup \alpha(\sigma, e_3))
\]
By Lemma 4, we obtain a contradiction for \( \sigma'(i) \).

(i) [Red Wrong]: Since \( \sigma'(i) = \text{wrong} \), we have \( \sigma' \not\subseteq R_i \).

3. \( \mathcal{W}_i / \rightarrow_i \) is false.
Proof: By definition of \( \mathcal{W}_i = \mathcal{W} \cap \{ \sigma \mid \sigma(i) \equiv \top \} \) 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. \( (\rightarrow_i \setminus R_i) \) 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 \( p_2 \rightarrow_j \). In the arguments that follow, we assume a well-formed state \( \sigma \).

- [Red Read]: Let \( p_1 = \sigma[i \mapsto \mathcal{E}[\rho, f]] \) and \( p_2 = \sigma[i \mapsto \mathcal{E}[v]] \), where \( \rho(f, v) = \sigma \). We proceed by case analysis on \( \gamma \) and field type.
  (a) \( \gamma = \) and \( f \in \text{Normal} \): In this case, no other thread reads or writes to \( \rho(f) \). Thus \( \rightarrow_i \setminus R_i \) and \( \rightarrow_j \) commute, since they operate on disjoint data.
  (b) \( \gamma = \) and \( f \in \text{Final} \): In this case, other threads may only read \( \rho(f) \). Thus \( \rightarrow_i \setminus R_i \) and \( \rightarrow_j \) commute, since reads of \( \rho(f) \) by threads \( i \) and \( j \) commute.
  (c) \( \gamma = \) and \( f \in \text{Volatile} \): By Lemma 3, we have
\[
Y(\mathcal{E}, \sigma(p_1, \rho, f)) \subseteq \alpha(p_1, \mathcal{E}[\rho, f])
\]
This is a contradiction: we have
\[
Y(\mathcal{E}, \sigma(p_1, \rho, f)) = \alpha(p_1, \rho, f) = \mathcal{AN}
\]
and \( \mathcal{AN} \) sequentially composed with \( \alpha(p_1, \mathcal{E}[v]) = \alpha(p_2, \mathcal{E}[v]) \) \( \subseteq \mathcal{CL} \) is undefined.
  (d) \( \gamma = \) and \( f \in \text{Volatile} \): By Lemma 3, we have
\[
Y(\mathcal{E}, \sigma(p_1, \rho, f)) \subseteq \alpha(p_1, \mathcal{E}[\rho, f])
\]
By rule [Exp Ref Race], we have
\[
Y(\mathcal{E}, \sigma(p_1, \rho, f)) \subseteq \alpha(p_1, \mathcal{E}[\rho, f]) \subseteq \alpha(p_2, \mathcal{E}[v]) \subseteq \mathcal{CL}
\]
while by assumption we have
\[
\alpha(p_1, \mathcal{E}[v]) = \alpha(p_2, \mathcal{E}[v]) \not\subseteq \mathcal{CL}
\]
This is a contradiction, since this sequential composition is undefined.

- [Red Write]: Let \( p_1 = \sigma[i \mapsto \mathcal{E}[\rho, f = v]] \) and \( p_2 = \sigma[i \mapsto \mathcal{E}[v]] \). We proceed by case analysis on \( \gamma \) and field type.
  (a) \( \gamma = \) and \( f \in \text{Normal} \): In this case, the threads \( i \) and \( j \) operate on disjoint data – hence the operations commute.
  (b) \( \gamma = \) and \( f \in \text{Volatile} \): By Lemma 3, we have
\[
Y(\mathcal{E}, \sigma(p_1, \rho, f = v)) \subseteq \alpha(p_1, \mathcal{E}[\rho, f = v])
\]
This is a contradiction, since
\[
Y(\mathcal{E}, \sigma(p_1, \rho, f = v)) = \alpha(p_1, \rho, f = v) = \mathcal{AN}
\]
and \( \mathcal{AN} \) sequentially composed with \( \alpha(p_1, \mathcal{E}[v]) = \alpha(p_1, \mathcal{E}[v]) \) \( \subseteq \mathcal{CL} \) is undefined.
(c) \( \gamma = \ldots \) and \( f \in \text{Volatile} \): By Lemma 3, we have
\[
Y(E, \alpha(p_1, \rho.f = v)) = \alpha(p_1, E[v])
\]
By rule \( \text{EXP ASSIGN VOLATILE} \), we have
\[
Y(E, \alpha(p_1, \rho.f = v)) = \alpha(p_1, E[v]) = \text{CL}
\]
while by assumption we have
\[
\alpha(p_1, E[v]) = \alpha(p_2, E[v]) \notin \text{CL}
\]
This is a contradiction, since this sequential composition is undefined.

- \( \text{RED NEW} \): Suppose the step \( \to_i \), creates a new object at
  address \( \rho \). Then observe that the step \( \to_i \) cannot access \( \rho \),
  since thread \( j \) must be well-typed in an environment that does not
  contain \( \rho \). Thus, \( \to_i \setminus \{i\} \) changes only thread \( i \)
  and \( \rho \), and \( \to_i \) changes only the thread \( j \) and maybe some
  other address \( \rho' \neq \rho \). So \( \to_i \setminus \{i\} \) and \( \to_j \) commute.

- \( \text{RED SYNC} \): Let \( p_1 = \sigma[i \to E[\rho.\text{sync } e]] \) and \( p_2 = \sigma[i \to E[\text{in-sync } \rho v]] \).
  To interfere with \( \to_i \), the step \( \to_j \) would need to obtain the lock \( \rho \).
  By inspection, observe that only rules \( \text{RED SYNC} \), \( \text{RED REENTRY} \),
  and \( \text{RED IN-SYNC} \) hold \( \rho \), all of which cannot occur.

- \( \text{RED REENTRY} \): Similar to previous case.

- \( \text{RED IN-SYNC} \): Let \( p_1 = \sigma[i \to E[\text{in-sync } \rho v]] \) and \( p_2 = \sigma[i \to E[v]] \).
  By Lemma 3, we have
\[
Y(E, \alpha(p_1, \text{in-sync } \rho v)) = \alpha(p_1, E[v])
\]
The function \( \mathcal{S} \) tells us that
\[
Y(E, \alpha(p_1, \text{in-sync } \rho v)) = \alpha(p_1, \text{in-sync } \rho v)
\]
However, \( \alpha(p_1, E[v]) = \alpha(p_2, E[v]) \notin \text{CL} \). Hence the
sequential composition is undefined, a contradiction.

- \( \text{RED FORK} \): Let \( p_1 = \sigma[i \to E[\text{fork } e]] \) and \( p_2 = \sigma[i \to E[\text{null}]] \).
  By Lemma 3, we have
\[
Y(E, \alpha(p_1, \text{fork } e)) = \alpha(p_1, \text{fork } e)
\]
The rule \( \text{EXP FORK} \) tells us that
\[
Y(E, \alpha(p_1, \text{fork } e)) = \alpha(p_1, \text{fork } e) = \text{AL}
\]
However, \( \alpha(p_1, \text{null}) = \alpha(p_2, \text{null}) \notin \text{CL} \). Hence the
sequential composition is undefined, a contradiction.

- \( \text{RED INVOKE} \), \( \text{RED INVoke COMPOUND} \), \( \text{RED LET} \), \( \text{RED IF-NONNULL} \),
  \( \text{RED IF-NULL} \), \( \text{RED WHILE} \), \( \text{RED READY} \), \( \text{RED WRONG} \):
  Since the operation by thread \( i \) is entirely local, it commutes
  with the operation of any other thread \( j \).

6. \( (L_i / \to_j) \) 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 \( \sigma \).

- \( \text{RED READ} \): Let \( p_2 = \sigma[i \to E[\rho.f]] \) and \( p_3 = \sigma[i \to E[v]] \).
  We proceed by case analysis on \( \gamma \) and field type.
  (a) \( \gamma = \ldots \) and \( f \in \text{Normal} \): In this case, field \( f \) is synchro-
  nized, and thread \( j \) may not read or write to \( \rho.f \). Since the
  two operations access disjoint data, they commute.
  (b) \( \gamma = \ldots \) and \( f \in \text{Final} \): In this case, thread \( j \) may read
  from (but not write to) \( \rho.f \). Since two reads commute freely, the
  two operations commute.
  (c) \( \gamma = \ldots \) and \( f \in \text{Volatile} \): By the Lemma 13, we have
  \( \alpha(p_2, E[v]) = \alpha(p_3, E[v]) \)
  By Lemma 3, we have
\[
Y(E, \alpha(p_2, \rho.f)) = \alpha(p_2, E[v]) \subseteq \alpha(p_2, E[\rho.f])
\]
where \( \sigma(p.f) = v \). By rule \( \text{EXP REF RACE} \), we have
\[
Y(E, \alpha(p_2, \rho.f)) = \alpha(p_2, \rho.f) = \text{AN}
\]
Since sequential composition of \( N \) with any mover effect
is either undefined or \( \subseteq \text{L} \), this contradicts our assumption
of \( p_2 \in L_i \).

- \( \text{RED WRITE} \): Let \( p_2 = \sigma[i \to E[\rho.f = v]] \) and \( p_3 = \sigma[i \to E[v]] \).
  We proceed by case analysis on \( \gamma \) and field type.
  (a) \( \gamma = \ldots \) and \( f \in \text{Normal} \): In this case, field \( f \) is synchro-
  nized, and thread \( j \) may not read or write to \( \rho.f \). Since the
  two operations access disjoint data, they commute.
  (b) \( \gamma = \ldots \) and \( f \in \text{Volatile} \): By Lemma 13, we have
  \( \alpha(p_2, E[v]) = \alpha(p_3, E[v]) \)
  By Lemma 3, we have
\[
Y(E, \alpha(p_2, \rho.f)) = \alpha(p_2, E[v]) \subseteq \alpha(p_2, E[\rho.f])
\]
Since sequential composition of \( N \) with any mover effect
is either undefined or \( \subseteq \text{L} \), this contradicts our assumption
of \( p_2 \in L_i \).

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

- \( \text{RED IN-SYNC} \): Let \( p_2 = \sigma[i \to E[\text{in-sync } e]] \) and \( p_3 = \sigma[i \to E[v]] \).
  We have \( \gamma = \ldots \), \( p_2 \in N_i \), and also \( p_2 \in L_i \): a contradiction.
[RED IN-SYNC], and any other operation by \( j \) successfully commutes with \( \rightarrow_{j} \).

- [RED SYNC REENTRANT]: Let \( p_{2} = \sigma[i \mapsto E[\rho_{2}, \text{sync } e], \rho \mapsto \{ \gamma = \tau \}] \) and \( p_{3} = \sigma[i \mapsto E[e], \rho \mapsto \{ \gamma = \tau \}] \). The argument proceeds similar to the previous case.

- [RED FORK]: Let \( p_{2} = \sigma[i \mapsto E[\text{fork } e]] \) and \( p_{3} = \sigma[i \mapsto E[\text{null}], i' \mapsto \text{ready } e] \), where \( i' \notin \text{Dom}(p_{3}) \). 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 \( p_{2} = \sigma[i \mapsto \text{ready } e] \), we have \( p_{3} \in N_{\theta} \): a contradiction.

- [RED INVOKE], [RED INVOKE COMPOUND], [RED LET], [RED IF-NULL], [RED WHILE], [RED WRONG]: If there is a \( \gamma \) in any of these operations, we have \( \gamma = \cdot \) since \( p_{2} \) is non-nesting in \( L_{\alpha} \). For these operations, all modifications to state are local, and cannot be affected by \( \rightarrow_{j} \). Thus \( \rightarrow_{j} \) and \( \rightarrow_{k} \) commute.

7. If \( p \rightarrow_{i} q \), then \( R_{j}(p) \Rightarrow R_{j}(q), L_{j}(p) \Rightarrow L_{j}(q), \) and \( W_{j}(p) \Rightarrow W_{j}(q) \), 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 \( R_{j}(p) \Rightarrow R_{j}(q) \) and \( L_{j}(p) \Rightarrow L_{j}(q) \) and \( W_{j}(p) \Rightarrow W_{j}(q) \).

Assume \( i \) forks a new thread \( k \). Since only these two changes have, for all threads \( j \) other than \( k \) and \( j \), we have \( R_{j}(p) \Rightarrow R_{j}(q) \) and \( L_{j}(p) \Rightarrow L_{j}(q) \) and \( W_{j}(p) \Rightarrow W_{j}(q) \). For thread \( k \), we have \( N_{\alpha}(p) \) since \( k \notin \text{Dom}(p) \), and \( N_{\alpha}(q) \), since the thread state at \( k \) is yielding in \( q \). This implies \( R_{k}(p) \Rightarrow R_{k}(q) \) and \( L_{k}(p) \Rightarrow L_{k}(q) \) and \( W_{k}(p) \Rightarrow W_{k}(q) \).

**Lemma 1** (Effect Monotonicity).

1. If \( a_{1} \sqsubseteq a_{2} \), then for all \( a: \)

\[
\begin{align*}
a_{1}; & \sqsubseteq a_{2}; a \\
\alpha; a_{1} & \sqsubseteq \alpha; a_{2} \\
a_{1} & \sqsubseteq \alpha; a_{2} \\
\theta(a_{1}) & \sqsubseteq \theta(a_{2})
\end{align*}
\]

where \( \theta \) is a set of substitutions \( \{ x_{i} \mapsto \tau_{i} \} \).

