Featherweight X10: A Core Calculus for Async-Finish Parallelism

Jonathan K. Lee Jens Palsberg
UCLA, University of California, Los Angeles
{jkenl,palsberg}@cs.ucla.edu

Abstract

We present a core calculus with two of X10's key constructs for parallelism, namely async and finish. Our calculus forms a convenient basis for type systems and static analyses for languages with async-finish parallelism, and for tractable proofs of correctness. For example, we give a short proof of the deadlock-freedom theorem of Saraswat and Jagadeesan. Our main contribution is a type system that solves the open problem of context-sensitive may-happen-in-parallel analysis for languages with async-finish parallelism. We prove the correctness of our type system and we report experimental results of performing type inference on 13,000 lines of X10 code. Our analysis runs in polynomial time, takes a total of 28 seconds on our benchmarks, and produces a low number of false positives, which suggests that our analysis is a good basis for other analyses such as race detectors.

Categories and Subject Descriptors D.3 Programming Languages [*Formal Definitions and Theory*]

General Terms Algorithms, Languages, Theory, Verification

Keywords parallelism, operational semantics, static analysis

1. Introduction

Two of X10's [5] key constructs for parallelism are async and finish. The async statement is a lightweight notation for spawning threads, while a finish statement `finish` s waits for termination of all async statement bodies started while executing s.

Our goal is to enable researchers to easily define type systems and static analyses for languages with async-finish parallelism, and prove their correctness. For that purpose we provide a Turing-complete calculus with a minimal syntax and a simple formal semantics. A program in our calculus consists of a collection of methods that all have access to an array. The body of a method is a statement that can be skip, assignment, sequence, while loop, async, finish, or method call. If we add some boilerplate syntax to a program in our calculus, the result is an executable X10 program.

We call our calculus Featherweight X10, abbreviated FX10. Featherweight X10 shares a key objective with Featherweight Java [8], namely to enable a fundamental theorem to have a proof that is concise, while still capturing the essence of the proof for the full language. Our hope is that other researchers will find it easy to work either with FX10 as it is or with small extensions that meet particular needs.

We demonstrate the usefulness of our calculus in two ways. First, we give a short proof of the deadlock-freedom theorem of Saraswat and Jagadeesan [17]. They considered a much larger subset of X10 but stated the deadlock-freedom theorem without proof. Second, we present a type system that solves the open problem of context-sensitive may-happen-in-parallel analysis for languages with async-finish parallelism. We prove the type system correct and then discuss our experience with type inference.

The goal of may-happen-in-parallel analysis is to identify pairs of statements that may happen in parallel during some execution of the program. May-happen-in-parallel analysis is also known as *pairwise reachability* [9]. While the problem is undecidable in general and NP-complete under certain assumptions [18], a static analysis that gives an approximate answer is useful as a basis for tools such as data race detectors [6]. Researchers have defined may-happen-in-parallel analysis for Ada [7, 13, 15, 16], Java [12, 3], X10 [2], and other languages. Those seven papers specify polynomial-time analyses using pseudo-code, data flow equations or set constraints, but they give no proofs of correctness with respect to a formal semantics. Additionally, the algorithms are either intraprocedural, rely on inlining of method calls before the analysis begins, or treat call sites in a context-insensitive fashion, that is, merge the information from different call sites.

We believe that when a program happen may execute two statements in parallel, it should be because the programmer *intended* it. Thus, may-happen-in-parallel information should be something the programmer has in mind while programming, rather than something discovered after the programming is done. The data flow equations and set constraints used in previous work are great for specifying what an analysis does, but are much less helpful for a working programmer. We will use a type system to specify a may-happen-in-parallel analysis that comes with all the advantages of type systems: syntax-directed type rules and a well-understood approach to proving correctness [20]. In our case, we also get a straightforward way to do modular, context-sensitive analysis of methods, that is, a way to analyze each method just once and avoid merging information from different call sites for the same method. The advantage of syntax-directed type rules is that each rule concentrates on just one form of statement, and explains using only local information why the may-happen-in-parallel information for that statement is the way it is.

The paper by Agarwal et al. [2] on an intraprocedural may-happen-in-parallel analysis for X10 first determines what cannot happen in parallel and then takes its complement. In contrast, our type system defines a modular, interprocedural may-happen-in-parallel analysis without use of double negation. Additionally, our analysis comes with a proof of correctness plus experiments.

PPoPP'10, January 9–14, 2010, Bangalore, India.

Naik and Aiken [14] presented a flow- and context-sensitive may-happen-in-parallel analysis for Java as part of a static race detector. Their problem differs from ours because Java has no construct like finish.

Previous approaches to interprocedural analysis of concurrent programs include the paper by Barik and Sarkar [4] on X10, and the paper by von Praun and Gross [19] on Java; both present analyses that differ from may-happen-in-parallel analysis. The paper by Barik and Sarkar mentions that a refinement of their analysis with may-happen-in-parallel information is left for future work.

For a program p, let $\mathsf{MHP}(p)$ be the true may-happen-in-parallel information. Intuitively, if an execution of p can reach a state in which two statements with labels l_1 and l_2 can both happen next, then $(l_1, l_2) \in \mathsf{MHP}(p)$, and only such pairs are members of $\mathsf{MHP}(p)$. We will show how to compute a conservative yet precise approximation of $\mathsf{MHP}(p)$. In our case, a conservative approximation is a superset of $\mathsf{MHP}(p)$.

Our approach is to assign a type E to a program; every program has a type. Intuitively, E is a method summary for each method in the program. Each summary is a pair (M, O), where M is may-happen-in-parallel information and O is a helper set that we explain in a later section. Our correctness theorem (Theorem 3) says that if p has type E, and $E(f_0) = (M, O)$, where f_0 is the name of the main method, then

$$\mathsf{MHP}(p) \subseteq M$$

In other words, M is a conservative approximation of $\mathsf{MHP}(p)$.

If E is given, then we can do type *checking*. In practice, we want to compute E from p, that is, we want to do type *inference*, without any annotations or other help at all. For type inference we use the following approach. From p we generate a family of set constraints $C(p)$, and then we solve $C(p)$ using a polynomial-time algorithm that resembles the algorithms used for iterative data flow analysis. We prove the equivalence result (Theorem 4) that the solutions to $C(p)$ coincides with the types of p.

The slogan of the overall approach could be: *the type system leads to syntax-directed type rules and a proof of correctness, the constraints lead to a polynomial time algorithm, and the type system and the constraints are equivalent*. Our use of types gives a high-level specification of the analysis, while the use of constraints for us is an implementation technique.

In the following section we give two examples of skeletons of programs in our core language, along with discussions of how our analysis works. In Section 3 we present our core calculus, in Section 4 we show our type system, and in Section 5 we show how to generate and solve constraints. Finally, in Section 6 we discuss our experimental results for 13,000 lines of X10 code, and in Section 7 we conclude. Three appendices give detailed proofs of our theorems.

2. Examples

In this section we will give a taste of how our may-happen-in-parallel analysis works, and what results it can produce.

Let us first outline the main challenges for may-happen-in-parallel analysis for async-finish parallelism. The key problems stem from async, finish, loops and method calls. An async statement allows the body to run in parallel with any statement that follows it. If the body of a finish statement executes an async (or a method call that executes an async), then only when the async completes execution will the finish statement itself complete execution. This means that any statement in the body of a finish statement cannot run in parallel with anything that happens after the finish statement. A loop requires determining which async statements may occur in the body and recognizing that the body of the loop may run in parallel with those statements. Any bodies of async statements that are executing at the time of a method call may run in parallel with anything that may be executed in the method body.

2.1 First Example: Intraprocedural Analysis

The first example is from a PPoPP 2007 paper by Agarwal et al. [2, Figure 4], with some minor changes.

```
void main() {
   S0: finish {
         S1: async {
               S13: finish {
                      S5: ...
                      S6: async S11
                      S7: async S12
                    }
               S8: ...
             }
         S2: ...
       }
   S3: ...
}
```

From this program, we generate set constraints. Each set constraint is an equality of a set variable and a set expression, where the set expression may use set union. In a later section, we will show the constraints in detail (Figure 5), and explain how we generate and solve them.

The output from our constraint solver says correctly that `S2` may happen in parallel with each of `S5`, `S6`, `S7`, `S8`, `S11`, and `S12`, as well as with the entire `finish` statement. This is correct because the async statement `S1` has the statement `S2` occurring after it, so the entire body of the async may happen in parallel with S2.

The output also says that `S11` and `S12` may happen in parallel. This is correct because the two asyncs are not enclosed in separate finish statements and thus may be executing until the end of the enclosing finish. Furthermore, the output says that `S7` and `S11` may happen in parallel. This is correct because the body of an async may run in parallel with any statement that occurs after it.

The type inference algorithm found correctly that no other statements may happen in parallel. In particular, the inner finish statement ensures that `S11` and `S12` cannot run in parallel with the statements that follow the inner finish statement.

We conclude that for this program our algorithm determines the best possible may-happen-in-parallel information.

2.2 Second Example: Modular Interprocedural Analysis

The second example illustrates the modularity and context-sensitive aspects of our analysis.

```
void f() { async S5 }

void main() {
  S1: finish {
        async S3
        f()
      }
  S2: finish {
        f()
        async S4
      }
}
```

The output from our constraint solver says that `S5` may happen in parallel with each of `S3`, `async S4`, and `S4`, and that `S3` may also happen in parallel with the first call `f()` and with `async S5`. This is correct because the body of an async may run in parallel with any

statement that occurs after it, including after method boundaries. Here, S3 will run in parallel with the call f(), which in turn will execute an async with body S5. So, S3 may happen in parallel with f(), async S5, and S5. In the second finish, we have f() executing first which will allow S5 to run in parallel with async S4 and S4.

The type inference algorithm found correctly that no other statements may happen in parallel. In particular, S1 and S2 are finish statements that prevent the body of S1 to run in parallel with the body of S2, and our algorithm determines that S3 cannot happen in parallel with S4.

We conclude that also for this program our algorithm determines the best possible may-happen-in-parallel information.

Let us contrast the results from our analysis (Section 4) with the results from a context-insensitive analysis (Section 7) that merges information from different call sites. The context-insensitive analysis would say that S3 may happen in parallel with S4. The reason is that the context-insensitive analysis will conservatively merge (i) the information from the first call site that S3 may be executing when method f completes its execution with (ii) the information from the second call site that S4 runs after the call completes execution. The pair of S3 and S4 is an example of a false positive: the context-insensitive analysis infers that they may happen in parallel when in fact they cannot happen in parallel. In contrast, our analysis doesn't produce this particular false positive.