2. If \( a_{1} \sqsubseteq a_{2} \) and \( P; E \vdash \alpha \uparrow a_{1} \) and \( P; E \vdash a_{2} \uparrow a_{2}' \) then \( a_{1}' \sqsubseteq a_{2}' \).

3. If \( a_{1} =_{h} a_{2} \), then for any valid lock expression \( \ell \) such that \( \ell \notin h \) and \( \ell \notin n \), we have

\[
\begin{align*}
a_{1} =_{h \cup \ell} a_{2} \\
\alpha_{1} =_{h \cup \ell} \alpha_{2}
\end{align*}
\]

4. If \( a_{1} =_{h} a_{2} \), then for all \( a: \)

\[
\begin{align*}
a_{1}; & \sqsubseteq h; a_{2} \\
a; a_{1} & \sqsubseteq h; a_{2}
\end{align*}
\]

**Proof**: Follows from the definitions of these operations.

**Lemma 2** (Context Subexpression). Suppose there is a deduction that concludes \( P; E \vdash E[e] : c \cdot a \). Then that deduction contains, at a position corresponding to the hole in \( E \), a subdeduction that concludes \( P; E \vdash e : c' \cdot a' \).

**Proof**: Induction over the derivation of \( P; E \vdash E[e] : c \cdot a \).

**Lemma 3** (Sequentiality). Let \( E \) a context, \( E \) a well-formed environment, \( e \) an expression, and \( \alpha \) a value. Suppose the following:

- \( P; E \vdash E[e] : c \cdot a \)
- \( P; E \vdash E[v] : c \cdot a' \)
- \( e \) is not a value

Then we may conclude

\[
\begin{align*}
\gamma & \cdot a \sqsubseteq a' \\text{ for all } a' \\
Y(E, \gamma' \cdot a) & \sqsubseteq a'
\end{align*}
\]

**Proof**: By induction over the derivation of \( P; E \vdash E[e] : c \cdot a \). First, by Lemma 2, we obtain \( P; E \vdash e : c' \cdot a'' \).

- \( E \equiv [\cdot] \): As assumptions, we have

\[
\begin{align*}
P; E & \vdash [e] : c \cdot a \\
P; E & \vdash [v] : c \cdot a'
\end{align*}
\]

where \( a' = \text{AF} \). From the first assumption, we immediately have \( a'' = a \). Also \text{AF} is the right identity for sequential composition. Hence we have the equality

\[
(a'', a') = (a; \text{AF}) = a
\]

By Lemma 17, we know

\[
Y(E, a'') \sqsubseteq a''
\]

so by Lemma 1, we have

\[
Y(E, a''); a' \sqsubseteq a''; a'
\]

and we may conclude with our desired result

\[
Y(E, a''); a' \sqsubseteq a
\]

- \( E \equiv \text{new } c(\tau, E', \tau) \): As assumptions, we have

\[
\begin{align*}
P; E & \vdash \text{new } c(\tau, E'[e], \tau) : c \cdot a \\
P; E & \vdash \text{new } c(\tau, E'[v], \tau) : c \cdot a'
\end{align*}
\]

Let us take \( E'[e] \) to be in the \( k \)-th position in the argument list. These assumptions are derivable only through rule \text{[EXP NEW]}; hence, we know

\[
\begin{align*}
class c \{ d_{i} & \mapsto \{ e_{i} \in 1..n \} \} \in P \\
P; E & \vdash v_{i} : c_{i} \cdot a_{i} \quad \forall i \in 1..k-1 \\
P; E & \vdash E'[e] : c_{i} \cdot a_{i} \\
P; E & \vdash E'[v] : c_{i} \cdot a_{i} \\
P; E & \vdash c_{i} \cdot a_{i} \quad \forall i \in k+1..n \\
a_{i} & = \text{AF} \quad \forall i \in 1..k-1 \\
a & = (a_{1}, \ldots, a_{k}, \ldots, a_{n}; \text{AM}) \\
a' & = (a_{1}, \ldots, a_{k}, \ldots, a_{n}; \text{AM})
\end{align*}
\]

Since \text{AF} is identity for sequential composition, we can simplify \( a \) and \( a' \) as follows:

\[
\begin{align*}
a & = (a_{k}, \ldots, a_{n}; \text{AM}) \\
a' & = (a_{k}, \ldots, a_{n}; \text{AM})
\end{align*}
\]

By IH, we have

\[
Y(E', a'') : a' \sqsubseteq a_{k}
\]

Since \text{locks}(E') = \text{locks}(E), we have

\[
Y(E, a'') : a' \sqsubseteq a
\]

By Lemma 1, we get

\[
Y(E, a'') : a' \sqsubseteq a
\]

Hence, by substitution, we obtain our desired result

\[
Y(E, a'') : a' \sqsubseteq a
\]

- \( E \equiv E', f \): As assumptions, we have

\[
\begin{align*}
P; E & \vdash E'[e], f : c \cdot a \\
P; E & \vdash E'[v], f : c \cdot a'
\end{align*}
\]
These assumptions are derivable only through rules [EXP REF] and [EXP REF RACE]; hence, we know
\[ P; E \vdash E'[\ell] : c_0 \cdot a \]
\[ P; E \vdash E'[\ell] : c_0 \cdot a' \]
\[ \text{class } c_0 \{ \ldots f \ldots \} \in P \]
\[ a = a_0; a_f \]
\[ a' = a_0'; a_f \]
where \( a_f \) is one of four cases

- **AM** if \( f \in \text{Normal} \) and \( \gamma = . \)
- **AF** if \( f \in \text{Final} \) and \( \gamma = . \)
- **AN** if \( f \in \text{Volatile} \) and \( \gamma = . \)
- **CL** if \( f \in \text{Volatile} \) and \( \gamma = . \)

By IH, we have
\[ Y(E', a'') ; a'_1 \subseteq a_0 \]
Since \( \text{locks}(E') = \text{locks}(E) \), we have
\[ Y(E, a'') ; a'_0 \subseteq a_0 \]
By Lemma 1, we get
\[ (Y(E, a'') ; a'_0; a_f) \subseteq (a_0; a_f) \]
Hence we obtain our desired result
\[ Y(E, a'') ; a' \subseteq a \]

- **\( E \equiv E', f = e' \):** As assumptions, we have
  \[ P; E \vdash E'[\ell] : c_0 \cdot a \]
  \[ P; E \vdash E'[\ell] : c_0 \cdot a' \]
  \[ \text{class } c_0 \{ \ldots f \ldots \} \in P \]
  \[ a = a_0; a_f \]
  \[ a' = a_0'; a_f \]
where \( a_f \) is one of four cases

- **AM** if \( f \in \text{Normal} \) and \( \gamma = . \)
- **AF** if \( f \in \text{Final} \) and \( \gamma = . \)
- **AN** if \( f \in \text{Volatile} \) and \( \gamma = . \)
- **CL** if \( f \in \text{Volatile} \) and \( \gamma = . \)

By IH, we have
\[ Y(E', a'') ; a'_1 \subseteq a_t \]
Since \( \text{locks}(E') = \text{locks}(E) \), we have
\[ Y(E, a'') ; a'_1 \subseteq a_t \]
By Lemma 1, and since AF is identity for sequential composition, we get
\[ (Y(E, a'') ; a'_0; a_f) \subseteq (a_0; a_0; a_f) \]
Hence, by substitution, we obtain our desired result
\[ Y(E, a'') ; a' \subseteq a \]

- **\( E \equiv E', m(\tau) \):** As assumptions, we have
  \[ P; E \vdash E'[\ell] , m(\tau) : c \cdot a \]
  \[ P; E \vdash E'[\ell] , m(\tau) : c \cdot a' \]
These assumptions are derivable only through rules [EXP ASSIGN] and [EXP ASSIGN RACE]; hence, we know
\[ P; E \vdash E'[\ell] : c_0 \cdot a \]
\[ P; E \vdash E'[\ell] : c_0 \cdot a' \]
\[ \text{class } c_0 \{ \ldots \text{meth} \ldots \} \]
\[ \text{meth} = a_m = c \cdot m(d;x_i^{i\in1..n}) \}
\[ P; E \vdash e_i : d_i ; a_i \quad \forall i \in 1..n \]
\[ P; E \vdash \text{this} := E'[\ell] , x_i := e_i^{i\in1..n} \uparrow a_\ell \]
\[ P; E \vdash \text{this} := E'[\ell] , x_i := e_i^{i\in1..n} \uparrow a_\ell \]
\[ a = a_0; a_1; \ldots; a_n ; [\tau] ; a_\ell \]
\[ a' = a_0'; a_1; \ldots; a_n ; [\tau] ; a_\ell \]
\[ a \subseteq \text{AN} \]
\[ a' \subseteq \text{AN} \]

By IH, we have
\[ Y(E', a'') ; a'_1 \subseteq a_0 \]
Since \( \text{locks}(E') = \text{locks}(E) \), we have
\[ Y(E, a'') ; a'_1 \subseteq a_0 \]
By Lemma 1, we get
\[ (Y(E, a'') ; a'_0; a_f) \subseteq (a_0; a_0; a_f) \]
Hence, by substitution, we obtain our desired result
\[ Y(E, a'') ; a' \subseteq a \]

- **\( E \equiv E', \tau \):** Similar argument to previous case.
• \( E \equiv \nu x. m(\tau, E', \tau) \): As assumptions, we have

\[
\begin{align*}
P; E &\vdash \nu x. m(\tau, E'[e], \tau) : c \cdot a \\
P; E &\vdash \nu x. m(\tau, E'[e], \tau) : c \cdot a'
\end{align*}
\]

These assumptions are derivable only through the rule \[\text{[exp invoke]}\]; hence, we know the following, where we take \( E'[e] \) to be in the \( k \)th position in the list of arguments.

\[
\begin{align*}
P; E &\vdash v' : a_0 \cdot a_0 \\
\text{class} &a_0 \{ \ldots \text{meth} \ldots \} \\
\text{meth} &\equiv a_0 \cdot m(d_1, x_1 : e_1, \ldots, d_n, x_1 : e_n)(e') \\
P; E &\vdash v_1 : d_1 \cdot a_1 \quad \forall i \in 1..k - 1 \\
P; E &\vdash E'[e] : c_k \cdot a_k \\
P; E &\vdash E'[v] : c_k \cdot a_k \\
P; E &\vdash e_i : d_i \cdot a_i \quad \forall i \in k + 1..n \\
P; E &\vdash a_m[\text{this} := v', x_1 := v_1, \ldots, x_k := E'[e], x_{k+1} := E'[v], \ldots, x_n := E'[v], x_1 := e, \ldots, x_{k-1} := e, x_{k+1} := e][g] : a' \\
\end{align*}
\]

Both \( a \) and \( a' \) simplify to the following, due to \( \text{AF} \) being the identity of sequential composition:

\[
\begin{align*}
a &\equiv a_0; \ldots; a_n; [\gamma]; a_{k} \\
a' &\equiv a'_k; \ldots; a_n; [\gamma]; a_{k}'
\end{align*}
\]

By IH, we have

\[
Y(E', a'''); a'_k \sqsubseteq a_k
\]

Since \( \text{locks}(E') = \text{locks}(E) \), we have

\[
Y(E', a''); a'_k \sqsubseteq a_k
\]

By Lemma 1, we get

\[
(Y(E', a''); a') \equiv (Y(E', a'''); a'_{k}) \sqsubseteq (a_k; \ldots; a_n; [\gamma]; a_k')
\]

Since \( E'[e] \) cannot be a value due to the presence of \( e \), by Lemma 11, we have \( a'_k \sqsubseteq a_k \). Again by Lemma 1, we get

\[
(a_k; \ldots; a_n; [\gamma]; a'_k) \sqsubseteq (a_k; \ldots; a_n; [\gamma]; a_k') = a
\]

Combining, we get our desired result

\[
Y(E', a'''); a' \sqsubseteq a
\]

• \( E \equiv \rho, m(\tau, E', \tau) \): Similar argument to previous case.
• \( E \equiv \text{let } x = e' \text{ in } e' \): As assumptions, we have

\[
\begin{align*}
P; E &\vdash \text{let } x = E'[e] \text{ in } e' : c \cdot a \\
P; E &\vdash \text{let } x = E'[v] \text{ in } e' : c \cdot a'
\end{align*}
\]

These assumptions are derivable only through rule \[\text{[exp let]}\]; hence we know

\[
\begin{align*}
P; E &\vdash E'[e] : c_x \cdot a_x \\
P; E &\vdash E'[v] : c_x \cdot a_x' \\
P; E; c_x x &\vdash e' : c \cdot a_0 \\
P; E &\vdash a_x[x := E'[e]] : a_x \\
P; E &\vdash a_x[x := E'[v]] : a_x' \\
a &\equiv (a_x'; a_x) \\
a' &\equiv (a'_{x'}; a'_{x})
\end{align*}
\]

By IH, we have

\[
Y(E', a'''); a'_x \sqsubseteq a_x
\]

Since \( \text{locks}(E') = \text{locks}(E) \), we have

\[
Y(E, a''); a'_x \sqsubseteq a_x
\]

By Lemma 1, we get

\[
(Y(E, a''); a') \equiv (Y(E, a'''); a'_{x}) \sqsubseteq (a_x; a')
\]

Since \( E'[e] \) cannot be a value due to the presence of \( e \), by Lemma 11, we have \( a'_x \sqsubseteq a_x \). Applying Lemma 1, we get

\[
(a_x; a'_x) \sqsubseteq (a_x; a_1) = a
\]

Combining, we get our desired result

\[
Y(E, a'''); a' \sqsubseteq a
\]

• \( E \equiv \nu x. m(\tau, E', \tau) \): As assumptions, we have

\[
\begin{align*}
P; E &\vdash \nu x. m(\tau, E'[e], \tau) : c \cdot a \\
P; E &\vdash \nu x. m(\tau, E'[e], \tau) : c \cdot a'
\end{align*}
\]

These assumptions are derivable only through rule \[\text{[exp invoke]}\]; hence we know

\[
\begin{align*}
P; E &\vdash E'[e] : c_x \cdot a_x \\
P; E &\vdash E'[v] : c_x \cdot a_x' \\
P; E; c_x x &\vdash e' : c \cdot a_0 \\
P; E &\vdash a_x[x := E'[e]] : a_x \\
P; E &\vdash a_x[x := E'[v]] : a_x' \\
a &\equiv (a_x; a_x) \\
a' &\equiv (a'_{x'}; a'_x)
\end{align*}
\]

By IH, we have

\[
Y(E', a'''); a'_x \sqsubseteq a_x
\]

Since \( \text{locks}(E') = \text{locks}(E) \), we have

\[
Y(E, a''); a'_x \sqsubseteq a_x
\]

By Lemma 1, we get

\[
(Y(E, a''); a') \equiv (Y(E, a'''); a'_{x}) \sqsubseteq (a_x; a')
\]

Since \( E'[e] \) cannot be a value due to the presence of \( e \), by Lemma 11, we have \( a'_x \sqsubseteq a_x \). Applying Lemma 1, we get

\[
(a_x; a'_x) \sqsubseteq (a_x; a_1) = a
\]

Combining, we get our desired result

\[
Y(E, a'''); a' \sqsubseteq a
\]

• \( E \equiv \rho, m(\tau, E', \tau) \): Similar argument to previous case.
• \( E \equiv \text{let } x = e' \text{ in } e' \): As assumptions, we have

\[
\begin{align*}
P; E &\vdash \text{let } x = E'[e] \text{ in } e' : c \cdot a \\
P; E &\vdash \text{let } x = E'[v] \text{ in } e' : c \cdot a'
\end{align*}
\]

These assumptions are derivable only through rule \[\text{[exp let]}\]; hence we know

\[
\begin{align*}
P; E &\vdash E'[e] : c_x \cdot a_x \\
P; E &\vdash E'[v] : c_x \cdot a_x' \\
P; E; c_x x &\vdash e' : c \cdot a_0 \\
P; E &\vdash a_x[x := E'[e]] : a_x \\
P; E &\vdash a_x[x := E'[v]] : a_x' \\
a &\equiv (a_x; a_x) \\
a' &\equiv (a'_{x'}; a'_x)
\end{align*}
\]

By IH, we have

\[
Y(E', a'''); a'_x \sqsubseteq a_x
\]

Since \( \text{locks}(E') = \text{locks}(E) \), we have

\[
Y(E, a''); a'_x \sqsubseteq a_x
\]

By Lemma 1, we get

\[
(Y(E, a''); a') \equiv (Y(E, a'''); a'_{x}) \sqsubseteq (a_x; a')
\]

Since \( E'[e] \) cannot be a value due to the presence of \( e \), by Lemma 11, we have \( a'_x \sqsubseteq a_x \). Applying Lemma 1, we get

\[
(a_x; a'_x) \sqsubseteq (a_x; a_1) = a
\]

Combining, we get our desired result

\[
Y(E, a'''); a' \sqsubseteq a
\]

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.

• \( E \equiv \text{in-sync } \rho, E' \): As assumptions, we have

\[
\begin{align*}
P; E &\vdash \text{in-sync } \rho, E'[e] : c \cdot a \\
P; E &\vdash \text{in-sync } \rho, E'[v] : c \cdot a'
\end{align*}
\]

These assumptions are derivable only through rule \[\text{[exp insync]}\]; hence we know

\[
\begin{align*}
P; E &\vdash \text{in-sync } \rho, E'[e] : c \cdot a \\
P; E &\vdash \text{in-sync } \rho, E'[v] : c \cdot a'
\end{align*}
\]

By IH, we have

\[
Y(E', a'''); a'_x \sqsubseteq a_x
\]
By repeated application of Lemma 17, we have
\[
Y(E, a'') \subseteq a''
\]
\[
Y(E', Y(E, a'')) \subseteq Y(E', a''')
\]
Since \(\text{locks}(E') \subseteq \text{locks}(E)\), we have
\[
Y(E, a''') = Y(E', Y(E, a'''))
\]
and so by transitivity and Lemma 1,
\[
Y(E, a'') \subseteq Y(E', a''')
\]
\[
Y(E, a''') ; a_b' \subseteq a_b
\]
We may strengthen the last relation by inclusion of \(\{\rho\}\) in the held lockset:
\[
Y(E, a''') ; a_b' \subseteq (\rho) a_b
\]
Then by Lemma 20, we have
\[
Y(E, a''') ; SI(\rho, a_b') \subseteq (\rho) SI(\rho, a_b)
\]
Since the left and right hand effects both are not conditional on \(\rho\), by Lemma 18 we have
\[
Y(E, a''') ; SI(\rho, a_b') \subseteq SI(\rho, a_b)
\]
Hence we obtain our desired result:
\[
Y(E, a''') ; a' \subseteq a
\]

**Lemma 4 (Context Replacement).** Suppose the following:
\[
P; E \vdash E'[e_1] : c \cdot a_1, \text{ and}
\]
\[
P; E \vdash e_1 : d \cdot a_1, \text{ and}
\]
\[
e_1 \text{ not a value, and}
\]
\[
P; E \vdash e_2 : d \cdot a_2, \text{ and}
\]
\[
a_2 \subseteq a_1.
\]
Then \(P; E \vdash E[e_2] : c \cdot a_2 \subseteq a\).

**Proof** Let \(h = \emptyset\), \(n = 0\), and \(\kappa_y = \text{AF}\). The assumption \(a_2 \subseteq a_1\) implies
\[
\kappa_y ; a_2 \subseteq \text{locks}(E) ; a_1
\]
Then applying Lemma 5, we obtain the judgment \(P; E \vdash E[e_2] : c \cdot a_1\) where \(a' \subseteq a\).

**Lemma 5 (Context Replacement 2).** Suppose the following:
\[
P; E \vdash E'[e_1] : c \cdot a_1', \text{ and}
\]
\[
P; E \vdash e_1 : d \cdot a_1', \text{ and}
\]
\[
e_1 \text{ not a value, and}
\]
\[
P; E \vdash e_2 : d \cdot a_2, \text{ and}
\]
\[
\exists h, n \cdot h \cap n \cap \text{locks}(E)
\]
\[
\kappa_y ; a_2 \subseteq \text{locks}(E) ; a_1
\]
Then \(P; E \vdash E[e_2] : c \cdot a_2\) and \((\kappa_y ; a_2) \subseteq h a_1\).

**Proof** By induction over the structure of \(E\).

- \(E \equiv [\ ]\): As assumptions, we have
\[
P; E \vdash e_1 : c \cdot a_1
\]
\[
P; E \vdash e_2 : d \cdot a_2
\]
and may obtain the equalities
\[
a_1' = a_1
\]
\[
a_2' = a_2
\]
Since \(E \equiv [\ ]\), we have \(\text{locks}(E) = \emptyset\). From the assumption \(\kappa_y ; a_2 \subseteq \text{locks}(E) ; a_1\) and substituting, we obtain
\[
\kappa_y ; a_2 \subseteq a_1
\]
which is our desired result.

- \(E \equiv \text{new } c (\tau, E', \tau)\): As an assumption, we have
\[
P; E \vdash \text{new } c (\tau, \tau') : c \cdot a_1'
\]
Let us take \(E'[e_1]\) to be in the \(k^{th}\) position in the list of arguments. This assumption is derivable only through rule \([\text{EXP NEW}]\); hence we know
\[
class c \{ d_i \cdot x_i \in 1..n \} \in P
\]
\[
P; E \vdash v_i : c_i \cdot a_i \quad \forall i \in 1..k - 1
\]
\[
P; E \vdash E'[e_1] : c_k \cdot a_k
\]
\[
P; E \vdash e_1 : c_i \cdot a_i \quad \forall i \in k + 1..n
\]
\[
a_i = \text{AF} \quad \forall i \in 1..k - 1
\]
\[
a_i' = (a_1; \ldots; a_i; \ldots ; a_n; \text{AM})
\]
We have \(\text{locks}(E) = \text{locks}(E')\), since the \(E \neq \text{in-sync} \ell E'\).
By HI, we get
\[
P; E \vdash E'[e_2] : c_k \cdot a_k'
\]
\[
(\kappa_y ; a_k') \subseteq h a_k
\]
Using rule \([\text{EXP NEW}]\), we may conclude
\[
P; E \vdash \text{new } c (\tau, \tau') : c \cdot a_2'
\]
\[
a_2' = (a_1; \ldots; a_n; \text{AM})
\]
Since \(\text{AF}\) is the identity for sequential composition, we may simplify \(a_1\) and \(a_2\) as follows:
\[
a_1' = (a_k; \ldots; a_n; \text{AM})
\]
\[
a_2' = (a_1; \ldots; a_n; \text{AM})
\]
By Lemma 1 and \((\kappa_y ; a_2) \subseteq h a_1\), we have
\[
(\kappa_y ; a_k') \subseteq h a_k
\]
Hence we obtain our desired result
\[
(\kappa_y ; a_2') \subseteq h a_1
\]

- \(E \equiv \text{E'} \cdot f\): As an assumption, we have
\[
P; E \vdash E'[e_1] : c \cdot a_1'
\]
This assumption is derivable only through rules \([\text{EXP REF}]\) and \([\text{EXP REF RACE}]\); hence we know
\[
P; E \vdash E'[e_2] : c \cdot a_2'
\]
\[
\text{class } c \{ \ldots \cdot f \ldots \} \in P
\]
\[
a_2' = (a_1; f)
\]
where \(a_1 \) is one of four cases
\[
= \text{AM} \quad \text{if } f \in \text{Normal} \text{ and } \gamma = .
\]
\[
= \text{AF} \quad \text{if } f \in \text{Final} \text{ and } \gamma = .
\]
\[
= \text{AN} \quad \text{if } f \in \text{Volatile} \text{ and } \gamma = .
\]
\[
= \text{CL} \quad \text{if } f \in \text{Volatile} \text{ and } \gamma = ..
\]
We have \(\text{locks}(E) = \text{locks}(E')\), since \(E \neq \text{in-sync} \ell E'\).
By HI, we have
\[
P; E \vdash E'[e_2] : c \cdot a_2'
\]
\[
(\kappa_y ; a_2') \subseteq h a_2
\]
Using rules \([\text{EXP REF}]\) and \([\text{EXP REF RACE}]\), we may conclude
\[
P; E \vdash E'[e_1] : f \cdot c \cdot a_2'
\]
\[
a_2' = (a_1; f)
\]
By Lemma 1 and \((\kappa_y ; a_2) \subseteq h a_1\), we get
\[
(\kappa_y ; a_2; a_f) \subseteq h (a_1; a_f)
\]
Hence we obtain our desired result
\((\kappa_0; a_1) \sqsubseteq_h a_1'\)

\(\mathcal{E} \equiv \mathcal{E}', f = c: \text{As an assumption, we have}\)
\(P; E \vdash \mathcal{E}'[e_1], f = c \cdot c': a_1'\)

This assumption is derivable only through rules [EXP ASSIGN] and [EXP ASSIGN RACE]; hence we know
\(P; E \vdash \mathcal{E}'[e_1] : c \cdot a_0\)
\(\text{class } c_0 \cdot \ldots \cdot c \cdot f \ldots \gamma \in P\)
\(a_1' = (a_0; a_1; a_f)\)

where \(a_f\) is one of three cases

AM: if \(f \in \text{Normal}\) and \(\gamma = \cdot\)
AN: if \(f \in \text{Volatile}\) and \(\gamma = \cdot\)
CL: if \(f \notin \text{Volatile}\) and \(\gamma = \cdot\)

We have \(\text{locks}(\mathcal{E}) = \text{locks}(\mathcal{E}')\), since \(\mathcal{E} \not\equiv \text{in-sync } \ell \mathcal{E}'\). By IH, we have
\(P; E \vdash \mathcal{E}'[e_2] : c_0 \cdot a_0'\)
\((\kappa_0; a_0') \sqsubseteq_h a_0\)

Using rules [EXP ASSIGN] and [EXP ASSIGN RACE], we may conclude
\(P; E \vdash \mathcal{E}'[e_2] : c \cdot a_1'\)
\(a_1' = (a_0; a_1; a_f)\)

By Lemma 1 and \((\kappa_0; a_0') \sqsubseteq_h a_0\), we get
\((\kappa_0; a_0'; a_1; a_f) \sqsubseteq_h (a_0; a_1; a_f)\)

Hence we obtain our desired result
\((\kappa_0; a_0') \sqsubseteq_h a_0'\)

\(\mathcal{E} \equiv \mathcal{E}', v \cdot f = \mathcal{E}'\): As an assumption, we have
\(P; E \vdash \mathcal{E}'[e_1] : c \cdot a_1'\)

This assumption is derivable only through rules [EXP ASSIGN] and [EXP ASSIGN RACE]; hence we know
\(P; E \vdash \mathcal{E}'[e_1] : c \cdot a_0\)
\(\text{class } c_0 \cdot \ldots \cdot c \cdot f \ldots \gamma \in P\)
\(a_1' = (a_0; a_1; a_f)\)

where \(a_f\) is one of three cases

AM: if \(f \in \text{Normal}\) and \(\gamma = \cdot\)
AN: if \(f \in \text{Volatile}\) and \(\gamma = \cdot\)
CL: if \(f \notin \text{Volatile}\) and \(\gamma = \cdot\)

We have \(\text{locks}(\mathcal{E}) = \text{locks}(\mathcal{E}')\), since \(\mathcal{E} \not\equiv \text{in-sync } \ell \mathcal{E}'\). By IH, we have
\(P; E \vdash \mathcal{E}'[e_2] : c \cdot a_0'\)
\((\kappa_0; a_0') \sqsubseteq_h a_0\)

Using rules [EXP ASSIGN], we may conclude
\(P; E \vdash \mathcal{E}'[e_2] : c \cdot a_2'\)
\(a_2' = (a_0; a_1; a_f)\)

By a separate application of Lemma 1 to \(a_0' \sqsubseteq a_0\), we have
\((\kappa_0; a_0'; a_1; a_f) \sqsubseteq_h (a_0; a_1; a_f)\)

Combining these, and substituting definitions of \(a_1'\) and \(a_2'\), we obtain our desired result
\((\kappa_0; a_0') \sqsubseteq_h a_0'\)

\(\mathcal{E} \equiv \mathcal{E}', m \# (\mathcal{T})\): Similar argument to previous case.