3. Featherweight X10

3.1 Design

FX10 is a core calculus in which sequential computation is the default, parallelism comes from the async statement, and synchronization comes from the finish statement.

A subset of X10. The language X10, version 1.5, is the starting point for the design of FX10. From X10 we take:

- a Turing-complete core consisting of while-loops, assignments, and a single one-dimensional integer array,
- methods and method calls, and
- the async and finish statements.

Both programs in Section 2 are FX10 programs, if we fill in the missing statements and ignore the labels of statements. We omit many features from X10, including places, distributions, and clocks.

Conventions and omitted boilerplate syntax. The grammar for FX10 uses $skip$ in place of the empty statement ";", and it specifies abstract syntax so it omits "{" and "}" for grouping of statements. The grammar for FX10 also omits some boilerplate syntax that is required to change an FX10 program into an executable X10 program. The boilerplate syntax consists of a main class plus one other class with a final field a that contains a one-dimensional integer array, a constructor, and then the methods from the FX10 program. For example, after we add the boilerplate syntax to the program in Section 2.2, it reads:

```
public class Main {
  public static void main(String[] args) {
    new C().main();
  }
}
class C {
  final int[:rank==1] a;
  public C() { ... }
  void f() { /* unchanged */ }
  void main() { /* unchanged */ }
}
```

The the constructor C() initializes the array variable a; for example, it might load the array's contents from a file.

One array. An FX10 computation works with a single shared memory given by an integer array variable named a. We chose to work with an array variable instead of a family of integer variables because of a subtlety in the X10 semantics of async. The body of an async statement can access variables outside the async statement only if those variables are declared *final*, that is, they can be initialized once but not updated later. We want to enable updates to variables, and therefore final integer variables are insufficient for our purposes. Instead we have a final integer array variable to which an array reference is assigned once, while the individual locations of the array can be updated and read multiple times.

Methods. FX10 contains methods and method calls to enable us to show our context-sensitive may-happen-in-parallel analysis. For studies in which methods play no particular role, researchers can easily remove methods from the language.

A method in FX10 has no arguments, no local variables, no return value, and no mechanism for early return. The reason is that the key problem for may-happen-in-parallel analysis stems from procedure calls themselves. A static analysis may be context insensitive (that is, merge the information from different call sites), or context sensitive (that is, separate the information from different call sites). As we will show in Section 7, for the case of may-happen-in-parallel analysis, the difference is significant.

Informal semantics. The semantics of FX10 uses the binary operator $\parallel$ in the semantics of async, it uses the binary operator $\triangleright$ in the semantics of finish, and it uses the constant $\surd$ to model a completed computation. A state in the semantics is a triple consisting of the program, the state of the array a, and a tree T that describes the code executing. The internal nodes of T are either $\parallel$ or $\triangleright$, while the leaves are either $\surd$ or $\langle s \rangle$, where s is a statement.

As an example of how the semantics works, we will now informally discuss an execution of the program in Section 2.2. Let us focus on the code that is being executed and let us ignore the state of the array a. The execution begins in main by executing the first finish statement.

$$\begin{aligned}
\langle \texttt{finish \{ async S3 f() \} S2} \rangle &\rightarrow \\
\langle \texttt{async S3 f()} \rangle \ \triangleright \ \langle \texttt{S2} \rangle &\rightarrow \\
(\langle \texttt{S3} \rangle \parallel \langle \texttt{f()} \rangle) \ \triangleright \ \langle \texttt{S2} \rangle &\rightarrow \\
(\langle \texttt{S3} \rangle \parallel \langle \texttt{async S5} \rangle) \ \triangleright \ \langle \texttt{S2} \rangle &\rightarrow \\
(\langle \texttt{S3} \rangle \parallel \langle \texttt{S5} \rangle) \ \triangleright \ \langle \texttt{S2} \rangle &
\end{aligned}$$

The first step illustrates the semantics of finish and introduces $\triangleright$ to signal that the left-hand side of $\triangleright$ must complete execution before the right-hand can proceed. The second step illustrates the semantics of async and introduces $\parallel$ to signal that S3 and f() should proceed in parallel. The third step illustrates the semantics of method call and replaces the call f() with the body async S5. The fourth step again illustrates the semantics of async. The two sides of $\parallel$ can execute in parallel, which we model with an interleaving semantics. When one of the sides completes execution, it will reach the state $\surd$. For example if S3 $\rightarrow \surd$, then the semantics can do (S3 $\parallel$ S5) $\rightarrow$ ($\surd \parallel$ S5) $\rightarrow$ S5. When also S5 completes execution, the semantics can finally proceed with the right-hand side of $\triangleright$.

3.2 Syntax

We use c to range over natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$, and we use l to range over labels. Figure 1 shows the grammar for the abstract syntax of FX10.

An FX10 program consists of a family of methods f_i, each with no arguments, return type void, and body s_i. We use $p(f_i)$ to denote s_i. Each s_i can access a nonempty one-dimensional array a with indices $0..n-1$, where $n > 0$. We use d to range over natural

$$
\begin{array}{lrcl}
\mathit{Program}: & p & ::= & \mathit{void}\ f_i()\ \{\ s_i\ \}, i \in 1..u \\
\mathit{Statement}: & s & ::= & \mathit{skip}^l \\
 & & \mid & i\ s \\
\mathit{Instruction}: & i & ::= & \mathit{skip}^l \\
 & & \mid & a[d] =^l e; \\
 & & \mid & \mathit{while}^l\ (a[d] \neq 0)\ s \\
 & & \mid & \mathit{async}^l\ s \\
 & & \mid & \mathit{finish}^l\ s \\
 & & \mid & f_i()^l \\
\mathit{Expression}: & e & ::= & c \\
 & & \mid & a[d] + 1
\end{array}
$$

Figure 1. The grammar of Featherweight X10.

numbers up to $n-1$: $\{0, 1, 2, \ldots, n-1\}$. When execution of the program begins, input values are loaded into all elements of the array a, and if the execution terminates, the result is in $a[0]$. Thus, the array a is fully initialized for all indices d when computation begins.

The body of each method is a statement. A statement is a sequence of labeled instructions. The labels have no impact on computation but are convenient for our may-happen-in-parallel analysis. Each instruction is either skip, assignment, while loop, async, finish, or method call.

The right-hand side of an assignment is an expression that can be either an integer constant or an array lookup plus one. An async statement $\texttt{async}^l\ s$ runs s in parallel with the continuation of the async statement. The async statement is a lightweight notation for spawning threads, while a finish statement $\texttt{finish}^l\ s$ waits for termination of all async bodies started while executing s.

It is straightforward to show that FX10 is Turing-complete, via a reduction from the while-programs of Kfoury et al. [10].

Compared to the core language for async-parallelism of Abadi and Plotkin [1], FX10 differs by having a finish statement and methods, while their language has constructs of yield and block.

3.3 Semantics

Our semantics of FX10 is inspired by the semantics for a larger subset of X10 given by Saraswat and Jagadeesan [17]. In FX10, all code runs on the same place.

We will now define a small-step operational semantics for FX10. In the semantics of while loops and method calls, we will use the following operator on statements. Let $s_1\ .\ s_2$ be defined as follows:

$$
\begin{array}{rcl}
\mathit{skip}^l\ .\ s_2 & \equiv & \mathit{skip}^l\ s_2 \\
(i\ s_1)\ .\ s_2 & \equiv & i\ (s_1\ .\ s_2)
\end{array}
$$

Our semantic structures are arrays, trees, and states:

$$
\begin{array}{rcl}
A \in \mathit{Array} & = & \mathbb{N} \to \mathbb{Z} \\
\mathit{Tree}: \quad T & ::= & T \rhd T \quad \mid \quad T \parallel T \quad \mid \quad \langle s \rangle \quad \mid \quad \surd \\
\mathit{State} & = & \mathit{Program} \times \mathit{Array} \times \mathit{Tree}
\end{array}
$$

We use A to denote the state of the array a, that is, a total mapping from natural numbers ($\mathbb{N}$) to integers ($\mathbb{Z}$). The initial state of a is called A_0. If c is a natural number, then $A(c)$ denotes the corresponding integer. We also define A on expressions: $A(c) = c$ and $A(a[d] + 1) = A(c) + 1$.

A tree $T_1 \rhd T_2$ is convenient for giving the semantics of finish: T_1 must complete execution before we move on to executing T_2. A tree $T_1 \parallel T_2$ represents a parallel execution of T_1 and T_2 that interleaves the execution of subtrees, except when disallowed by $\rhd$. A tree $\langle s \rangle$ represents statement s running. A tree $\surd$ has completed execution.

A state in the semantics is a triple (p, A, T). We will define the semantics via a binary relation on states, written $(p, A, T) \to (p, A', T')$. The initial state of an execution of p is $(p, A_0, \langle s_0 \rangle)$ where s_0 is the body of f_0, and f_0 is the name of the main method. Now we show the rules for taking a step from (p, A, T). Rules (1)–(6) below cover the cases where T is either of the form $(T_1 \rhd T_2)$ or of the form $(T_1 \parallel T_2)$, while Figure 2 shows the rules where T is of the form $\langle s \rangle$. There is no rule for the case of $(p, A, \surd)$.

$$(p, A, \surd \rhd T_2) \to (p, A, T_2) \tag{1}$$

$$\frac{(p, A, T_1) \to (p, A', T_1')}{(p, A, T_1 \rhd T_2) \to (p, A', T_1' \rhd T_2)} \tag{2}$$

$$(p, A, \surd \parallel T_2) \to (p, A, T_2) \tag{3}$$

$$(p, A, T_1 \parallel \surd) \to (p, A, T_1) \tag{4}$$

$$\frac{(p, A, T_1) \to (p, A', T_1')}{(p, A, T_1 \parallel T_2) \to (p, A', T_1' \parallel T_2)} \tag{5}$$

$$\frac{(p, A, T_2) \to (p, A', T_2')}{(p, A, T_1 \parallel T_2) \to (p, A', T_1 \parallel T_2')} \tag{6}$$

We can now state the deadlock-freedom theorem of Saraswat and Jagadeesan. Let $\to^*$ be the reflexive, transitive closure of $\to$.

THEOREM 1. **(Deadlock freedom)** *For every state (p, A, T), either $T = \surd$ or there exists A', T' such that $(p, A, T) \to (p, A', T')$.*

Proof. See Appendix A. □

4. May-Happen-in-Parallel Analysis

We use a type system to specify our modular, context-sensitive may-happen-in-parallel analysis. Every program has a type (Theorem 6) in our type system, which means that we can derive may-happen-in-parallel information for all programs. We first define three abstract domains and nine helper functions, and then proceed to show our type rules.