\(\mathcal{E} \equiv v, m (\mathcal{T}, \mathcal{E}', \mathcal{T})\): As an assumption, we have
\(P; E \vdash \mathcal{E}'[e_1] : c \cdot a_1'\)

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
\(P; E \vdash v \cdot c_0 \cdot a_0\)
\(\text{class } c_0 \cdot \ldots \cdot \text{meth } \ldots \gamma \in P\)
\(\text{meth } = a_0 \cdot m (d_i, x_i) \in \{ e' \}\)
\(P; E \vdash v_i \cdot d_i \cdot a_k\)
\(\forall i \in 1 .. k - 1\)
\(P; E \vdash \mathcal{E}'[e_1] : d_k \cdot a_k\)
\(\forall i \in k \ldots n\)
\(P; E \vdash \mathcal{E}'[e_1] : x_k := \mathcal{E}'[e_1], x_i := e_i \cdot e_i \in k+1, k+1 \uparrow a_m\)

\(a_0' = \text{AF}\)
\(a_0' \sqsubseteq_h a_0\)
\(a_1' = (a_0; a_1; \ldots; a_k; \ldots; a_0; [\gamma]; a_0')\)

We have \(\text{locks}(\mathcal{E}) = \text{locks}(\mathcal{E}')\), since \(\mathcal{E} \not\equiv \text{in-sync } \ell \mathcal{E}'\). By IH, we have
\(P; E \vdash \mathcal{E}'[e_2] : d_k \cdot a_k'\)
\((\kappa_0; a_k') \sqsubseteq_h a_k\)
From $P \vdash E$ and the lifting rules we may derive
\[ P; E \vdash a_m \{\text{this} := v, x_1 := v_1, \ldots, x_k := v_k \} \uparrow a_m' \]
By Lemma 11 and the fact that $E'[e_1]$ cannot be a value, we have
\[ a_m' \subseteq a_m'' \]
Using rule [\textsc{exp invoke}], we may conclude
\[ P; E \vdash v, y; m \{\pi, E'[e_2], \tau\} : c \cdot a'_2 \]
\[ a'_2 = (a_0; a_1; \ldots; a_k; \ldots; a_m; [\gamma]; a_m') \]
Since $AF$ is the identity for sequential composition, we may simplify $a'_1$ and $a'_2$ to the following:
\[ a'_1 = a_k; \ldots; a_n; [\gamma]; a_m' \]
\[ a'_2 = a'_k; \ldots; a'_n; [\gamma]; a_m'' \]
By Lemma 1 and $(\kappa_y; a'_k) \subseteq_h a_k$ we have
\[ (\kappa_y; a'_k; \ldots; a_n; [\gamma]; a_m' \subseteq_h (a_k; \ldots; a_n; [\gamma]; a_m') \]
By a separate application of Lemma 1 to $a_m'' \subseteq a_m'$, we have
\[ (a_k; \ldots; a_n; [\gamma]; a_m'' \subseteq (a_k; \ldots; a_n; [\gamma]; a_m') \]
Combining these, and substituting definitions of $a'_1$ and $a'_2$, we obtain our desired result:
\[ (\kappa_y; a'_2) \subseteq_h a_1' \]
- $E \equiv \rho; m \{\pi, E', \tau\}$: Similar argument to previous case.
- $E \equiv \rho; m \{\pi, E', \tau\}$: As an assumption, we have
\[ P; E \vdash \text{let } x = E'[e_1] \in c : c \cdot a'_1 \]
This assumption is derivable only through rule [\textsc{exp let}]; hence we know
\[ P; E \vdash E'[e_1] : c_x \cdot a_x \]
\[ P; E, c, x \vdash x : c \cdot a_b \]
\[ P; E \vdash a_b[x := E'[e_1]] \uparrow a_\ell \]
\[ a_1' = (a_x; a_\ell) \]
We have $\text{locks}(E) = \text{locks}(E')$, since $E \not\in \text{in-sync } E'$. By IH, we have
\[ P; E \vdash E'[e_2] : c_x \cdot a'_x \]
\[ (\kappa_y; a'_x) \subseteq_h a_x \]
From $P \vdash E$ and the lifting rules we may conclude
\[ P; E \vdash a_b[x := E'[e_2]] \uparrow a_\ell \]
By Lemma 11 and the fact that $E'[e_1]$ cannot be a value, we have
\[ a_\ell' \subseteq a_\ell \]
Using rule [\textsc{exp let}], we may conclude
\[ P; E \vdash \text{let } x = E'[e_2] \in c : c \cdot a'_2 \]
\[ a'_2 = (a'_x; a'_1) \]
By Lemma 1 and $(\kappa_y; a'_2) \subseteq a_x$, we get
\[ (\kappa_y; a'_2) = (\kappa_y; a'_x; a'_1) \subseteq_h (a_x; a_\ell) \]
By a separate application of Lemma 1 to $a'_x \subseteq a_x$, we have
\[ (a_x; a'_x) \subseteq (a_x; a_\ell) = a'_1 \]
Combining these, we get our desired result
\[ (\kappa_y; a'_2) \subseteq_h a'_1 \]
- $E \equiv \text{if } E' \in e_2$: As an assumption, we have
\[ P; E \vdash \text{if } E'[e_1] \in e \in e : c \cdot a'_1 \]
This assumption is derivable only through rule [\textsc{exp if}]; hence we know
\[ P; E \vdash E'[e_1] : d \cdot a_0 \]
\[ P; E \vdash e \in c : a_0 \]
\[ P; E \vdash e \in c : a_0 \]
\[ a'_1 = (a_0; (a_\ell \cup a_x)) \]
We have $\text{locks}(E) = \text{locks}(E')$, since $E \not\in \text{in-sync } E'$. By IH, we have
\[ P; E \vdash E'[e_2] : c \cdot a'_0 \]
\[ (\kappa_y; a'_0) \subseteq_h a_0 \]
Using rule [\textsc{exp if}], we may conclude
\[ P; E \vdash \text{if } E'[e_1] \in e_{e_2} \in e : c \cdot a'_2 \]
\[ a'_2 = (a'_0; (a_\ell \cup a_x)) \]
By Lemma 1 and $(\kappa_y; a'_0) \subseteq_h a_0$, we get
\[ (\kappa_y; a'_0; (a_\ell \cup a_x)) \subseteq_h (a_0; (a_\ell \cup a_x)) \]
By substituting definitions of $a'_1$ and $a'_2$, we obtain our desired result
\[ (\kappa_y; a'_2) \subseteq_h a'_1 \]
- $E \equiv \text{in-sync } E'$: As an assumption, we have
\[ P; E \vdash \text{in-sync } \rho; E'[e_1] : c \cdot a'_1 \]
This assumption is derivable only through rule [\textsc{exp insync}]; hence we know
\[ P; E \vdash \text{lock } E'[e_1] \]
\[ P; E \vdash E'[e_1] : c \cdot a_0 \]
\[ a'_1 = \text{SI}(\rho, a_0) \]
Rewriting our assumption, since
\[ \text{locks}(E) = \{\rho\} \cup \text{locks}(E') \]
\[ \{\rho\} \not\subseteq \text{locks}(E) \]
we have
\[ \kappa_y; a_2 \subseteq_h \{\rho\} \cup \text{locks}(E') \]
By IH, we have
\[ P; E \vdash E'[e_2] : c \cdot a'_0 \]
\[ (\kappa_y; a'_0) \subseteq_h \{\rho\} a_0 \]
By Lemma 20, we have
\[ \kappa_y; \text{SI}(\rho, a'_0) \subseteq_h \text{SI}(\rho, a_0) \]
Since the left and right sides both do not contain a conditional on $\rho$, by Lemma 18, we have
\[ \kappa_y; \text{SI}(\rho, a'_0) \subseteq_h \text{SI}(\rho, a_0) \]
By substituting definitions of $a'_1$ and $a'_2$, we obtain our desired result
\[ \kappa_y; a'_2 \subseteq_h a'_1 \]
Lemma 6 (Substitution). Suppose we have \( P; E \vdash v : c\cdot AF \). Then the following is true:

1. If \( P \vdash (E, c, E') \) then \( P; (E, E'[x := v]) \).
2. If \( P; (E, c, E') \vdash \text{meth} \) then \( P; (E, E'[x := v]) \vdash \text{meth}[x := v] \).
3. If \( P; (E, c, E') \vdash a \) then \( P; (E, E'[x := v]) \vdash a[x := v] \).
4. If \( P; (E, c, E') \vdash \text{lock } \ell \) then \( P; (E, E'[x := v]) \vdash \text{lock } \ell[x := v] \).
5. If \( P; (E, c, E') \vdash a \uparrow a' \) then \( P; (E, E'[x := v]) \vdash a[x := v] \uparrow a'[x := v] \).
6. If \( P; (E, c, E') \vdash c \cdot a \) then \( P; (E, E'[x := v]) \vdash e[x := v] : c' \cdot a[x := v] \).

Proof. By simultaneous induction on all parts of the lemma.

- Assume \( P \vdash (E, c, E') \). 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 \( P; (E, c, E') \vdash \text{meth} \), where \( \text{meth} = a d m(\overline{a} \pi) \).

Then we know, by rule [METHOD], that

\[
P; (E, c, E', \overline{a} \pi) \vdash e : d \cdot a'
\]

\[
P; (E, c, E', \overline{a} \pi) \vdash a
\]

\[
a' \subseteq a[x := v]
\]

By IH, we have

\[
P; (E, E'[x := v], \overline{a} \pi[x := v]) \vdash e[x := v] : d \cdot a'[x := v]
\]

\[
P; (E, E'[x := v], \overline{a} \pi[x := v]) \vdash a[x := v]
\]

By Lemma 1,

\[
a'[x := v] \subseteq a[x := v]
\]

By rule [METHOD], we may conclude

\[
P; (E, E'[x := v]) \vdash a[x := v] d m(\overline{a} \pi[x := v]) \cdot e[x := v]
\]

and hence we obtain our desired result

\[
P; (E, E'[x := v]) \vdash \text{meth}[x := v]
\]

- Assume \( P; (E, c, E') \vdash a \). 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

\[
P; (E, E'[x := v])
\]

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

\[
P; (E, c, x, E') \vdash \text{lock } \ell
\]

\[
P; (E, c, x, E') \vdash a_1
\]

\[
P; (E, c, x, E') \vdash a_2
\]

By IH, we have

\[
P; (E, E'[x := v]) \vdash \text{lock } \ell[x := v]
\]

\[
P; (E, E'[x := v]) \vdash a_1[x := v]
\]

\[
P; (E, E'[x := v]) \vdash a_2[x := v]
\]

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 \( P; (E, c, x, E') \vdash a \uparrow a' \). We proceed by case analysis on the structure of effect \( a \).

  a = b. By rule [LIFT BASE], we know

\[
P \vdash (E, c, x, E')
\]

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

\[
a = a[x := v]
\]

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

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 \( \ell \).