4.1 Abstract Domains and Helper Functions

We use $\mathcal{P}(S)$ to denote the powerset of a set S.

We define $\mathit{LabelSet} = \mathcal{P}(\mathit{Label})$. We use A, B, O, R to range over $\mathit{LabelSet}$.

We define $\mathit{LabelPairSet} = \mathcal{P}(\mathit{Label} \times \mathit{Label})$. We use M to range over $\mathit{LabelPairSet}$.

We define $\mathit{TypeEnv} = \mathit{MethodName} \to (\mathit{LabelPairSet} \times \mathit{LabelSet})$. We use E to range over $\mathit{TypeEnv}$; we will call each E a type environment.

Intuitively, we will use $\mathit{LabelSet}$ for collecting sets of labels of statements; we will use $\mathit{LabelPairSet}$ for collecting labels of pairs of statements that may happen in parallel; and we will use $\mathit{TypeEnv}$ to map methods to statements that may happen in parallel and to statements that may still be executing when the method completes execution.

We define nine functions on the data sets Tree, $\mathit{Statement}$, Label, $\mathit{LabelSet}$, and $\mathit{LabelPairSet}$, see Figure 3.

The function call $\mathit{Slabels}_p(s)$ conservatively approximates the set of labels of statements that may be executed during the execution of the statement s in program p. The function call $\mathit{Tlabels}_p(T)$ conservatively approximates the set of labels of statements that may be executed during the execution of the tree T in program p. Notice that $\mathit{Tlabels}$ is defined in terms of $\mathit{Slabels}$. The function call $\mathit{FSlabels}(s)$ returns the singleton set consisting of the label of s.

$$
\begin{array}{rcll}
(p, A, \langle skip^l \rangle) & \rightarrow & (p, A, \surd) & (7)\\
(p, A, \langle skip^l\ k \rangle) & \rightarrow & (p, A, \langle k \rangle) & (8)\\
(p, A, \langle a[d] =^l e; k \rangle) & \rightarrow & (p, A[c := A(e)], \langle k \rangle) & (9)\\
(p, A, \langle (while^l\ (a[d] \neq 0)\ s)\ k \rangle) & \rightarrow & (p, A, \langle k \rangle) \quad (\text{if } A(c) = 0) & (10)\\
(p, A, \langle (while^l\ (a[d] \neq 0)\ s)\ k \rangle) & \rightarrow & (p, A, \langle s\ .\ (while^l\ (a[d] \neq 0)\ s)\ k \rangle) \quad (\text{if } A(c) \neq 0) & (11)\\
(p, A, \langle (async^l\ s)\ k \rangle) & \rightarrow & (p, A, \langle s \rangle \parallel \langle k \rangle) & (12)\\
(p, A, \langle (finish^l\ s)\ k \rangle) & \rightarrow & (p, A, \langle s \rangle \rhd \langle k \rangle) & (13)\\
(p, A, \langle f_i()^l\ k \rangle) & \rightarrow & (p, A, \langle s_i\ .\ k \rangle) \quad (\text{where } p(f_i) = s_i) & (14)
\end{array}
$$

Figure 2. Operational semantics rules for (p, A, T) where T is of the form s.

The function call $FTlabels(T)$ conservatively approximates the set of labels of statements that can be executed next in the tree T. Notice that $FTlabels$ is defined in terms of $FSlabels$. The function call $symcross(A, B)$ returns the union of the crossproduct of A and B with the crossproduct of B and A. We need $symcross$ to help produce a symmetric set of pairs of labels. The functions $Lcross$, $Scross$, and $Tcross$ are convenient abbreviations of calls to $symcross$. The function call $parallel(T)$ specifies for the tree T a set of pairs of labels of statements that are "executing in parallel right now", that is, for each pair, both can take a step now. Notice that $parallel$ is defined in terms of $symcross$ and $FTlabels$. The function $parallel$ is central to our definition of correctness: for every reachable tree T, we must conservatively approximate $parallel(T)$.

4.2 Type Rules

We will use type judgments of three forms:

$$
\begin{array}{rcl}
 & \vdash & p : E\\
p, E, R & \vdash & T : M\\
p, E, R & \vdash & s : M, O
\end{array}
$$

The first form of judgment says that program p is well typed and that the methods in p have the types given by E. The second form of judgment says that tree T is well typed in a situation where R is a set of labels of statements that may run in parallel with T when T starts execution, and M is a set of pairs of labels such that for each pair (l_1, l_2), the instructions with labels l_1 and l_2 may happen in parallel during the execution of T. We will call M the *may-happen-in-parallel set*. The third form of judgment says that statement s is well typed in a situation much like the previous one, now with the addition that O is the set of labels of instructions that may be executing when the execution of s terminates. For $p, E, R \vdash s : M, O$, we will always have $R \subseteq O$; in other words, O can contain labels of both statements that started before s and statements that started during the execution of s. A type environment E that maps a method name f_i to a pair (M_i, O_i) represents that during a call to f_i, the pairs in M_i may happen in parallel, and the statements with labels in O_i may be executing when the call to f_i returns.

Figure 4 shows the type rules.

Rule (45) says that a program is well typed with a type environment E if each method body has the type specified by E in a situation where $R = \emptyset$. This rule enables modular type checking: we only need to type check each method once, even though method calls may be made in situations where $R \neq \emptyset$.

Rule (46) says three things. First, the set of labels R of statements that may run in parallel with $T_1 \rhd T_2$ when $T_1 \rhd T_2$ starts execution, are also the set of labels of statements that may run in parallel with T_1 and with T_2 when each of them starts execution. Second, we get the may-happen-in-parallel set for $T_1 \rhd T_2$ by taking the union of the may-happen-in-parallel set for T_1 and the may-happen-in-parallel set for T_2. Third, there is no interaction between T_1 and T_2 that produces new pairs of labels of statements that may happen in parallel. This rule has a close cousin in Rule (55) for finish statements.

Rule (47) says that for a tree $T_1 \parallel T_2$, the analysis of T_1 must take into account that T_2 may already be executing, and vice versa. We do that by extending R with labels from the appropriate subtree, for example $Tlabels(T_2)$. This rule has a close cousin in Rule (54) for async statements.

Rule (48) says that we can type check a tree $\langle s \rangle$ by typing the statement s.

Rule (49) says that if a subtree has completed execution, then nothing runs in parallel with it.

Rule (50) says that the skip instruction runs in parallel with the statements with labels in R. The $Lcross()$ function represents every possible pairing of the labels in R with skip's label. Since skip does not generate statements that may run in parallel after the execution of the skip, we see that the set of labels of instructions that may be executing when skip terminates is R.

Rule (51) works similarly to the previous rule, with the exception that we are additionally dealing with a substatement after the skip statement. We see that the skip label may run in parallel with the R labels, which is represented via the use of $Lcross()$. We now type the substatement s_1 where we retain the same R for the environment because skip doesn't generate anything that can run in parallel. The resulting O labels from the s_1 judgement will be the O labels returned from the judgement for $skip; s_1$. The may-happen-in-parallel set is the union of the set produced by $Lcross()$ that we have seen above and the M from the s_1 judgement.

Rule (52) is similar to Rule (51).

Rule (53) is based on a conservative assumption: the loop body will be executed at least twice. Two iterations are sufficient to model situations in which the loop body may happen in parallel with itself. The rule relies on the assumption when it includes $Lcross(l, O_1)$ and $Scross_p(s_1, O_1)$ in the may-happen-in-parallel set. The rule also shows how we use the set O_1 when typing a sequence of statements, which here is a sequence of a while loop and s_2: we use the set O_1 as the set of labels of statements executing at the beginning of execution of s_2.

Rule (54) says that for a statement $async^l\ s_1\ s_2$ the analysis of s_1 must take into account that s_2 may already be executing, and vice versa. We do that by extending R with labels from the appropriate statement, for example $Slabels(s_2)$. By adding the entire $Slabels(s_2)$ we make the conservative assumption that the entire async body may run in parallel with the continuation, and vice versa. Notice that the label set O_1 appears once in the first

$Slabels : Program \to (Statement \to LabelSet)$
$Slabels_p$ is the $\subseteq$-least solution to the following equations.

$$
\begin{aligned}
Slabels_p(skip^l) &= \{l\} && (15)\\
Slabels_p(skip^l\ k) &= \{l\} \cup Slabels_p(k) && (16)\\
Slabels_p(a[d] =^l e;\ k) &= \{l\} \cup Slabels_p(k) && (17)\\
Slabels_p(while^l\ (a[d] \neq 0)\ s\ k) &= \{l\} \cup Slabels_p(s) \cup Slabels_p(k) && (18)\\
Slabels_p(async^l\ s\ k) &= \{l\} \cup Slabels_p(s) \cup Slabels_p(k) && (19)\\
Slabels_p(finish^l\ s\ k) &= \{l\} \cup Slabels_p(s) \cup Slabels_p(k) && (20)\\
Slabels_p(f_i()^l\ k) &= \{l\} \cup Slabels_p(s_i) \cup Slabels_p(k) \text{ if } p(f_i) = s_i && (21)
\end{aligned}
$$

$Tlabels : Program \to (Tree \to LabelSet)$

$$
\begin{aligned}
Tlabels_p(\surd) &= \emptyset && (22)\\
Tlabels_p(T_1 \rhd T_2) &= Tlabels_p(T_1) \cup Tlabels_p(T_2) && (23)\\
Tlabels_p(T_1 \parallel T_2) &= Tlabels_p(T_1) \cup Tlabels_p(T_2) && (24)\\
Tlabels_p(\langle s \rangle) &= Slabels_p(s) && (25)
\end{aligned}
$$

$FSlabels : Statement \to LabelSet$

$$
\begin{aligned}
FSlabels(skip^l) &= \{l\} && (26)\\
FSlabels(skip^l\ k) &= \{l\} && (27)\\
FSlabels(a[d] =^l e;\ k) &= \{l\} && (28)\\
FSlabels(while^l\ (a[d] \neq 0)\ s\ k) &= \{l\} && (29)\\
FSlabels(async^l\ s\ k) &= \{l\} && (30)\\
FSlabels(finish^l\ s\ k) &= \{l\} && (31)\\
FSlabels(f_i()^l\ k) &= \{l\} && (32)
\end{aligned}
$$

$FTlabels : Tree \to LabelSet$

$$
\begin{aligned}
FTlabels(\surd) &= \emptyset && (33)\\
FTlabels(T_1 \rhd T_2) &= FTlabels(T_1) && (34)\\
FTlabels(T_1 \parallel T_2) &= FTlabels(T_1) \cup FTlabels(T_2) && (35)\\
FTlabels(\langle s \rangle) &= FSlabels(s) && (36)
\end{aligned}
$$

$symcross : LabelSet \times LabelSet \to LabelPairSet$

$$symcross(A, B) = (A \times B) \cup (B \times A) \quad (37)$$

$Lcross : Label \times LabelSet \to LabelPairSet$

$$Lcross(l, A) = symcross(\{l\}, A) \quad (38)$$

$Scross : Program \to (Statement \times LabelSet \to LabelPairSet)$

$$Scross_p(s, A) = symcross(Slabels_p(s), A) \quad (39)$$

$Tcross : Program \to (Tree \times LabelSet \to LabelPairSet)$

$$Tcross_p(T, A) = symcross(Tlabels_p(T), A) \quad (40)$$

$parallel : Tree \to LabelPairSet$

$$
\begin{aligned}
parallel(\surd) &= \emptyset && (41)\\
parallel(T_1 \rhd T_2) &= parallel(T_1) && (42)\\
parallel(T_1 \parallel T_2) &= parallel(T_1) \cup parallel(T_2) \cup symcross(FTlabels(T_1), FTlabels(T_2)) && (43)\\
parallel(\langle s \rangle) &= \emptyset && (44)
\end{aligned}
$$

Figure 3. Helper definitions.

$$\frac{\begin{array}{l} p = void\ f_i()\ \{\ s_i\ \}, 1..u \\ E = \{f_i \mapsto (M_i, O_i)\} \\ p, E, \emptyset \vdash s_i : M_i, O_i \end{array}}{\vdash p : E} \quad (45)$$

$$\frac{p, E, R \vdash T_1 : M_1 \qquad p, E, R \vdash T_2 : M_2}{p, E, R \vdash T_1 \rhd T_2 : M_1 \cup M_2} \quad (46)$$

$$\frac{\begin{array}{l} p, E, Tlabels(T_2) \cup R \vdash T_1 : M_1 \\ p, E, Tlabels(T_1) \cup R \vdash T_2 : M_2 \end{array}}{p, E, R \vdash T_1 \parallel T_2 : M_1 \cup M_2} \quad (47)$$

$$\frac{p, E, R \vdash s : M_s, O}{p, E, R \vdash \langle s \rangle : M_s} \quad (48)$$

$$\frac{}{p, E, R \vdash \surd : \emptyset} \quad (49)$$

$$\frac{}{p, E, R \vdash skip^l : Lcross(l, R), R} \quad (50)$$

$$\frac{p, E, R \vdash s_1 : M, O}{p, E, R \vdash skip^l\ s_1 : Lcross(l, R) \cup M, O} \quad (51)$$

$$\frac{p, E, R \vdash s_1 : M, O}{p, E, R \vdash a[d] =^l e;\ s_1 : Lcross(l, R) \cup M, O} \quad (52)$$

$$\frac{p, E, R \vdash s_1 : M_1, O_1 \qquad p, E, O_1 \vdash s_2 : M_2, O_2}{\begin{array}{l} p, E, R \vdash while^l\ (a[d] \neq 0)\ s_1\ s_2 : \\ \quad Lcross(l, O_1) \cup Scross_p(s_1, O_1) \cup M_1 \cup M_2, O_2 \end{array}} \quad (53)$$

$$\frac{\begin{array}{l} p, E, Slabels_p(s_2) \cup R \vdash s_1 : M_1, O_1 \\ p, E, Slabels_p(s_1) \cup R \vdash s_2 : M_2, O_2 \end{array}}{\begin{array}{l} p, E, R \vdash async^l\ s_1\ s_2 : \\ \quad Lcross(l, R) \cup M_1 \cup M_2, O_2 \end{array}} \quad (54)$$

$$\frac{p, E, R \vdash s_1 : M_1, O_1 \qquad p, E, R \vdash s_2 : M_2, O_2}{p, E, R \vdash finish^l\ s_1\ s_2 : Lcross(l, R) \cup M_1 \cup M_2, O_2} \quad (55)$$

$$\frac{E(f_i) = (M_i, O_i) \qquad p, E, R \cup O_i \vdash k : M', O'}{\begin{array}{l} p, E, R \vdash f_i()^l\ k : \\ \quad Lcross(l, R) \cup symcross(Slabels_p(p(f_i)), R) \cup \\ \quad M_i \cup M', \\ \quad O' \end{array}} \quad (56)$$

Figure 4. Type rules.

hypothesis and never again; let us explain why this seemingly strange phenomenon makes sense. In any typing judgement such as $p, E, R \vdash s_1 : M_1, O_1$, we have that O_1 is a union of R and some O' (this is a lemma in our proof of correctness). The set O' must be a subset of $Slabels_p(s_1)$ as the async statement is the only time where new labels are introduced into O. So, in the typing of s_2, the set $Slabels_p(s_1)$ contains O'.

Rule (55) says that the set O_1 produced by the typing of the finish body can be ignored. So, we use the initial R for typing both the finish body s_1 and the continuation s_2, and thereby indicate that we are disregarding whatever statements that may be running as a result of executing s_1. In other words, we don't use O_1 in the typing of s_2. As a result, if any labels occur in O_1 that are not in R, the rule reflects that the corresponding statements will not happen in parallel with s_2. The statements with labels in R that were executing when s_1 started execution may still be executing when s_2 starts execution so we use R in the typing of s_2 to account for that.

Rule (56) shows how to type check a call with an arbitrary R even though Rule (45) has only provided a type environment in which methods have been type checked with $R = \emptyset$. The type environment says that for $R = \emptyset$, the set O_i contains the labels of statements that may be executing at the end of the call. We then simply take the union of R and O_i and use that for typing the continuation k. The may-happen-in-parallel set for the method call contains $symcross(Slabels_p(p(f_i)), R)$, a set that reflects that anything that may happen in parallel with call may also happen in parallel with the body.

The following soundness theorem says that for a program p and any tree T reachable by executing p, the set $parallel(T)$ is a subset of the the may-happen-in-parallel set determined by type checking p. Intuitively, the type system conservatively approximates all $parallel(T)$.

THEOREM 2. **(Soundness)** *If* $\vdash p : E$ *and* $p, E, \emptyset \vdash \langle s_0 \rangle : M$ *and* $(p, A_0, \langle s_0 \rangle) \rightarrow^* (p, A, T)$ *then* $parallel(T) \subseteq M$.

Proof. See Appendix B. □

For a program p, define

$$\mathsf{MHP}(p) \quad = \quad \bigcup \{\ parallel(T) \mid (p, A_0, \langle s_0 \rangle) \rightarrow^* (p, A, T)\ \}$$

THEOREM 3. **(Correctness)** *If* $\vdash p : E$ *and* $E(f_0) = (M, O)$, *then* $\mathsf{MHP}(p) \subseteq M$.

Proof. Immediate from Theorem 2. □

5. Type Inference

The type inference problem is: given a program p, find E such that $\vdash p : E$. We will do type inference in two steps: first we rephrase the type inference problem as an equivalent constraint problem, and then we solve the constraint problem.

5.1 Constraints

Variables. For every statement s we will generate three set variables: r_s, o_s, and m_s. The variables r_s and o_s will range over sets of labels, while the variable m_s will range over sets of pairs of labels. For every method f_i we will generate two set variables: o_i and m_i.

Kinds of constraints. We will use two kinds of constraints. The *level-1* constraints are of the forms:

$$\begin{array}{lcl} v & = & v' \\ v & = & c \\ v & = & c \cup v' \end{array}$$

where v is an r variable or an o variable, v' is an r variable or an o variable, and c is a set constant. The *level-2* constraints are of the forms:

$$\begin{array}{lcl} v & = & v'' \\ v & = & Lcross(l, v') \\ v & = & Lcross(l, v') \cup v'' \\ v & = & Lcross(l, v') \cup v'' \cup v''' \\ v & = & Lcross(l, v') \cup Scross(c, v') \cup v'' \cup v''' \\ v & = & Lcross(l, v') \cup symcross(c, v') \cup v'' \cup v''' \end{array}$$

where v is an m variable, v' is an r variable or an o variable, v'' and v''' are m variables, l is the label associated with a statement, and c is a set constant.

$$
\begin{aligned}
r_{S0} &= \{\} \\
r_{S1} &= r_{S0} \\
r_{S13} &= \{S2\} \cup r_{S1} \\
r_{S5} &= r_{S13} \\
r_{S6} &= r_{S5} \\
r_{S11} &= \{S7, S12\} \cup r_{S6} \\
o_{S11} &= r_{S11} \\
r_{S7} &= \{S11\} \cup r_{S6} \\
r_{S12} &= r_{S7} \\
o_{S12} &= r_{S12} \\
o_{S7} &= \{S12\} \cup r_{S7} \\
o_{S6} &= o_{S7} \\
o_{S5} &= o_{S6} \\
r_{S8} &= r_{S13} \\
o_{S8} &= r_{S8} \\
o_{S13} &= o_{S8} \\
r_{S2} &= \{S5, S6, S7, S8, S11, S12, S13\} \cup r_{S1} \\
o_{S2} &= r_{S2} \\
o_{S1} &= o_{S2} \\
r_{S3} &= r_{S0} \\
o_{S3} &= r_{S3} \\
o_{S0} &= o_{S3}
\end{aligned}
$$

$$
\begin{aligned}
m_{S1} &= Lcross(S1, r_{S1}) \cup m_{S13} \cup m_{S2} \\
m_{S6} &= Lcross(S6, r_{S6}) \cup m_{S11} \cup m_{S7} \\
m_{S11} &= Lcross(S11, r_{S11}) \\
m_{S7} &= Lcross(S7, r_{S7}) \cup m_{S12} \\
m_{S12} &= Lcross(S12, r_{S12}) \\
m_{S5} &= Lcross(S5, r_{S5}) \cup m_{S6} \\
m_{S8} &= Lcross(S8, r_{S8}) \\
m_{S13} &= Lcross(S13, r_{S13}) \cup m_{S5} \cup m_{S8} \\
m_{S2} &= Lcross(S2, r_{S2}) \\
m_{S3} &= Lcross(S3, r_{S3}) \\
m_{S0} &= Lcross(S0, r_{S0}) \cup m_{S1} \cup m_{S3}
\end{aligned}
$$

Figure 5. Constraints for the example program in Section 2.1.

Valuations. For a given system of constraints $\mathcal{C}$, let L be the set of labels that occur in $\mathcal{C}$. Let $\mathcal{D}$ denote the domain of *valuations* of the set variables: each function in $\mathcal{D}$ maps each r and o variable that occurs in $\mathcal{C}$ to a subset of L, it maps each m variable that occurs in $\mathcal{C}$ to a subset of $L \times L$, and, for convenience, it maps each o_i variable to a subset of L and it maps each m_i variable to a subset of $L \times L$, without regard to whether those o_i and m_i variables occur in $\mathcal{C}$. It is straightforward to show that $\mathcal{D}$ is a finite lattice.