  - P; (E, c, x, E') \vdash \text{lock } \ell. By rule [LIFT GOOD LOCK], we know

\[
P; (E, c, x, E') \vdash a_1 \uparrow a_1'
\]

\[
P; (E, c, x, E') \vdash a_2 \uparrow a_2'
\]

By rule [LIFT BAD LOCK], we may conclude

\[
P; (E, E'[x := v]) \vdash \ell[x := v] ? a_1[x := v] : a_2[x := v]
\]

\[
P; (E, E'[x := v]) \vdash a_1'[x := v] \uparrow a_2'[x := v]
\]

Hence we obtain our desired result

\[
P; (E, E'[x := v]) \vdash a[x := v] \uparrow a'[x := v]
\]

Since the lock judgment does not hold, \( \ell \) must contain some non-constant expression \( e \). Thus \( \ell[x := v] \) would
still contain e and hence the locking judgment will not hold for ℓ[x := v]. By IH, we have

\[ P; (E, E'[x := v]) \vdash \text{lock} \ell[x := v] \]
\[ P; (E, E'[x := v]) \vdash a_1 \uparrow a_1' \]
\[ P; (E, E'[x := v]) \vdash a_2 \uparrow a_2' \]

Then by rule \text{[LIFT BAD LOCK]}, we may conclude

\[ P; (E, E'[x := v]) \vdash ( \ell[x := v] \uparrow a_1' [x := v] \uparrow a_2' [x := v] ) \]

Hence we obtain our desired result

\[ P; (E, E'[x := v]) \vdash a \uparrow [x := v] \]

\[ \bullet \quad \text{Assume } P; (E, c x, E') \vdash e \cdot d \cdot a. \text{ We proceed by induction over the derivation of this judgment.} \]

\[ e \equiv y. \text{ Then our assumption is} \]

\[ P; (E, c x, E') \vdash y : d \cdot a \]

Since y is a variable, this assumption is derivable only through rule \text{[EXP VAR]; hence we know}

\[ P \vdash (E, c x, E') \]
\[ a = AF \]

We proceed by case analysis of membership of (d y) in the environment; one of three cases holds:

\[ (d y) \in E \quad \text{By IH, we have} \]
\[ P \vdash E, E'[x := v] \]

so we may still conclude, by \text{[EXP VAR]},

\[ P; (E, E'[x := v]) \vdash y : d \cdot AF \]

Since y \neq x and AF is a combined (basic) effect, we have

\[ P; (E, E'[x := v]) \vdash y [x := v] : d \cdot AF \]

Hence we obtain our desired result

\[ P; (E, E'[x := v]) \vdash e [x := v] : d \cdot a[x := v] \]

\[ \quad \text{By assumption, we have} \]

\[ P; E \vdash v : c \cdot AF \]

By Lemma 12, we can strengthen the environment, so we have

\[ P; (E, E'[x := v]) \vdash v : c \cdot AF \]

Since \( v = x[x := v] \) and AF = AF[x := v], by substituting we have

\[ P; (E, E'[x := v]) \vdash x [x := v] : c \cdot AF \]

Hence we obtain our desired result

\[ P; (E, E'[x := v]) \vdash e [x := v] : d \cdot a[x := v] \]

\[ \quad \text{By IH, we have} \]

\[ P \vdash E, E'[x := v] \]

Furthermore, (d y) \in E'[x := v], since y \neq x and type d does not capture x. By rule \text{[EXP VAR]}, we may conclude

\[ P; (E, E'[x := v]) \vdash y : d \cdot AF \]

Since \( y = y[x := v] \) and AF = AF[x := v], we have

\[ P; (E, E'[x := v]) \vdash y[x := v] : d \cdot AF \]

Hence we obtain our desired result

\[ P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v] \]

\[ \bullet \quad e \equiv \text{null}. \text{ Then our assumption is} \]

\[ P; (E, c x, E') \vdash \text{null} : d \cdot a \]

This assumption is derivable only with rule \text{[EXP NULL]; hence we know}

\[ P \vdash (E, c x, E') \]
\[ a = AF \]

By IH, we have

\[ P \vdash (E, E'[x := v]) \]

By rule \text{[EXP NULL]}, we may conclude

\[ P; (E, E'[x := v]) \vdash \text{null} : d \cdot AF \]

Since \text{null} = \text{null}[x := v] and AF = AF[x := v], we have

\[ P; (E, E'[x := v]) \vdash \text{null}[x := v] : d \cdot AF \]

Hence we obtain our desired result

\[ P; (E, E'[x := v]) \vdash e [x := v] : d \cdot a[x := v] \]

\[ \bullet \quad e \equiv \rho. \text{ Then our assumption is} \]

\[ P; (E, c x, E') \vdash \rho : d \cdot a \]

This assumption is derivable only with rule \text{[EXP ADDR]; hence we know}

\[ P \vdash (E', c x, E'') \]
\[ (E, c x, E') = (E'', \rho \cdot \rho, E''') \]
\[ \rho \in \text{Addr}_A \]
\[ a = AF \]

Since \( \rho \) is an address, and \( x \) is a variable, it must be, by the structure of environments (where for some state \( \sigma, E_{\sigma} \) is the environment initially passed), that \( \rho \in E \). Hence we have

\[ (E, c x, E') = (E_1, d \rho, E_2, c x, E_A) \]

By IH, we have

\[ P \vdash (E_1, d \rho, E_2, E_3[x := v]) \]

By rule \text{[EXP ADDR]}, we may conclude

\[ P; (E_1, d \rho, E_2, E_3[x := v]) \vdash \rho : d \cdot AF \]

Since \( \rho = \rho[x := v] \) and AF = AF[x := v], we have

\[ P; (E_1, d \rho, E_2, E_3[x := v]) \vdash \rho[x := v] : d \cdot AF \]

Hence we obtain our desired result

\[ P; (E, E'[x := v]) \vdash e [x := v] : d \cdot a[x := v] \]

\[ \bullet \quad e \equiv \rho, f. \text{ Then our assumption is} \]

\[ P; (E, c x, E') \vdash \rho \cdot f : d \cdot a \]

This assumption is derivable only through rules \text{[EXP REF]} and \text{[EXP REF RACE]; hence we know}

\[ P; (E, c x, E') \vdash \rho \cdot f : d \cdot a \]

\[ \text{class } d_0 \{ \ldots . \quad d \quad f \ldots \} \in P \]
\[ a = (a_0; a_f) \]
where $a_f$ is one of four cases

- **AM** if $f \in \text{Normal}$ and $\gamma = \ldots$
- **AF** if $f \in \text{Final}$ and $\gamma = \ldots$
- **AN** if $f \in \text{Volatile}$ and $\gamma = \ldots$
- **CL** if $f \in \text{Volatile}$ and $\gamma = \ldots$

By **IH**, we have

$$P; (E, E'[x := v]) \vdash e'[x := v] : d \cdot a_{o \cdot A}[x := v]$$

By rule **[exp ref]**/[**exp ref e**], we may conclude

$$P; (E, E'[x := v]) \vdash e'[x := v], f : d \cdot a'$$

$a' = (a_{o \cdot A}[x := v]; a_f)$

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

$$P; (E, E'[x := v]) \vdash (e' \cdot f)[x := v] : d \cdot (a_o; a_f)[x := v]$$

Hence we obtain our desired result

$$P; (E, E'[x := v]) ; (e' \cdot f)[x := v] : d \cdot a[x := v]$$

- $e \equiv e' \cdot f = e''$. Then our assumption is

$$P; (E, e, c, E') \vdash e' \cdot f = e'' : d \cdot a$$

This assumption is derivable only through rules **[exp assign]** and **[exp assign race]**; hence we know

$$P; (E, e, c, E') \vdash e' : d \cdot a_o$$

$P; (E, e, c, E') \vdash e'' : d \cdot a_v$

$\text{class } d \{ \ldots d f \ldots \} \in P$

$a = (a_o; a_v; a_f)$

where $a_f$ is one of three cases

- **AM** if $f \in \text{Normal}$ and $\gamma = \ldots$
- **AN** if $f \in \text{Volatile}$ and $\gamma = \ldots$
- **CL** if $f \in \text{Volatile}$ and $\gamma = \ldots$

By **IH**, we have

$$P; (E, E'[x := v]) \vdash e'[x := v] : d \cdot a_{o \cdot a}[x := v]$$

$$P; (E, E'[x := v]) \vdash e''[x := v] : d \cdot a_{v \cdot a}[x := v]$$

By rule **[exp assign]**/[**exp assign race**], we may conclude

$$P; (E, E'[x := v]) \vdash e'[x := v], f : e''[x := v], d \cdot a'$$

$a' = (a_o[x := v]; a_v[x := v]; a_f[x := v])$

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

$$P; (E, E'[x := v]) \vdash (e' \cdot f)[x := v] : d \cdot (a_o; a_v; a_f)[x := v]$$

Hence we obtain our desired result

$$P; (E, E'[x := v]) ; (e' \cdot f)[x := v] : d \cdot a[x := v]$$

- $e \equiv e' \cdot f \equiv e''$. Then our assumption is

$$P; (E, e, c, E') \vdash e' \cdot f = e'' : d \cdot a$$

This assumption is derivable only through rule **[exp assign]**; hence we know

$$\text{class } d \{ \ldots d x_i \in \{1 ... n\} \} \in P$$

$P; (E, e, c, E') \vdash a \cdot d_i \cdot a_i$

$$\text{lock } \ell \\in \ldots a_o[x := v]; a_v[x := v]; a_f[x := v]$$

Since we have the following equalities,

- **AM** if $f \in \text{Normal}$ and $\gamma = \ldots$
- **AF** if $f \in \text{Final}$ and $\gamma = \ldots$
- **AN** if $f \in \text{Volatile}$ and $\gamma = \ldots$
- **CL** if $f \in \text{Volatile}$ and $\gamma = \ldots$

By **IH**, we have

$$P; (E, E'[x := v]) \vdash e'[x := v] : d \cdot a_i[x := v]$$

By rule **[exp new]**, we may conclude

$$P; (E, E'[x := v]) \vdash \text{new } d (c_i[x := v] \in \{1 ... n\}) : d \cdot a'$$

$a' = (a_1[x := v]; \ldots; a_o[x := v]; a_v[x := v]; a_f[x := v])$

Hence we obtain our desired result

$$P; (E, E'[x := v]) ; (e' \cdot f)[x := v] : d \cdot a[x := v]$$

- $e \equiv e' \cdot f \equiv e''$. Then our assumption is

$$P; (E, e, c, E') \vdash e' \cdot f = e'' : d \cdot a$$

This assumption is derivable only through rule **[exp invoke]**; hence we know

$$P; (E, e, c, E') \vdash \ell \cdot \text{sync } e' : d \cdot a$$

By **IH**, we have

$$P; (E, E'[x := v]) \vdash e'[x := v] : d \cdot a_o[x := v]$$

By rule **[exp sync]**, we may conclude

$$P; (E, E'[x := v]) \vdash e'[x := v] : d \cdot a_o[x := v]$$

By **IH**, we have

$$P; (E, E'[x := v]) \vdash e'[x := v] : d \cdot a_o[x := v]$$

By rule **[exp sync]**, we may conclude

$$P; (E, E'[x := v]) \vdash e'[x := v] : d \cdot a_o[x := v]$$

We proceed by case analysis on the structure of effect $a_o$. 2011/8/12
− \( a_b = b \). In this case, since \( b' = b'[x := v] \) for any combined (basic) effect, including \( a_b, \rho, \) and \( AL \), we have

\[
\begin{align*}
  a' &= S(\ell [x := v], \gamma, a_b [x := v]) \\
  &= \ell [x := v] ? a_1 [x := v] : (\gamma; \ar; a_0 [x := v] : \al) \\
  &= (\ell' ? \alpha_1 : (\gamma; \ar; a_0; \al)) [x := v] \\
  &= a [x := v]
\end{align*}
\]

− \( a_b = \ell' ? a_1 : a_2 \). Then applying the substitution, we get

\[
\]

Proceeding with \( \ell' \),

\[
\begin{align*}
  a' &= S(\ell [x := v], \gamma, a_b [x := v]) \\
  &= S(\ell [x := v], \gamma, \ell [x := v] ? a_1 [x := v] : a_2 [x := v]) \\
  &= S(\ell [x := v], \gamma, a_1 [x := v]) \\
  &= S(\ell, \gamma, a_0 [x := v]) \\
  &= a [x := v]
\end{align*}
\]

− \( a_b = \ell' ? a_1 : a_2 \), where \( \ell' \neq \ell \). Then applying the substitution, we get

\[
a_0 [x := v] = \ell' [x := v] ? a_1 [x := v] : a_2 [x := v]
\]

Let \( \theta (E) = E [x := v] \), for any expression or effect \( E \).