Solutions. We say that $\varphi \in \mathcal{D}$ is a *solution* of the system of constraints if for every constraint $v = rhs$, we have $\varphi(v) = \varphi(rhs)$. Here we use rhs to range over the possible right-hand sides of the constraints, and we use $\varphi(rhs)$ to denote rhs with each variable v' occurring in rhs replaced with $\varphi(v')$.

Constraint generation. We use $C(p)$ to denote the constraints generated from a program p, and we use $C(s)$ to denote the constraints generated a from statement s. We will define $C(p)$ and $C(s)$ below.

For each method f_i in $p \equiv void\ f_i()\ \{\ s_i\ \}, 1..u$, we define $C(p) = \bigcup_i (D_i \cup C(s_i))$. We define D_i to have the following constraints:

$$
\begin{aligned}
r_{s_i} &= \emptyset && (57)\\
o_i &= o_{s_i} && (58)\\
m_i &= m_{s_i} && (59)
\end{aligned}
$$

For $s \equiv skip^l$ we define $C(s) = D_s$ where D_s is defined by the following constraints:

$$
\begin{aligned}
o_s &= r_s && (60)\\
m_s &= Lcross(l, r_s) && (61)
\end{aligned}
$$

For $s \equiv skip^l\ s_1$ we define $C(s) = D_s \cup C(s_1)$ where D_s contains the following constraints:

$$
\begin{aligned}
r_{s_1} &= r_s && (62)\\
o_s &= o_{s_1} && (63)\\
m_s &= Lcross(l, r_s) \cup m_{s_1} && (64)
\end{aligned}
$$

For $s \equiv a[d] =^l e;\ s_1$ we define $C(s) = D_s \cup C(s_1)$ where we define D_s to have the constraints below:

$$
\begin{aligned}
r_{s_1} &= r_s && (65)\\
o_s &= o_{s_1} && (66)\\
m_s &= Lcross(l, r_s) \cup m_{s_1} && (67)
\end{aligned}
$$

For $s \equiv while^l\ (a[d] \neq 0)\ s_1\ s_2$ we define $C(s) = D_s \cup C(s_1) \cup C(s_2)$ where we define D_s to have the following constraints:

$$
\begin{aligned}
r_{s_1} &= r_s && (68)\\
r_{s_2} &= o_{s_1} && (69)\\
o_s &= o_{s_2} && (70)\\
m_s &= \left(\begin{array}{l} Lcross(l, o_{s_1}) \cup Scross_p(s_1, o_{s_1}) \cup \\ m_{s_1} \cup m_{s_2} \end{array} \right) && (71)
\end{aligned}
$$

For $s \equiv async^l\ s_1\ s_2$ we define $C(s) = D_s \cup C(s_1) \cup C(s_2)$ and define D_s to have the constraints:

$$
\begin{aligned}
r_{s_1} &= Slabels(s_2) \cup r_s && (72)\\
r_{s_2} &= Slabels(s_1) \cup r_s && (73)\\
o_s &= o_{s_2} && (74)\\
m_s &= Lcross(l, r_s) \cup m_{s_1} \cup m_{s_2} && (75)
\end{aligned}
$$

For $s \equiv finish^l\ s_1\ s_2$ we have $C(s) = D_s \cup C(s_1) \cup C(s_2)$. We define D_s to have the following constraints:

$$
\begin{aligned}
r_{s_1} &= r_s && (76)\\
r_{s_2} &= r_s && (77)\\
o_s &= o_{s_2} && (78)\\
m_s &= Lcross(l, r_s) \cup m_{s_1} \cup m_{s_2} && (79)
\end{aligned}
$$

And finally for $s \equiv f_i()^l\ k$ we have $C(s) = D_s \cup C(k)$. D_s is defined to have the following constraints:

$$
\begin{aligned}
r_k &= r_s \cup o_i && (80)\\
o_s &= o_k && (81)\\
m_s &= \left(\begin{array}{l} Lcross(l, r_s) \cup \\ symcross(Slabels_p(p(f_i)), r_s) \cup \\ m_i \cup m_k \end{array} \right) && (82)
\end{aligned}
$$

Types and constraints are equivalent in the sense of Theorem 4 below. Intuitively, a program has a type if and only if the constraints are solvable. Additionally, we can map a type derivation to a solution to the constraint system, and vice versa. To state the theorem,

we need the following definition. For $\varphi \in \mathcal{D}$, we say that φ extends E if and only if $\forall f_i \in dom(E) : (\varphi(m_i), \varphi(o_i)) = E(f_i)$.

THEOREM 4. **(Equivalence)** $\vdash p : E$ *if and only if there exists a solution* φ *of* $C(p)$ *where* φ *extends* E.

Proof. See Appendix C. □

Theorems like Theorem 4 that relate types and constraints have been known since a paper by Kozen et al. [11].

5.2 Solving Constraints

We will now explain how to solve the constraints $C(p)$ generated from a program p. Our solution procedure resembles the algorithms used for iterative data flow analysis.

Notice that the constraints in $C(p)$ have distinct left-hand sides and that every variable is the left-hand side of some constraint. This enables us to define the function

$$\begin{aligned} F &: \mathcal{D} \to \mathcal{D} \\ F &= \lambda\varphi \in \mathcal{D}.\lambda v.\varphi(rhs) \\ &\qquad (\text{where } v = rhs \text{ is a constraint}) \end{aligned}$$

It is straightforward to show that F is monotone. So, F is a monotone function from a finite lattice $\mathcal{D}$ to itself. The least-fixed-point theorem guarantees that F has a least fixed point. Moreover, it is straightforward to see that the fixed points of F coincide with the solutions of $C(p)$. Hence, the least fixed point of F is the least solution of $C(p)$ and thus we have shown the following theorem.

THEOREM 5. $C(p)$ *has a least solution.*

We solve the constraints $C(p)$ by executing the fixed-point computation that computes the least fixed point of F. The worst-case time complexity is $O(n^6)$ where n is the size of the constraint system. Let us explain the reason for the $O(n^6)$ time complexity in detail. First, we have $O(n)$ m variables that each can contain $O(n^2)$ pairs, so we have $O(n^3)$ iterations. In each iteration we consider $O(n)$ constraints and for each one we must do a finite number of set unions. If we represent each set as a bit vector with $O(n^2)$ entries, then set union takes $O(n^2)$ time. The total is thus $O(n^3) \times O(n) \times O(n^2) = O(n^6)$.

The guaranteed existence of a least solution of $C(p)$ implies that p has a type, as expressed in the following theorem.

THEOREM 6. *There exists* E *such that* $\vdash p : E$.

Proof. Combine Theorem 4 and Theorem 5. □

5.3 Implementation

One approach to implementing type inference would be to solve the constraints all at once. As an optimization of that, our implementation of type inference proceeds in three steps:

1. solve the equations that define $Slabels$,
2. solve the level-1 constraints, and finally
3. solve the level-2 constraints.

The level-1 constraints don't involve m variables so we can solve them without involving the level-2 constraints. Once we have a solution to the level-1 constraints, we can simplify the level-2 constraints by replacing each r variable and o variable with its solved form. The simplified level-2 constraints are of the forms

$$\begin{aligned} v &= v'' \\ v &= c \\ v &= c \cup v'' \\ v &= c \cup v'' \cup v''' \end{aligned}$$

where v, v'', v''' are m variables, and c is a set constant.

The equations that define $Slabels$ are in the form of simplified level-2 constraints and we solve them using the same iterative approach that we use for level-2 constraints.

The constraints for FX10 are all we need to type inference for the full X10 language; the remaining constructs generate constraints that are similar to those for FX10.

5.4 Example

From the program in Section 2.1, we generate the constraints listed in Figure 5. As explained in Section 2.1, the output from our constraint solver says correctly that `S2` may happen in parallel with each of `S5`, `S6`, `S7`, `S8`, `S11`, and `S12`, as well as with the entire `finish` statement, that `S11` and `S12` may happen in parallel, and that `S7` and `S11` may happen in parallel.

6. Experimental Results

We ran our experiments on a system that has dual *Intel Xeon CPUs* running at 3.06GHz with 512 KB of cache and 4GB of main memory.

We use 13 benchmarks taken from the HPC challenge benchmarks, the Java Grande benchmarks in X10, the NAS benchmarks, and two benchmarks written by ourselves. Figure 6 shows the number of lines of code (LOC), the number of asyncs and the number of constraints. The number of asyncs includes the number of foreach and ateach loops, which are X10 constructs that let all the loop iterations run in parallel. We can think of foreach and ateach as plain loops where the body is wrapped in an async. Our own plasma simulation benchmark, called plasma, is the longest and by far the most complicated benchmark with 151 asyncs.

Figure 6 shows a division of the asyncs into two categories: loop asyncs and place-switching asyncs. Loop asyncs are asyncs that occur in loops and are not wrapped in a finish; such asyncs may happen in parallel with asyncs from different iterations of the same loop. The vast majority of the loop asyncs occur in ateach and foreach loops. Place-switching asyncs are based on a more general form of async than what FX10 supports and are used to switch between places. Our implementation handles the more general form of async in exactly the same way as the asyncs in FX10. Most often such place-switching enables data transfers or remote computation. A common usage found in our benchmarks is creating a data value such that it may be usable across async boundaries and then storing that data in a buffer on the place where the data is needed. Note here that for an ateach loop, we count the implicit async as a loop async even though it also serves the purpose of place switching.

Our implementation of type inference for X10 first translates an X10 program to a condensed form that closely resembles FX10, and then it proceeds to generate and solve constraints. The condensed form has ten kinds of nodes, namely end, async, call, finish, if, loop, method, return, skip, and switch, see Figure 7. The total number of nodes is a good measure of the size of the input to our type inference algorithm. Switch nodes are unlike anything we have in FX10; we use them to accommodate various control-flow statements. End nodes do not correspond to any program point in the code, but act as place holders for our constraint system. Skip nodes are all the various statements and expressions that don't affect the analysis and represent blocks of code that don't contain any method calls, returns, asyncs or finishes.

Figure 6 lists the numbers of constraints, and Figure 8 lists the time to do type inference and the executed number of iterations. Method calls appear to add a significant amount of time to solve the constraints, most notably seen in the number of iterations required to solve the $Slabels$ constraints. When an iteration for computing label sets completes, a call site will need to propagate any new labels to neighboring statements and eventually the enclosing

	LOC	#async			#constraints		
		total	loop	place switch	Slabels	level-1	level-2
HPC challenge benchmarks:							
stream	70	4	3	1	103	232	103
fragstream	73	4	3	1	103	232	103
Java Grande benchmarks:							
sor	185	7	2	5	132	298	132
series	290	3	1	2	90	224	90
sparsemm	366	4	1	3	173	370	173
crypt	562	2	2	0	149	326	149
moldyn	699	14	6	8	241	596	241
linpack	781	8	3	5	225	547	225
raytracer	1,205	13	2	11	478	1,045	478
montecarlo	3,153	3	1	2	345	727	345
NAS benchmarks:							
mg	1,858	57	37	20	1,028	2,518	1,028
Our own benchmarks:							
mapreduce	53	3	1	2	40	96	40
plasma	4,623	151	120	31	2,596	6,230	2,596

Figure 6. Experimental results: static measurements.

	#nodes										
	Total	End	Async	Call	Finish	If	Loop	Method	Return	Skip	Switch
HPC challenge benchmarks:											
stream	126	23	4	5	4	3	10	20	21	36	0
fragstream	126	23	4	5	4	3	10	20	21	36	0
Java Grande benchmarks:											
sor	161	29	7	21	5	1	7	24	16	51	0
series	119	29	3	17	2	3	7	14	7	36	1
sparsemm	201	28	4	25	3	0	16	32	27	66	0
crypt	175	26	2	25	2	5	9	24	21	61	0
moldyn	316	75	14	25	14	2	29	36	22	99	0
linpack	286	61	8	42	6	10	19	25	17	98	0
raytracer	555	77	13	132	9	16	8	65	50	185	0
montecarlo	405	60	3	80	3	2	6	83	39	129	0
NAS benchmarks:											
mg	1,320	292	57	248	52	40	68	122	87	354	0
Our own benchmarks:											
mapreduce	52	12	3	5	2	0	3	8	4	15	0
plasma	3,200	604	151	505	84	93	231	170	221	1,140	1

Figure 7. Experimental results: number of nodes.

	time	space	Number of iterations			#pairs of async bodies that MHP			
	(ms)	(MB)	Slabels	level-1	level-2	total	self	same	diff
HPC challenge benchmarks:									
stream	153	5	3	2	2	5	4	1	0
fragstream	158	5	3	2	2	5	4	1	0
Java Grande benchmarks:									
sor	219	6	5	2	3	13	6	3	4
series	230	9	4	2	4	1	1	0	0
sparsemm	225	8	4	2	3	3	2	1	0
crypt	218	8	4	2	2	2	2	0	0
moldyn	420	24	5	2	3	59	14	36	9
linpack	331	13	4	3	3	10	6	1	3
raytracer	3,105	173	5	2	4	49	13	24	12
montecarlo	1,403	132	6	2	4	4	3	1	0
NAS benchmarks:									
mg	5,197	196	6	3	5	272	51	17	204
Our own benchmarks:									
mapreduce	96	3	3	2	3	1	1	0	0
plasma	16,476	257	6	2	6	258	134	120	4

Figure 8. Experimental results: type inference.

	analysis	time (ms)	space MB)	Number of iterations Slabels	level-1	level-2	#pairs of async bodies that MHP total	self	same	diff
NAS benchmarks:										
mg	context-sensitive	5,197	196	6	3	5	272	51	17	204
mg	context-insensitive	25,935	350	6	17	5	681	52	23	606
Our own benchmarks:										
plasma	context-sensitive	16,476	257	6	2	6	258	134	120	4
plasma	context-insensitive	167,828	1,429	6	14	6	2,281	136	126	2,019

Figure 9. Experimental results: comparison of our context-sensitive analysis to a context-insensitive analysis.

method will need another iteration to disseminate new sets to its callers. This effect does not appear when solving the level-1 and level-2 constraints; we believe that finish statements help limit the propagation. Finish statements cap how far the sets can flow down a call chain, which translates into fewer iterations.

For evaluation of the quality of our analysis, we focus on counting pairs of labels of entire async bodies. Figure 8 shows the number of pairs of async bodies that may happen in parallel, according to our analysis, together with three exhaustive and disjoint subcategories. The legend of Figure 8 is: self = an async body may happen in parallel with itself; same = two different async bodies in the same method may happen in parallel; diff = two async bodies in different methods may happen in parallel. Let us discuss each of the columns in turn. A typical scenario for the *self* category is:

```
while (...) { async S1 }
```

Notice that `S1` may happen in parallel with itself. If we compare the self column to the total number of asyncs in the program, we can easily determine how many asyncs appear in loops (or in methods called in loops) without a finish for wrapping the async. Most of the benchmarks have a high percentage of such asyncs, which we expected as this is the easiest way to generate parallelism in X10. Some of the smaller benchmarks like series and mapreduce use just one loop to do most of the processing, but also need to perform some communication which is done with the other asyncs.

In our benchmarks, a typical scenario for the *same* category is:

```
while (...) {
  async {
    finish async S1
    finish async S2
  }
}
```

Here, `S1` and `S2` may happen in parallel because separate iterations of the loop run in parallel with each other. Such code is useful when we don't need synchronization among separate iterations of a loop but need a strict order of execution during a single iteration.

The example is Section 2.2 is a typical scenario for the diff category. For example, statements `S5` and `S3` may happen in parallel and are in separate methods. Most of the benchmarks have few pairs of async bodies in this category. However, one can easily move a pair from the *same* category to the diff category by moving an async in a loop to a method that the loop then calls. In mg, we have several methods with asyncs in their bodies that are called from several different loops. Some calls were deeply nested in several loop async bodies.

We manually examined the type-inference output for stream, fragstream, sor, series, sparsemm, crypt, and mapreduce to look for *false positives*, that is, pairs of async bodies that our algorithm says can happen in parallel but actually can't. We found none! For the other, larger benchmarks, the generated number of pairs is large and we performed only a brief examination and noticed no obvious false positives. Asyncs in the bodies of loops are typical in the benchmarks and don't provide false positives unless the loop guard is always false which we believe is not the case in any of the examples we closely examined, for the inputs we used.

7. Context-insensitive Analysis

We will now compare our context-sensitive analysis to a context-insensitive analysis that merges information from different call sites. Let us first explain how the context-insensitive analysis works: it uses the *same* set variables and constraints as our context-sensitive analysis, *except* for the following differences.

Variables. For every method f_i, we generate an extra set variable r_i.

Constraint generation. For $s \equiv f_i()^l\ k$ we add the following constraint:

$$r_s \subseteq r_i \tag{83}$$

We also replace Rule (57) with the following constraint:

$$r_{s_i} = r_i \tag{84}$$

The effect of these changes is a merge of the r_s variables from different call sites. Thus, the context-insensitive analysis says that the method may happen in parallel with the labels in the sets for all those r_s variables at once.

A subtlety is that for a context-insensitive analysis we can remove $Scross_p(p(f_i), R)$ from Rule (82) without changing the analysis. This is because the pairs generated by $Scross_p(p(f_i), R)$ will eventually be added anyway due to the new $r_s \subseteq r_i$ constraint.

We ran the context-insensitive analysis on our benchmarks. For the 11 smallest benchmarks, the runs used roughly the same amount of time and space, and we got the exact same results. Only for the two largest benchmarks, plasma and mg, did the context-insensitive analysis produce any additional label pairs in the may-happen-in-parallel sets. Figure (9) shows a comparison of our context-sensitive analysis and a context-insensitive analysis of plasma and mg. The context-insensitive analysis requires more time and space, and it produces many more pairs of async bodies that may happen in parallel.

The increase in run time and space usage of the context-insensitive analysis compared to our context-sensitive analysis is somewhat unsurprising. First, the context-insensitive analysis is more conservative so the number of label pairs that are generated and copied through the constraint variables is higher. In particular, the higher number of pairs increases the time required to perform the set operations. Second, the introduction of subset constraints leads to an increase in the number of level-1 iterations. The reason is that each call site can contribute labels to r_i and then the constraint solver needs additional iterations to propagate the additional labels amongst the constraints.

The increase in the number of label pairs is mostly for async bodies in different methods. As far as we can tell, the increase is due to a few methods that are called in many different places. Such a method can easily have an overly conservative o set that then leads to many spurious pairs. The reason is that call site contributes

to the r_i set for a method, and the set for r_i will be a subset of o_i, so now o_i has many elements to be paired with labels of statements that follow each call.

The example in Section 2.2 illustrates this effect. The statement `S3` is running at the beginning of the first call `f()`, and so it will running when that call completes execution. Due to the merging of information from different call sites, the analysis finds that `S3` is also running at the end of the second call `f()`. When the analysis consider the statement `async S4` that follows the second call `f()`, it will conclude that `S3` and `S4` may happen in parallel.

We thank Vivek Sarkar (personal communication, 2009) for the following observation. The intraprocedural analysis of [2] ignores function calls and uses the two finish statements to conclude that `S3` and `S4` cannot happen in parallel. The context-insensitive analysis of function calls creates an infeasible datapath from the body of one finish statement to the body of another finish statement and therefore the spurious pair of `S3` and `S4`. In contrast, our analysis avoids such infeasible datapaths and doesn't produce the spurious pair of `S3` and `S4`.

8. Conclusion

We have presented a core calculus for async-finish parallelism along with a type system for modular, context-sensitive may-happen-in-parallel analysis. Type inference is straightforward: generate and solve simple set constraints in polynomial time. Compared to a context-insensitive analysis, our context-sensitive analysis is faster, uses less space, and produces better results.

Our experiments suggest that our analysis produces few false positives and should therefore be a good basis for other static program analyses. In fact we have been unable to find any false positives at all! One way a false positive can occur is if a program has a loop that is never executed: our analysis will analyze the loop anyway. For example:

```
while (...) { async S1 }
async S2
```

Suppose the while loop is never executed. Our analysis will nevertheless say that `S1` and `S2` may happen in parallel. We found no occurrences of the above pattern in our benchmarks.

Our detailed proof of correctness is evidence that our core calculus is a good basis for type systems and static analyses for languages with async-finish parallelism, and tractable proofs of correctness. We leave further investigation of the precision of the analysis to future work. While our analysis produces an overapproximation of may-happen-in-parallel information, one might use a dynamic analysis that instead gives an underapproximation. The difference between an overapproximation and an underapproximation will shed light on the precision of the overapproximation.

We can straightforwardly extend our calculus to support other features of X10. For example, a worthwhile extension of our calculus would be to model the X10 notion of clocks. Another idea is to support computation with multiple places by changing trees of the form s to be of the form $\langle P, s\rangle$ where P is a place. A tree $\langle P, s\rangle$ means that statement s is executing on place P. One could then consider refining our analysis by asking whether two statements may happen in parallel on the *same* place. We leave such an analysis to future work.