Proceeding with \( \ell' \),

\[
\begin{align*}
  a' &= S(\theta (\ell), \gamma, \theta (a_0)) \\
  &= S(\theta (\ell), \gamma, \theta (\ell' ? a_1 : a_2)) \\
  &= S(\theta (\ell), \gamma, a_1) : S(\theta (\ell), \gamma, a_2) \\
  &= S(\theta (\ell), \gamma, a_1) : S(\ell, \gamma, a_2) \\
  &= S(\ell, \gamma, a_0 [x := v]) \\
  &= a [x := v]
\end{align*}
\]

Hence we obtain our desired result

\[
P; (E, E' [x := v]) \vdash e [x := v] : d \cdot a [x := v]
\]

\[\bullet \ e \equiv \text{fork } e'. \] Then our assumption is

\[
P; (E, e x, E') \vdash \text{fork } e' : d \cdot a
\]

This assumption is derivable only through rule \([\exp \text{fork}]\); hence we know

\[
P; (E, e x, E') \vdash e' : d \cdot a
\]

By IH, we have

\[
P; (E, E' [x := v]) \vdash \text{lock } \rho [x := v]
\]

Then by rule \([\exp \text{fork}]\) we have

\[
P; (E, E' [x := v]) \vdash e' [x := v] : d \cdot a' \\
  a' = \al
\]

Since \( AL = \al [x := v] \) and \( \text{fork } e' [x := v] = (\text{fork } e') [x := v] \), we obtain our desired result

\[
P; (E, E' [x := v]) \vdash e [x := v] : d \cdot a [x := v]
\]

\[\bullet \ e \equiv \text{let } x_1 e_1 \text{ in } e_2. \] Then our assumption is

\[
P; (E, e x, E') \vdash \text{let } x_1 = e_1 \text{ in } e_2 : d \cdot a
\]

This assumption is derivable only through rule \([\exp \text{let}]\); hence we know

\[
P; (E, e x, E') \vdash e_1 : c_1 \cdot a_1 \\
  P; (E, e x, E') \vdash e_2 : d \cdot a_2 \\
  P; (E, e x, E') \vdash a_2 [x := e_1] \uparrow a_2' \\
  a_2' = (a_1, a_2')
\]

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
\[ P; (E, E'[x := v]) \vdash e_1[x := v] : c_1 \cdot a_1 \] for \( x \neq x_1 \).

By rule \( \text{EXP LET} \), we may conclude
\[ P; (E, E'[x := v]) \vdash (a_2[x := v]) \mid x_1 := e_1[x := v] \uparrow a_2'[x := v] \]

In the last judgment, since \( x \neq x_1 \), we have
\[ P; (E, E'[x := v]) \vdash (a_2[x := v]) \mid x_1 := e_1[x := v] \uparrow a_2'[x := v] \]

By rule \( \text{EXP LET} \), we may conclude
\[ P; (E, E'[x := v]) \vdash \text{let } x_1 = (e_1[x := v]) \text{ in } (e_2[x := v]) : d \cdot a' \]
\[ a' = (a_1[x := v] ; a_2'[x := v]) = a[x := v] \]

Hence we obtain our desired result
\[ P; (E, E'[x := v]) \vdash e[x := v] : d \cdot a[x := v] \]

- \( e \equiv e_1 \cdot e_2 \cdot e_3 \). Then our assumption is
\[ P; (E, c \cdot x, E') \vdash e_1[d \cdot a_1] \]

This assumption is derivable only through rule \( \text{EXP IF} \); hence we know
\[ P; (E, c \cdot x, E') \vdash e_1 \cdot a_1 \]
\[ P; (E, c \cdot x, E') \vdash e_2 \cdot a_2 \\
\[ a = (a_1 ; (a_2 \sqcup a_3)) \]

By IH, we have
\[ P; (E, E'[x := v]) \vdash e_1[x := v] : d \cdot a_1[x := v] \]
\[ P; (E, E'[x := v]) \vdash e_2[x := v] : d \cdot a_2[x := v] \]
\[ P; (E, E'[x := v]) \vdash e_3[x := v] : d \cdot a_3[x := v] \]

By rule \( \text{EXP IF} \), we may conclude
\[ P; (E, E'[x := v]) \vdash \text{if } e_1[x := v] \text{ then } e_2[x := v] \text{ else } e_3[x := v] : d \cdot a' \]
\[ a' = (a_1[x := v] ; (a_2[x := v] \sqcup a_3[x := v])) \]

Since the following equalities hold
\[ (\text{if } e_1[x := v] ; e_2[x := v] ; e_3[x := v]) = (e_1[x := v]) = e[x := v] \]
\[ a' = (a_1[x := v] ; (a_2[x := v] \sqcup a_3[x := v])) = (a_1 ; (a_2 \sqcup a_3)) = a[x := v] \]

by substituting, we obtain our desired result
\[ P; (E, E'[x := v]) \vdash e[x := v] : a[x := v] \]

- \( e \equiv \text{while } e_1 \text{ do } a_2 \). Then our assumption is
\[ P; (E, c \cdot x, E') \vdash \text{while } e_1 \text{ do } a_2[d \cdot a] \]

This assumption is derivable only through rule \( \text{EXP WHILE} \); hence we know
\[ P; (E, c \cdot x, E') \vdash e_1 \cdot a_1 \\
\[ P; (E, c \cdot x, E') \vdash e_2 \cdot a_2 \\
\[ a = (a_1 ; (a_2 ; a_3)) \]

By IH, we have
\[ P; (E, E'[x := v]) \vdash e_1[x := v] : c_1 \cdot a_1[x := v] \]
\[ P; (E, E'[x := v]) \vdash e_2[x := v] : c_2 \cdot a_2[x := v] \]

By rule \( \text{EXP WHILE} \), we may conclude
\[ P; (E, E'[x := v]) \vdash \text{while } e_1[x := v] \text{ do } a_2[x := v] : d \cdot a' \]
\[ a' = (a_1[x := v] ; (a_2[x := v] ; a_1[x := v])) \]

Since the following equalities hold
\[ \text{while } e_1[x := v] \text{ do } e_2[x := v] = (\text{while } e_1 \text{ do } e_2)[x := v] = e[x := v] \]
\[ a' = (a_1[x := v] ; (a_2[x := v] ; a_1[x := v])) \]
\[ (a_1 ; (a_2 ; a_3))[x := v] = a[x := v] \]

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:

- \( \text{EXP VAR} \): \( e \equiv x \)
- \( \text{EXP NULL} \): \( e \equiv \text{null} \)
- \( \text{EXP ADDR} \): \( e \equiv \rho \)
- \( \text{EXP REF} \): \( e \equiv e'.\rho \)
- \( \text{EXP REF RACE} \): \( e \equiv e'.\rho \)
- \( \text{EXP ASSIGN} \): \( e \equiv e'.\rho \)
- \( \text{EXP Assign RACE} \): \( e \equiv e'.\rho = e'' \)
- \( \text{EXP NEW} \): \( e \equiv \text{new } c \) \( e_1, \ldots, e_n \)
- \( \text{EXP FORK} \): \( e \equiv \text{fork } c \) \( e_1, \ldots, e_n \)
- \( \text{EXP IF} \): \( e \equiv \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \)
- \( \text{EXP WHILE} \): \( e \equiv \text{while } e_1 \text{ do } a \)

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 \( \text{AT BASE} \) applies, and we may conclude
\[ P; E \vdash b \]

- \( a = \ell \cdot a_1 \cdot a_2 \). In this case, all lock expressions contained in \( a \), including \( \ell \), are constant, by IH. Sequential composition, iterative closure, and join operations do not change or add lock expressions. Hence, by rule \( \text{AT COND} \), we may conclude
\[ P; E \vdash a \]

There are five remaining cases to consider.

- \( \text{EXP SYNC} \) As an assumption, we have
\[ P; E \vdash \ell \cdot \text{sync } e' : c \cdot a \]

This assumption may be derived only through rule \( \text{EXP SYNC} \); hence we know
\[ P; E \vdash \ell \cdot \text{lock } \ell \]
\[ P; E \vdash e' : c \cdot a' \]

By IH, we have \( P; E \vdash a' \). We proceed by case analysis on the structure of \( a' \).

- \( a' = b \). In this case, by the first case of function \( S \), we know
\[ a = \ell \cdot a' = (\{\ell\} ; \text{AR}; a' ; \text{AL}) \]

Since \( \ell \), the only lock expression in \( a \), is constant, and since both branches in \( a \) are well-typed effects, by rule \( \text{AT COND} \) we may conclude
\[ P; E \vdash a \]
• $a' = \ell? a_1 : a_2$. In this case, by the second case of function $S$, we know
  
  $a = S(\ell, \gamma, a_1)$
  
  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, $\ell$ is a constant lock expression. Thus any lock expression in $S(\ell, \gamma, a_1)$ 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 $S$, we know
  
  $a = \ell'? S(\ell, \gamma, a_1) : S(\ell, \gamma, a_2)$
  
  Since by IH $a'$ is a well-typed effect, we know that $\ell'$ is constant, and that $a_1$ and $a_2$ are well-typed. Thus any lock expression in $S(\ell, \gamma, a_1)$ and $S(\ell, \gamma, a_2)$ is constant. By rule [AT COND], we may conclude
  
  $P; E \vdash a$
  
  - [EXP INSYNC] As an assumption, we have
    
    $P; E \vdash \text{in-sync } \rho' \cdot c : c \cdot a$
    
    This assumption may be derived only through rule [EXP INSYNC]; hence we know
    
    $P; E \vdash \text{lock } \rho$
    
    $P; E \vdash e' : c \cdot a'$
    
    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 $SL$, 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 $SL$, we know
        
        $a = SL(\rho, a_1)$
        
        Since $a'$ is well-typed, inductively $a_1$ is also well-typed. Thus we know any lock expression contained in $SL(\rho, a_1)$ is constant. Thus by rule [AT COND], we may conclude
        
        $P; E \vdash a$
        
        If $SL(\rho, a_1)$ 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 $SL$, we know
          
          $a = \rho' ? SL(\rho, a_1) : SL(\rho, a_2)$
          
          By IH, $a'$ is well-typed; hence inductively $\rho'$ is a constant lock expression and $a_1$ and $a_2$ are well-typed. Since $SL$ may at most introduce $\rho'$ 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
            
            $P; E \vdash e' \cdot m(e_1..n) : c \cdot a$
            
            This assumption is derivable only through rule [EXP INVOKE]; hence we know
            
            $P; E \vdash e' : c' \cdot a'$
            
            We proceed by case analysis on the structure of effect $a'$.
            
            - $a' = c'$, a well-typed effect, we know that $a_1$ is also well-typed. Hence all locks in $a_1$ are constant. Furthermore, $\ell$ is a constant lock expression. Thus any lock expression in $S(\ell, \gamma, a_1)$ is constant. By rule [AT COND], we may conclude
              
              $P; E \vdash a$
              
              $\ell, \gamma, a \cdot \rho, \text{meth } E \cdot c \cdot a \cdot i \cdot \text{this } := e'.x_1 := e_i \cdot \ell' \cdot \text{this } := e' \cdot .x := e_i \cdot i \cdot \ell \cdot \text{this } := e' \cdot .x := e_i \cdot i \cdot \ell \cdot \text{this } := e' \cdot .x := e_i \cdot a''$
              
              By IH, we have
              
              $P; E \vdash a'$
              
              $P; E \vdash a_i \cdot \forall i \in 1..n$
              
              $P; E \vdash a''$
              
              In particular, for the last judgment, if any lock expression $\ell'$ in $a''[\text{this } := e', x_1 := e_i \cdot \ell' \cdot \text{this } := e' \cdot .x := e_i \cdot i \cdot \ell \cdot \text{this } := e' \cdot .x := e_i \cdot a'']$ is not constant, the lift will eliminate $\ell'$. Then by rule [EXP INVOKE], we have
              
              $a = (a'; \ldots ; a_n); [\gamma]; a''$
              
              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
                
                $P; E \vdash e' ; \gamma_2 .m(\gamma_1..n) : c \cdot a$
                
                The argument proceeds in a similar manner to the previous case.
                
                - [EXP LET] As an assumption, we have
                  
                  $P; E \vdash \text{let } x = e' \cdot \text{in } e'' : c \cdot a$
                  
                  This assumption is derivable only through rule [EXP LET]; hence we know
                  
                  $P; E \vdash e' : c' \cdot a'$
                  
                  $P; (E, e' \cdot x) \vdash e'' : c' \cdot a''$
                  
                  $P; E \vdash a''[x := e'] \cdot a''$
                  
                  By IH, we know
                  
                  $P; E \vdash a'$
                  
                  $P; E \vdash a''$
                  
                  In particular, for the last judgment, if any lock expression $\ell'$ in $a''[x := e']$ is not constant, the lift will eliminate $\ell'$. 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
                  
                  $P; E \vdash a$
• $a = \ell? a_1 : a_2$. We proceed by case analysis on the constancy of $\ell$.
  - $P; E \vdash_{\text{lock}} \ell$. Then $P; E \vdash a \uparrow a'$ is only derivable with rule \[\text{LIFT GOOD LOCK}\]; hence we know
    
    $P; E \vdash a_1 \uparrow a'_1$
    
    $P; E \vdash a_2 \uparrow a'_2$
    
    $a' = (\ell? a_1 : a_2)$

    By IH, we have
    
    $P; E \vdash a'_1$
    
    $P; E \vdash a'_2$

    By rule \[\text{AT COND}\], we may conclude
    
    $P; E \vdash a'$

  - $P; E \nmid_{\text{lock}} \ell$. Then $P; E \vdash a \uparrow a'$ is only derivable with rule \[\text{LIFT BAD LOCK}\]; hence we know
    
    $P; E \vdash a_1 \uparrow a'_1$
    
    $P; E \vdash a_2 \uparrow a'_2$
    
    $a' = (a'_1 \sqcup a'_2)$

    By IH, we have
    
    $P; E \vdash a'_1$
    
    $P; E \vdash a'_2$

    Since all locks in $a'_1$ and $a'_2$ are constant, and joining does not change or add lock expressions, by rules \[\text{AT BASE}\] and \[\text{AT COND}\] we may conclude
    
    $P; E \vdash a'$

---

**Lemma 9** (Substitution with well-formed lock). If we have

$P; E, c \ x \vdash a$

$P; E \vdash \ell$

then we can conclude $P; E \vdash a[\ell := \ell'] \uparrow a[\ell := \ell]$.