Acknowledgments. We thank Christian Grothoff, Shu-Yu Guo, Riyaz Haque, and the anonymous reviewers for helpful comments on a draft the paper.

References

[1] Martín Abadi and Gordon D. Plotkin. A model of cooperative threads. In *POPL*, pages 29–40, 2009.

[2] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. In *PPoPP*, pages 183–193, 2007.

[3] Rajkishore Barik. Efficient computation of may-happen-in-parallel information for concurrent Java programs. In *LCPC*, pages 152–169, 2005.

[4] Rajkishore Barik and Vivek Sarkar. Interprocedural load elimination for optimization of parallel programs. In *PACT*, 2009.

[5] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vivek Sarkar, and Christoph Von Praun. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pages 519–538, 2005.

[6] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, pages 258–269, 2002.

[7] Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *Symposium on Testing, Analysis, and Verification*, pages 36–48, 1991.

[8] Atsushi Igarashi, Benjamion Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, pages 132–146, 1999.

[9] Vineet Kahlon. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks. In *LICS*, pages 27–36, 2009.

[10] A. J. Kfoury, Michael A. Arbib, and Robert N. Moll. *A Programming Approach to Computability*. Springer-Verlag, 1982.

[11] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient inference of partial types. *Journal of Computer and System Sciences*, 49(2):306–324, 1994.

[12] Lin Li and Clark Verbrugge. A practical MHP information analysis for concurrent Java programs. In *LCPC*, pages 194–208, 2004.

[13] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In *PPoPP*, pages 129–138, 1993.

[14] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. In *POPL*, pages 327–338, 2007.

[15] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statement that may happen in parallel. In *SIGSOFT FSE*, pages 24–34, 1998.

[16] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing HP information for concurrent Java programs. In *ESEC / SIGSOFT FSE*, pages 338–354, 1999.

[17] Vijay A. Saraswat and Radha Jagadeesan. Concurrent clustered programming. In *CONCUR*, pages 353–367, 2005.

[18] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Inf.*, 19:57–84, 1983.

[19] Christoph von Praun and Thomas R. Gross. Static conflict analyis for multi-threaded object-oriented programs. In *PLDI*, pages 115–128, 2003.

[20] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

Appendix A: Proof of Theorem 1

(Deadlock freedom) For every state (p, A, T), either $T = \surd$ or there exists A', T' such that $(p, A, T) \rightarrow (p, A', T')$.

Proof. We proceed by induction on T. We have four cases. If $T \equiv \surd$, then the result is immediate.

If $T \equiv \langle s \rangle$, then we have from Rules (7)–(14) that there exists A', T' such that $(p, A, T) \rightarrow (p, A', T')$.

If $T \equiv (T_1 \rhd T_2)$, then from the induction hypothesis we have that either $T_1 = \surd$ or there exists A', T_1' such that $(p, A, T_1) \rightarrow (p, A', T_1')$. If $T_1 = \surd$, then (p, A, T) can take a step by Rule (1). If there exists A', T_1' such that $(p, A, T_1) \rightarrow (p, A', T_1')$, then (p, A, T) can take a step by Rule (2).

If $T \equiv (T_1 \parallel T_2)$, then from the induction hypothesis we have that either $T_1 = \surd$ or there exists A', T_1' such that $(p, A, T_1) \rightarrow (p, A', T_1')$, and we have that either $T_2 = \surd$ or there exists A', T_2' such that $(p, A, T_2) \rightarrow (p, A', T_2')$. In all four cases, one of Rules (3)–(6) applies to enable (p, A, T) to take a step. This completes the proof of progress. □

Appendix B: Proof of Theorem 2

8.1 A Lemma about the Helper Functions

We begin with a lemma that states 19 useful properties of $symcross$, $Lcross$, $Scross$, $Tcross$, $Slabels$, $FSlabels$, and $FTlabels$.

LEMMA 7. *1.* $symcross(A, B) = symcross(B, A)$
2. If $A' \subseteq A$ *and* $B' \subseteq B$ *then* $symcross(A', B') \subseteq symcross(A, B)$.
3. $symcross(A, C) \cup symcross(B, C) = symcross(A \cup B, C)$

4. $Lcross(l, A \cup B) = Lcross(l, A) \cup Lcross(l, B)$
5. $Scross_p(s, A \cup B) = Scross_p(s, A) \cup Scross_p(s, B)$
6. $Scross_p(s_1, Slabels_p(s_2)) = Scross_p(s_2, Slabels_p(s_1))$
7. $Tcross_p(T, A \cup B) = Tcross_p(T, A) \cup Tcross_p(T, B)$
8. $Tcross_p(T_1, Tlabels_p(T_2)) = Tcross_p(T_2, Tlabels_p(T_1))$
9. $Tcross_p(\surd, A) = \emptyset$
10. If $R' \subseteq R$ *then* $Tcross_p(T, R') \subseteq Tcross_p(T, R)$.
11. $Slabels_p(s_a \,.\, s_b) = Slabels_p(s_a) \cup Slabels_p(s_b)$
12. $FSlabels(s) \subseteq Slabels_p(s)$
13. $FTlabels(T) \subseteq Tlabels_p(T)$
14. $symcross(FTlabels(T_1), FTlabels(T_2)) \subseteq Tcross_p(T_1, Tlabels_p(T_2))$
15. If $(p, A, T) \rightarrow (p, A', T')$ *then* $Tlabels_p(T') \subseteq Tlabels_p(T)$.

16. If $Slabels_p(s) = \{l\} \cup Slabels_p(k)$ *then* $Scross_p(s, R) = Lcross(l, R) \cup Scross_p(k, R)$.
17. If $Slabels_p(s) = \{l\} \cup Slabels_p(s_1) \cup Slabels_p(s_2)$ *then* $Scross_p(s, R) = Lcross(l, R) \cup Scross_p(s_1, R) \cup Scross_p(s_2, R)$.
18. $Tcross_p(\langle s \rangle, R) = Scross_p(s, R)$
19. If $Tlabels_p(T) = Tlabels_p(T_1) \cup Tlabels_p(T_2)$ *then* $Tcross_p(T, R) = Tcross_p(T_1, R) \cup Tcross_p(T_2, R)$.

Proof.

1. By examining the definition of $symcross()$ we see this is trivially true.
2. We also see that this is true by the definition of $symcross()$.
3. From Lemma (7.2) we have
   1) $symcross(A, C) \subseteq symcross(A \cup B, C)$ and
   2) $symcross(B, C) \subseteq symcross(A \cup B, C)$, therefore giving us 3) $symcross(A, C) \cup symcross(B, C) \subseteq symcross(A \cup B, C)$. Suppose we have $l \in A \cup B$ and $l' \in C$. This implies that $l \in A \vee l \in B$. If $l \in A$ then $(l, l') \in symcross(A, C)$. If $l \in B$ then $(l, l') \in symcross(B, C)$. Thus we have that $symcross(A \cup B, C) \subseteq symcross(A, C) \cup symcross(B, C)$, which with 3) gives us our conclusion.
4. We unfold $Lcross()$, then apply (7.1), (7.3) and finally can use the definition $Lcross()$ again to reach our conclusion.
5. We unfold $Scross_p()$, then apply (7.1), (7.3) and finally can use the definition $Scross_p()$ again to reach our conclusion.
6. We unfold the definition of $Scross_p()$, apply (7.1) and finally apply the definition of $Scross_p()$ to reach our conclusion.
7. We unfold $Tcross_p()$, then apply (7.1), (7.3) and finally use the definition of $Tcross_p()$ once again to get our conclusion.
8. We unfold the definition of $Tcross_p()$, apply (7.1) and finally apply the definition of $Tcross_p()$ to reach our conclusion.
9. Unfolding the definition of $Tcross_p()$ and then $Tlabels_p()$ gives us $Tcross_p(\surd, A) = symcross(\emptyset, A)$. From the definition of $symcross()$ we have our conclusion.
10. Let us unfold the definition of $Tcross_p()$ and then apply (7.2) to reach our conclusion.