**Proof** The judgment $P; E \vdash_{\text{lock}} \ell$ is derivable only through rule \[\text{LOCK EXP}\]; hence we know

$P; E \vdash \ell : c \cdot AF$

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 \[\text{AT BASE}\]; hence we know

$P; (E, c \ x) \vdash a$

We may further weaken the environment by rule \[\text{ENV X}\]:

$P \vdash E$

Then by rule \[\text{LIFT BASE}\] we may conclude

$P; E \vdash a \uparrow a$

Since $a$ does not mention $x$, we have the equality

$a = a[\ell := \ell']$

Substituting, we obtain our desired result

$P; E \vdash a[\ell := \ell'] \uparrow a[\ell := \ell]$

• $a = \ell'? a_1 : a_2$. In this case, our assumption of $P; (E, c \ x) \vdash a$ is derivable only through rule \[\text{AT COND}\]; hence we know

$P; (E, c \ x) \vdash_{\text{lock}} \ell'$

$P; (E, c \ x) \vdash a_1$

$P; (E, c \ x) \vdash a_2$

By IH, we have

$P; E \vdash a_1[\ell := \ell] \uparrow a_1[\ell := \ell]$

$P; E \vdash a_2[\ell := \ell] \uparrow a_2[\ell := \ell]$

By Lemma 6 and $P; (E, c \ x) \vdash_{\text{lock}} \ell'$, we may conclude

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

By rule \[\text{LIFT COND}\], we may conclude

$P; E \vdash a[\ell := \ell] \uparrow a[\ell := \ell]$

Simplifying, we obtain our desired result

$P; E \vdash a[\ell := \ell] \uparrow a[\ell := \ell]$

---

**Lemma 10** (While equivalence). If we have a judgment $P; E \vdash \text{while } e_1 e_2 : c \cdot a$ then we can also conclude $P; E \vdash \text{if } e_1 (e_2) \text{ null } a$

**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 \text{while} judgment assumption is derivable only through rule \[\text{EXP WHILE}\]; hence we know

$P; E \vdash e_1 : e_1 \cdot a_1$

$P; E \vdash e_2 : e_2 \cdot a_2$

$a = (a_1; (a_2; a_1)^*)$

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 \[\text{EXP IF}\], we may conclude

$P; E \vdash \text{if } e_1 (e_2) \text{ null } c \cdot a'$

$a' = a_1; ((a_2; a) \sqcup \text{AF})$

Expanding $a$ and distributing $a_2; a_1$ across the joins, we get the equality

$a' = a_1; (a_2; a_1; (\text{AF} \sqcup (a_2; a_1) \sqcup (a_2; a_1; a_2; a_1) \sqcup \ldots) \sqcup \text{AF})$

$= a_1; (\text{AF} \sqcup (a_2; a_1) \sqcup (a_2; a_1; a_2; a_1) \sqcup \ldots)$

$= a$

Thus we obtain our desired result.

---

**Lemma 11** (Lifting After Substitution). Suppose the following:

$P; E \vdash a[\ell := \ell'] \uparrow a'$

$c' \text{ not a value}$

Then we can conclude $a'' \subseteq a'$.

**Proof** By induction on structure of effect $a$.

• $a = b$. In this case, the following holds:

$a'' = (b[\ell := \ell']) = b = (b[\ell := \ell']) = a'$

Hence we may conclude

$a'' \subseteq a'$
• \(a = \ell \cdot a_1 : a_2\). 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:

\[
P; E \vdash a_1[x := e'] \uparrow a'_1
\]

and

\[
P; E \vdash a_2[x := e'] \uparrow a'_2
\]

By IH, we know

\[
a'_1 \sqsubseteq a'_2
da'_2 \sqsubseteq a_2
\]

We proceed by case analysis on whether \(x\) is free in \(\ell\).

• \(x\) is free in \(\ell\). In this case, \([\ell[x := e']\) is not a value, since \(e'\) is not a value. The lock judgment does not hold:

\[
P; E \not\vdash \ell[x := e']
\]

Hence, by rule [LIFT BAD LOCK] we get

\[
a' = a'_1 \cup a'_2
\]

Case analysis on lock judgment for \([\ell[x := e'']\):

- \(P; E \vdash \ell[x := e'']\): In this case,

\[
a'' = ([\ell[x := e'']] \cdot a''_1 : a''_2) \sqsubseteq \{a'_1 \cup a'_2\} = a'
\]

- \(P; E \not\vdash \ell[x := e'']\): In this case,

\[
a'' = \{a''_1 \cup a''_2\} \sqsubseteq \{a'_1 \cup a'_2\} = a'
\]

• \(x\) is not free in \(\ell\). In this case, the following holds:

\[
\ell[x := e'] = \ell = [\ell[x := e'']]
\]

Case analysis on lock judgment for \(\ell\):

- \(P; E \vdash \ell[x := e'']\): In this case, we may conclude

\[
a'' = ([\ell ? a''_1 : a''_2] \sqsubseteq ([\ell ? a'_1 : a'_2] = a'
\]

- \(P; E \not\vdash \ell[x := e'']\): In this case, we may conclude

\[
a'' = \{a'_1 \cup a'_2\} \sqsubseteq \{a''_1 \cup a''_2\} = a'
\]

Because both \(\sigma\) and \(\sigma'\) have equivalent domains, every address \(\rho\) has the same type \(d\) in both \(\sigma\) and \(\sigma'\). The second judgment we have, then, is equivalent to

\[
P; E_\alpha \vdash e \cdot c \cdot a'
\]

We conclude with our desired result:

\[
a = a'
\]

**Lemma 14** (Sequentiality for Sync). Assume for some state \(\sigma\) the following:

\[
P; E_\sigma \vdash \text{lock } \rho
\]

\[
P; E_\sigma \vdash a
\]

disjoint locksets \(h\) and \(n\)

\[
\rho \in n
\]

Then we may conclude

\[
[\gamma]; S_I(p, a) \sqsubseteq_h S(p, \gamma, a)
\]

**Proof** By induction over structure of effect \(a\).

• \(a = \kappa\). In this case,

\[
S_I(p, a) = \kappa; AL
\]

\[
S(p, \gamma, a) = \rho ? a : ([\gamma]; AR; \kappa; AL)
\]

With \(\rho \notin h\), we can show for any \(\kappa\):

\[
[\gamma]; \kappa; AL \sqsubseteq_h [\gamma]; AR; \kappa; AL
\]

Hence we obtain our desired result

\[
[\gamma]; S_I(p, a) \sqsubseteq_h S(p, \gamma, a)
\]

• \(a = \rho ? a_1 : a_2\). In this case,

\[
S_I(p, a) = S_I(p, a_1)
\]

\[
S(p, \gamma, a) = S(p, \gamma, a_1)
\]

By IH,

\[
[\gamma]; S_I(p, a_1) \sqsubseteq_h S(p, \gamma, a_1)
\]

Hence we obtain our desired result

\[
[\gamma]; S_I(p, a) \sqsubseteq_h S(p, \gamma, a)
\]

• \(a = \rho' ? a_1 : a_2\) where \(\rho' \neq \rho\). In this case,

\[
S_I(p, a) = \rho' ? S_I(p, a_1) : S(p, a_2)
\]

\[
S(p, \gamma, a) = \rho' ? S(p, \gamma, a_1) : S(p, \gamma, a_2)
\]

Assume that \(\rho' \notin n\). Then by IH,

\[
[\gamma]; S_I(p, a_1) \sqsubseteq_{h'} {\rho'} S(p, \gamma, a_1)
\]

Assume that \(\rho' \notin h\). Then by IH,

\[
[\gamma]; S_I(p, a_2) \sqsubseteq_{h_2} {\rho'} S(p, \gamma, a_2)
\]

Combining these, we may conclude

\[
\rho' ? ([\gamma]; S_I(p, a_1)) : ([\gamma]; S_I(p, a_2)) \sqsubseteq_h \rho' ? S(p, \gamma, a_1) : S(p, \gamma, a_2)
\]

By definition of conditional effects, we extract \(\rho'\):

\[
[\gamma]; \rho' ? S_I(p, a_1) : S_I(p, a_2) \sqsubseteq_h \rho' ? S(p, \gamma, a_1) : S(p, \gamma, a_2)
\]

Substituting, we obtain our desired result

\[
[\gamma]; S_I(p, a) \sqsubseteq_h S(p, \gamma, a)
\]
Then we may conclude
\[ a \subseteq_n h(U)(\rho) S(\rho, \gamma, a) \]

Proof By induction over structure of effect \(a\).

- \(a = \kappa\). In this case,
  \[ S(\rho, \gamma, a) = \rho ? \kappa : ([\gamma]; AR; \kappa; AL) \]

We can conclude the following
\[ \kappa \subseteq_{h(U)(\rho)} \kappa \]

only if the following are true:
\[ \rho \notin n \Rightarrow \kappa \subseteq_{h(U)(\rho)} \kappa \]
\[ \rho \notin h \cup \{ \rho \} \Rightarrow \kappa \subseteq_{h(U)(\rho)} ([\gamma]; AR; \kappa; AL) \]

The first statement is true since it relates two combined (basic) effects. The second statement is true vacuously. Hence we obtain our desired result
\[ a \subseteq_{h(U)(\rho)} S(\rho, \gamma, a) \]

- \(a = \rho ? a_1 : a_2\). In this case,
  \[ S(\rho, \gamma, a) = S(\rho, \gamma, \rho ? a_1 : a_2) = S(\rho, \gamma, a_1) \]

By IH, we have
\[ a_1 \subseteq_{h(U)(\rho)} S(\rho, \gamma, a_1) \]

We can conclude the following
\[ \rho ? a_1 : a_2 \subseteq_{h(U)(\rho)} S(\rho, \gamma, \rho ? a_1 : a_2) \]

only if the following are true:
\[ \rho \notin n \Rightarrow a_1 \subseteq_{h(U)(\rho)} S(\rho, \gamma, \rho ? a_1 : a_2) \]
\[ \rho \notin h \cup \{ \rho \} \Rightarrow a_2 \subseteq_{h(U)(\rho)} S(\rho, \gamma, \rho ? a_1 : a_2) \]

The first statement is true due to IH. The second statement is true vacuously. Hence we obtain our desired result
\[ a \subseteq_{h(U)(\rho)} S(\rho, \gamma, a) \]

- \(a = \rho' ? a_1 : a_2\), where \(\rho' \neq \rho\). In this case,
  \[ S(\rho, \gamma, a) = S(\rho, \gamma, \rho' ? a_1 : a_2) = \rho' ? S(\rho, \gamma, a_1) : S(\rho, \gamma, a_2) \]

We can conclude the following
\[ \rho' ? a_1 : a_2 \subseteq_{h(U)(\rho)} S(\rho, \gamma, \rho' ? a_1 : a_2) \]

only if the following are true:
\[ \rho' \notin n \Rightarrow a_1 \subseteq_{h(U)(\rho', \rho)} S(\rho, \gamma, a_1) \]
\[ \rho' \notin h \cup \{ \rho \} \Rightarrow a_2 \subseteq_{h(U)(\rho', \rho)} S(\rho, \gamma, a_2) \]

Assume \(\rho' \notin n\). Then by IH,
\[ a_1 \subseteq_{h(U)(\rho', \rho)} S(\rho, \gamma, a_1) \]

Assume \(\rho' \notin h \cup \{ \rho \}\). Then by IH,
\[ a_2 \subseteq_{h(U)(\rho', \rho)} S(\rho, \gamma, a_2) \]

Hence we obtain our desired result
\[ a \subseteq_{h(U)(\rho)} S(\rho, \gamma, a) \]
Hence we obtain our desired result
\[ Y(\mathcal{E}, a) \subseteq a \]

\( a = \rho ? a_1 : a_2 \). We proceed by case analysis on \( \rho \):

\( \rho \in \text{locks}(\mathcal{E}) \). In this case,
\[ Y(\mathcal{E}, \rho ? a_1 : a_2) = Y(\mathcal{E}, a_1) \]

By IH, we have
\[ Y(\mathcal{E}, a_1) \subseteq a_1 \]

Also, may conclude the following
\[ a_1 \subseteq \rho ? a_1 : a_2 \]

only if the following hold:
\[ \rho \not\in n \Rightarrow a_1 \sqsubseteq (\rho) a_1 \\
\rho \not\in h \Rightarrow a_1 \sqsubseteq (\rho) a_2 \]

The first statement is trivially true. The second statement is vacuously true, due to a contradiction: \( \rho \) cannot be in the not-held lockset, since we know \( \rho \) is held in \( \mathcal{E} \).

Hence we obtain our desired result
\[ Y(\mathcal{E}, a) \subseteq a \]

\( \rho \not\in \text{locks}(\mathcal{E}) \). 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
\[ Y(\mathcal{E}, a_1) \subseteq a_1 \]
\[ Y(\mathcal{E}, a_2) \subseteq a_2 \]

We may conclude the following:
\[ \rho ? Y(\mathcal{E}, a_1) : Y(\mathcal{E}, a_2) \subseteq \rho ? a_1 : a_2 \]

only if the following hold:
\[ \rho \not\in n \Rightarrow Y(\mathcal{E}, a_1) \sqsubseteq (\rho) a_1 \\
\rho \not\in h \Rightarrow Y(\mathcal{E}, a_2) \sqsubseteq (\rho) a_2 \]

Both these statements are true due to IH. Since \( \rho \not\in \text{locks}(\mathcal{E}) \), 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
\[ Y(\mathcal{E}, a) \subseteq a \]

**Lemma 18 (Lock Removal).** Let \( h \) and \( n \) be lock sets, and \( \rho \) a lock. Let \( a_1 \) and \( a_2 \) be effects. Assume the following:

\[- \]
\[ P; E \vdash \text{lock } \rho \\
P; E \vdash a_1 \\
P; E \vdash a_2 \\
\exists h, n. h \not\cap n \not\in \{\rho\} \\
a_1 \text{ and } a_2 \text{ are not conditional on } \rho \\
a_1 \sqsubseteq_{\mathcal{E}}^h (\rho) a_2 \]

Then we may conclude
\[ a_1 \sqsubseteq_{\mathcal{E}}^h a_2 \]

**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
\[ \kappa_1 \sqsubseteq_{\mathcal{E}}^h (\rho) \kappa_2 \]

By inversion on this derivation, we know
\[ \kappa_1 \sqsubseteq_{\mathcal{E}}^h \kappa_2 \]

Then by definition of effect ordering on conditionals, we may conclude
\[ \kappa_1 \sqsubseteq_{\mathcal{E}}^h \kappa_2 \]

\( (a_1, a_2) = (\kappa_1, \kappa_2) \). In this case, we assume
\[ \kappa \sqsubseteq_{\mathcal{E}}^h (\rho) \kappa \]

By inversion on this derivation, we know
\[ \kappa \sqsubseteq_{\mathcal{E}}^h (\rho) \kappa \]

By definition of effect ordering on conditionals, we may conclude
\[ \kappa \sqsubseteq_{\mathcal{E}}^h \kappa \]

\( \bullet a = \rho ? a_1 : a_2 \). In this case, we assume
\[ a = \kappa \]

By Lemma 1,
\[ \kappa \sqsubseteq_{\mathcal{E}}^h \kappa \]

By definition of \( \mathcal{SI} \),
\[ \kappa \sqsubseteq_{\mathcal{E}}^h \mathcal{SI}(\rho, \kappa) \]

We obtain our desired result
\[ a \sqsubseteq_{\mathcal{E}}^h \mathcal{SI}(\rho, a) \]

**Lemma 19** (Yet Another In-Sync Subeffect Relation). Let \( a \) be an effect, and \( \rho \) a lock. Assume the following:

\[- \]
\[ P; E \vdash \text{lock } \rho \\
P; E \vdash a \\
\exists h, n. h \not\cap n \not\in \{\rho\} \\
\]

Then we may conclude
\[ a; \mathcal{AL} \sqsubseteq_{\mathcal{E}}^h \mathcal{SI}(\rho, a) \]

**Proof** By induction over structure of effect \( a \).

\( a = \kappa \). In this case, by definition of \( \sqsubseteq_{\mathcal{E}}^h \),
\[ \kappa \sqsubseteq_{\mathcal{E}}^h \kappa \]

By Lemma 1,
\[ \kappa \sqsubseteq_{\mathcal{E}}^h \kappa \]

By definition of \( \mathcal{SI} \),
\[ \kappa \sqsubseteq_{\mathcal{E}}^h \mathcal{SI}(\rho, \kappa) \]

We obtain our desired result
\[ a; \mathcal{AL} \sqsubseteq_{\mathcal{E}}^h \mathcal{SI}(\rho, a) \]
\[ a = \rho ? a_1 : a_2. \] In this case, our conclusion, which is
\[
\rho ? a_1 : a_2; \text{AL} \sqsubseteq_h \SI (\rho, \rho ? a_1 : a_2)
\]
\[ \Leftrightarrow \rho ? (a_1 : \text{AL}) : (a_2 : \text{AL}) \sqsubseteq_h \SI (\rho, a_1) \]
can only be true if the antecedents are true, by inversion on the derivation:
\[
(\rho \notin n) \Rightarrow a_1 : \text{AL} \sqsubseteq_h \SI (\rho, a_1)
\]
\[
(\rho \notin h) \Rightarrow a_2 : \text{AL} \sqsubseteq_{n \cup \{\rho\}} \SI (\rho, a_1)
\]
The first statement is true by IH. The second statement is vacuously true, since \(\rho \in h\). Hence we may conclude our desired result
\[ a; \text{AL} \sqsubseteq_h \SI (\rho, a) \]
\[ a = \rho' ? a_1 : a_2 \text{ where } \rho' \neq \rho. \] In this case, our conclusion, which is
\[
\rho' ? a_1 : a_2; \text{AL} \sqsubseteq_h \SI (\rho, \rho' ? a_1 : a_2)
\]
\[ \Leftrightarrow \rho' ? (a_1 : \text{AL}) : (a_2 : \text{AL}) \sqsubseteq_h \rho' ? \SI (\rho, a_1) : \SI (\rho, a_2) \]
can only be true if the antecedents are true, by inversion on the derivation:
\[
(\rho' \notin n) \Rightarrow a_1 : \text{AL} \sqsubseteq_{n \cup \{\rho'\}} \SI (\rho, a_1)
\]
\[
(\rho' \notin h) \Rightarrow a_2 : \text{AL} \sqsubseteq_{n \cup \{\rho'\}} \SI (\rho, a_2)
\]
Both statements are true by IH. Hence we may conclude our desired result
\[ a; \text{AL} \sqsubseteq_h \SI (\rho, a) \]
LEMMA 20 (Subeffect Relation for In-Sync). Let $a_y$, $a$ and $a'$ be effects, and $\rho$ be a lock. Assume the following:

\[ P; E \vdash_{\text{lock}} \rho \]
\[ P; E \vdash a_y \quad \text{and} \quad P; E \vdash a \quad \text{and} \quad P; E \vdash a' \]
\[ \exists h, n . h \cap n \land \rho \in h \]
\[ a_y; a' \sqsubseteq^h a \]

Then we may conclude

\[ a_y; SI(\rho, a') \sqsubseteq^h SI(\rho, a) \]

**Proof** By induction over structure of effect $a'$.

- $a' = \kappa$. In this case, we have
  \[ a_y; \kappa \sqsubseteq^h a \]
  by assumption
  \[ \iff a_y; \kappa; AL \sqsubseteq^h a; AL \]
  by Lemma 1
  \[ \implies a_y; \kappa; AL \sqsubseteq^h SI(\rho, a) \]
  by Lemma 19
  \[ \iff a_y; SI(\rho, \kappa) \sqsubseteq^h SI(\rho, a) \]
  by definition of $SI$
  \[ \iff a_y; SI(\rho, a') \sqsubseteq^h SI(\rho, a) \]
  our conclusion

- $a' = \rho ? a_1 : a_2$.
  \[ a_y; (\rho ? a_1 : a_2) \sqsubseteq^h a \]
  by assumption
  \[ \iff \rho ? (a_y; a_1) : (a_y; a_1) \sqsubseteq^h a \]
  by sequential composition
  \[ \implies (\rho \notin n) \Rightarrow a_y; a_1 \sqsubseteq^h a \]
  by inversion on derivation
  \[ \iff a_y; SI(\rho, a_1) \sqsubseteq^h SI(\rho, a) \]
  by IH
  \[ \iff a_y; SI(\rho, a') \sqsubseteq^h SI(\rho, a) \]
  by definition of $SI$
  \[ \iff a_y; SI(\rho, a') \sqsubseteq^h SI(\rho, a) \]
  our conclusion

- $a' = \rho' ? a_1 : a_2$ where $\rho' \neq \rho$.
  \[ a_y; (\rho' ? a_1 : a_2) \sqsubseteq^h a \]
  by assumption
  \[ \iff \rho' ? (a_y; a_1) : (a_y; a_1) \sqsubseteq^h a \]
  by sequential composition
  \[ \implies (\rho' \notin n) \Rightarrow a_y; a_1 \sqsubseteq^h a \]
  by inversion on derivation
  \[ \iff (\rho' \notin h) \Rightarrow a_y; a_2 \sqsubseteq^h a \]
  by inversion on derivation
  \[ \iff (\rho' \notin n) \Rightarrow a_y; SI(\rho, a_1) \sqsubseteq^h SI(\rho, a) \]
  by IH
  \[ \iff (\rho' \notin h) \Rightarrow a_y; SI(\rho, a_2) \sqsubseteq^h SI(\rho, a) \]
  by definition of $\sqsubseteq^h$
  \[ \iff a_y; SI(\rho, \rho' ? a_1 : a_2) \sqsubseteq^h SI(\rho, a) \]
  by sequential composition
  \[ \iff a_y; SI(\rho, a') \sqsubseteq^h SI(\rho, a) \]
  by definition of $SI$
  \[ \iff a_y; SI(\rho, a') \sqsubseteq^h SI(\rho, a) \]
  our conclusion
Lemma 21 (CS Context Subexpression). Suppose there is a deduction that concludes $ls \vdash_{cs} E[e]$. Then that deduction contains, at a position corresponding to the hole in $E$, a subdeduction that concludes $ls' \vdash_{cs} e$, where $ls' \subseteq ls$.

Proof. Induction over the derivation of $ls \vdash_{cs} E[e]$.

Lemma 22 (CS Context Substitution). Suppose the following:

- $ls \vdash_{cs} E[e]$, and
- $ls_e \vdash_{cs} e'$, and
- $ls'_e \vdash_{cs} e'$, and
- $ls'_e \not\vdash (ls \setminus ls_e)$

Then $ls' \vdash_{cs} E[e']$ and $ls' = (ls \setminus ls_e) \cup ls'_e$.

Proof. By induction over the structure of $E$.

- $E \equiv []$: In this case, we assume
  
  $$
  \begin{align*}
  ls & \vdash_{cs} [e] \\
  ls_e & \vdash_{cs} e \\
  ls'_e & \vdash_{cs} e' \\
  ls'_e & \not\vdash (ls \setminus ls_e)
  \end{align*}
  $$

  and may immediately conclude our desired result

  $$
  \begin{align*}
  ls' & \vdash_{cs} [e'] \\
  ls' & = ls'_e = (ls \setminus ls_e) \cup ls'_e
  \end{align*}
  $$

- $E \equiv new c(\tau, E', \tau)$: In this case, we assume
  
  $$
  \begin{align*}
  ls & \vdash_{cs} new c(\tau, E'[e], \tau) \\
  ls_e & \vdash_{cs} e \\
  ls'_e & \vdash_{cs} e' \\
  ls'_e & \not\vdash (ls \setminus ls_e)
  \end{align*}
  $$

  In the first assumption, this judgment may be concluded only by rule [CS NOT IN-SYNC]; hence we know

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

  By IH, we may conclude

  $$
  \begin{align*}
  ls' & \vdash_{cs} E'[e'] \\
  ls' & = (ls \setminus ls_e) \cup ls'_e
  \end{align*}
  $$

  Since in-sync $\not\subseteq$ new $c(\tau, [], \tau)$, by rule [CS NOT IN-SYNC] we obtain our desired result

  $$
  \begin{align*}
  ls' & \vdash_{cs} new c(\tau, E'[e'], \tau) \\
  ls' & = (ls \setminus ls_e) \cup ls'_e
  \end{align*}
  $$

- $E \equiv E' \gamma f$: Similar to previous argument.
- $E \equiv E' \gamma f = e'$: Similar to previous argument.
- $E \equiv \gamma v \gamma f = E'$: Similar to previous argument.
- $E \equiv E' \gamma m(\tau)$: Similar to previous argument.
- $E \equiv E' \gamma m#(\tau)$: Similar to previous argument.
- $E \equiv \rho, m(\tau, E', \tau)$: Similar to previous argument.
- $E \equiv \rho, m#(\tau, E', \tau)$: Similar to previous argument.
- $E \equiv \text{let } x = E' \text{ in } e'$: Similar to previous argument.
- $E \equiv \text{if } E' \text{ e1 e2}$: Similar to previous argument.
- $E \equiv E' \gamma \text{ sync } e'$: Similar to previous argument.
- $E \equiv \text{in-sync } \rho E'$: In this case, we assume
  
  $$
  \begin{align*}
  ls & \vdash_{cs} \text{in-sync } \rho E'[e] \\
  ls_e & \vdash_{cs} e \\
  ls'_e & \vdash_{cs} e' \\
  ls'_e & \not\vdash (ls \setminus ls_e)
  \end{align*}
  $$

  In the first assumption, this judgment may be concluded only by rule [CS IN-SYNC]; hence we know

  $$
  ls \setminus \{ \rho \} \vdash_{cs} E'[e]
  $$

From this judgment and Lemma 21, we know that $\rho \not\subseteq ls_e$. By IH, we may conclude

$$
\begin{align*}
ls' & \vdash_{cs} E'[e'] \\
ls' & = (ls \setminus \{ \rho \} \setminus ls_e) \cup ls'_e
\end{align*}
$$

Since $\rho \not\subseteq ls'$, by rule [CS IN-SYNC] we obtain our desired result

$$
\begin{align*}
ls' \cup \{ \rho \} & \vdash_{cs} \text{in-sync } \rho E'[e'] \\
ls' \cup \{ \rho \} & = (ls \setminus \{ \rho \} \setminus ls_e) \cup ls'_e \cup \{ \rho \} \\
& = (ls \setminus ls_e) \cup ls'_e
\end{align*}
$$