11. Let us perform induction on s_a. This gives us seven cases to examine.
If $s_a \equiv skip^l$ then from the definition of . we have $s_a \ . \ s_b = skip^l \ s_b$ and from Rule (16) $Slabels_p(s_a \ . \ s_b) = \{l\} \cup Slabels_p(s_b)$. From Rule (15) we have $Slabels_p(s_a) = \{l\}$. From here we can use substitution to reach our conclusion.
If $s_a \equiv skip^l \ s_1$ then from the definition of . we have that $s_a \ . \ s_b = skip^l \ (s_1 \ . \ s_b)$. Using Rule (16) we have $Slabels_p(s_a \ . \ s_b) = \{l\} \cup Slabels_p(s_1 \ . \ s_b)$. Using the induction hypothesis we have that
$Slabels_p(s_1 \ . \ s_b) = Slabels_p(s_1) \cup Slabels_p(s_b)$. We may now substitute that in and use Rule (16) to get our conclusion $Slabels_p(s_a \ . \ s_b) = Slabels_p(skip^l \ s_1) \cup Slabels_p(s_b) = Slabels_p(s_a) \cup Slabels_p(s_b)$.
If $s_a \equiv a[d] =^l e; \ s_1$ then we proceed using similar reasoning as the previous case.
If $s_a \equiv while^l \ (a[d] \neq 0) \ s_1 \ s_2$ then from the definition of . we have $s_a \ . \ s_b = while^l \ (a[d] \neq 0) \ s_1 \ (s_2 \ . \ s_b)$. From Rule (18) we have $Slabels_p(s_a \ . \ s_b) = \{l\} \cup Slabels_p(s_1) \cup Slabels_p(s_2 \ . \ s_b)$. Using the induction hypothesis we get $Slabels_p(s_2 \ . \ s_b) = Slabels_p(s_2) \cup Slabels_p(s_b)$. We may now substitute and use Rule (18) we get $Slabels_p(s_a \ . \ s_b) = Slabels_p(while^l(a[d] \neq 0) \ s_1 \ s_2) \cup Slabels_p(s_b) = Slabels_p(s_a) \cup Slabels_p(s_b)$
If $s_a \equiv async^l \ s_1 \ s_2$ then we may proceed using similar logic as the previous case.
If $s_a \equiv finish^l \ s_1 \ s_2$ then we may proceed using similar logic as the previous case.
If $s_a \equiv f_i()^l \ k$ then from the definition of . we have $s_a \ . \ s_b = f_i()^l \ (s_1 \ . \ s_b)$. From Rule (21) we have $Slabels_p(s_a \ . \ s_b) = \{l\} \cup Slabels_p(s_i) \cup Slabels_p(k \ . \ s_b)$ where $p(f_i) = s_i$. From the induction hypothesis we have that $Slabels_p(k \ . \ s_b) = Slabels_p(k) \cup Slabels_p(s_b)$. We substitute and use Rule (21) to get $Slabels_p(s_a \ . \ s_b) = Slabels_p(f_i()^l \ k) \cup Slabels_p(s_b) = Slabels_p(s_a) \cup Slabels_p(s_b)$.
12. Let us perform case analysis on s. As we examine each case with the definitions of $FSlabels()$ and $Slabels_p()$ we see that the conclusion is obvious.
13. Let us perform induction on T. This gives us four cases.
If $T \equiv \surd$ then examining the definitions we see $FTlabels(\surd) = Tlabels_p(\surd)$. The conclusion is obviously true.
If $T \equiv T_1 \rhd T_2$ then $FTlabels(T) = FTlabels(T_1)$ and $Tlabels_p(T) = Tlabels_p(T_1) \cup Tlabels_p(T_2)$. From the induction hypothesis we have that $FTlabels(T_1) \subseteq Tlabels_p(T_1)$ and thus we can see that our conclusion is true.
If $T \equiv T_1 \parallel T_2$ then
$FTlabels(T) = FTlabels(T_1) \cup FTlabels(T_2)$ and
$Tlabels_p(T) = Tlabels_p(T_1) \cup Tlabels_p(T_2)$. From the induction hypothesis we have that $FTlabels(T_1) \subseteq Tlabels_p(T_1)$ and $FTlabels(T_2) \subseteq Tlabels_p(T_2)$. From here it is easy to reach our conclusion.
If $T \equiv \langle s \rangle$ then examining the definitions we see
$FTlabels(T) = FSlabels(s)$ and
$Tlabels_p(T) = Slabels_p(s)$. From (7.12) we reach our conclusion.
14. From (7.13) we have 1) $FTlabels(T_1) \subseteq Tlabels_p(T_1)$ and 2) $FTlabels(T_2) \subseteq Tlabels_p(T_2)$. From unfolding $Tcross()$ we have 3) $Tcross_p(T_1, Tlabels_p(T_2)) = symcross(Tlabels_p(T_1), Tlabels_p(T_2))$. Using (7.2) with 1) and 2) gives us
4) $symcross(FTlabels(T_1), FTlabels(T_2)) \subseteq symcross(Tlabels_p(T_1), Tlabels_p(T_2))$. From 3) and 4) we have our conclusion.
15. Let us perform induction on T. This gives us four cases.
If $T \equiv \surd$ then we do not take a step.
If $T \equiv T_1 \rhd T_2$ then there are two rules by which we may take a step.
Suppose we step by Rule (1) and $T' = T_2$. From the definition of $Tlabels_p()$ we have $Tlabels_p(T) = Tlabels_p(T_1) \cup Tlabels_p(T_2)$ and $Tlabels_p(T') = Tlabels_p(T_2)$. We see from this that the conclusion is true.
Suppose we step by Rule (2) then we have 1) $T' = T_1' \rhd T_2$ and 2) $(p, A, T_1) \rightarrow (p, A', T_1')$. Unfolding the definition of $Tlabels_p()$ we have 3) $Tlabels_p(T) = Tlabels_p(T_1) \cup Tlabels_p(T_2)$ and
4) $Tlabels_p(T') = Tlabels_p(T_1') \cup Tlabels_p(T_2)$. From the induction hypothesis we have that 3) $Tlabels_p(T_1') \subseteq Tlabels_p(T_1)$ and from here we easily may arrive at the conclusion.
If $T \equiv T_1 \parallel T_2$ then there are four rules by which we can step.
Suppose we step by Rule (3) then we may use similar logic as the case where $T \equiv T_1 \rhd T_2$ and we step by Rule (1).
Suppose we step by Rule (4) then we proceed using similar logic as the previous case.
Suppose we step by Rule (5) then we may use similar logic as the case where $T \equiv T_1 \rhd T_2$ and we step by Rule (2).
Suppose we step by Rule (6) then we may proceed using similar logic as the previous case.
If $T \equiv \langle s \rangle$ then we now perform induction on s to give us an additional seven cases.
If $s \equiv skip^l$ then we step by Rule (7) and $T' = \surd$. From the definition of $Tlabels()$ we have $Tlabels_p(T') = \emptyset$ and $Tlabels_p(T) = \{l\}$. Thus we see that the conclusion is true.
If $s \equiv skip^l \ s_1$ then we step by Rule (8) and $T' = \langle s_1 \rangle$. We see that by the definition of $Tlabels()$ that $Tlabels_p(T') = Slabels_p(s_1)$ and $Tlabels_p(T) = \{l\} \cup Slabels_p(s_1)$. We now can easily arrive at the conclusion.
If $s \equiv a[d] =^l e; \ s_1$ then we step by Rule (9) and proceed using similar reasoning as the previous case.
If $s \equiv while^l \ (a[d] \neq 0) \ s_1 \ s_2$ then there are two rules by which we may take a step.
Suppose we step by Rule (10) then $T' = \langle s_2 \rangle$. From the definition of $Tlabels_p()$ we have $Tlabels_p(T') = Slabels_p(s_2)$ and $Tlabels_p(T) = \{l\} \cup Slabels_p(s_1) \cup Slabels_p(s_2)$. The conclusion is obvious.
Suppose we step by Rule (11) then $T' = \langle s_1 \ . \ while^l \ (a[d] \neq 0) \ s_1 \ s_2 \rangle$. From the definition of $Tlabels_p()$ and (7.11) we have $Tlabels_p(T') = Slabels_p(s_1) \cup \{l\} \cup Slabels_p(s_1) \cup Slabels_p(s_2) = Slabels_p(s)$ and $Tlabels_p(T) = Slabels_p(s)$. The conclusion is obviously true.
If $s \equiv async^l \ s_1 \ s_2$ then we step by Rule (12) and $T' = \langle s_1 \rangle \parallel \langle s_2 \rangle$. Using the definition of $Tlabels_p()$ we have $Tlabels_p(T') = Slabels_p(s_1) \cup Slabels_p(s_2)$ and $Tlabels_p(T) = \{l\} \cup Slabels_p(s_1) \cup Slabels_p(s_2)$. The conclusion is now obvious.
If $s \equiv finish^l \ s_1 \ s_2$ then we step by Rule (13) and $T' = \langle s_1 \rangle \rhd \langle s_2 \rangle$. From the definition of $Tlabels_p()$
we get $Tlabels_p(T') = Slabels_p(s_1) \cup Slabels_p(s_2)$ and $Tlabels_p(T) = \{l\} \cup Slabels_p(s_1) \cup Slabels_p(s_2)$. The conclusion is easily reached from here.
If $s \equiv f_i()^l \ s_1$ then we step by Rule (14) and $T' = \langle s_i \ . \ s_1 \rangle$ where $p(f_i) = s_i$. From the definition of $Tlabels_p()$ we get $Tlabels_p(T') = Slabels_p(s_i \ . \ s_1)$ and $Tlabels_p(T) = \{l\} \cup Slabels_p(s_i) \cup Slabels_p(s_1)$. From (7.11) we have $Slabels_p(s_i \ . \ s_1) = Slabels_p(s_i) \cup Slabels_p(s_1)$. From here we can easily arrive at our conclusion.
16. Let us unfold the definition of $Scross()$ to get
1) $Scross_p(s, R) = symcross(Slabels_p(s), R)$. We may now substitute to get 2) $Scross_p(s, R) = symcross(\{l\} \cup$

$Slabels_p(k), R)$. Let us apply the (7.3) to get
3) $Scross_p(s, R) = symcross(\{l\}, R) \cup$
$symcross(Slabels_p(k), R)$. We may now use the definitions of $Lcross()$ and $Scross_p()$ achieve our conclusion.

17. Let us use the definition of $Scross()$ to get 1) $Scross_p(s, R) = symcross(Slabels_p(s), R)$. We may substitute to get
2) $Scross_p(s, R) = symcross(\{l\} \cup Slabels_p(s_1) \cup$
$Slabels_p(s_2), R)$. Using (7.3) we can get 3) $Scross_p(s, R) = symcross(\{l\}, R) \cup symcross(Slabels_p(s_1), R) \cup$
$symcross(Slabels_p(s_2), R)$. We may now use the definition of $Lcross()$ and $Scross_p()$ to arrive at our conclusion.
18. Unfolding $Tcross()$ gives us
1) $Tcross_p(\langle s \rangle, R) = symcross(Tlabels_p(s), R)$. We unfold $Tlabels_p()$ to get
2) $Tcross_p(\langle s \rangle, R) = symcross(Slabels_p(s), R)$. We apply the definition of $Scross_p()$ to get our conclusion
$Tcross_p(\langle s \rangle, R) = Scross_p(s, R)$.
19. Let us unfold the definition of $Tcross_p()$ to get
1) $Tcross_p(T, R) = symcross(Tlabels_p(T), R)$. Substituting the premise in 1) gives us 2) $Tcross_p(T, R) =$
$symcross(Tlabels_p(T_1) \cup Tcross_p(T_2), R)$. We apply (7.3) on 2) to get 3) $Tcross_p(T, R) =$
$symcross(Tlabels_p(T_1), R) \cup symcross(Tlabels_p(T_2), R)$. Finally we apply the definition of $Tcross()$ on 3) to get our conclusion,
$Tcross_p(T, R) = Tcross_p(T_1, R) \cup Tcross_p(T_2, R)$.

□

8.2 Unique Typing

We first observe that any label set R and statement s will always uniquely determine M and O. We will use this property often to show that types are equal.

LEMMA 8. *If $p, E, R \vdash s : M_1, O_1$ and $p, E, R \vdash s : M_2, O_2$ then $M_1 = M_2$ and $O_1 = O_2$.*

Proof. Let us perform induction on s and examine the seven cases.

If $s \equiv skip^l$ the conclusion is immediately obvious from Rule (50).

If $s \equiv skip^l\ s_1$ then from Rule (51) we have 1) $p, E, R \vdash s_1 : M_1', O_1'$, 2) $M_1 = Lcross(l, R) \cup M_1'$, 3) $O_1 = O_1'$, 4) $p, E, R \vdash s_1 : M_2', O_2'$, 5) $M_2 = Lcross(l, R) \cup M_2'$ and 6) $O_2 = O_2'$. Using the induction hypothesis on 1) and 4) we get 7) $M_1' = M_2'$ and 8) $O_1' = O_2'$. Applying some substitution among 2),3),5),6),7) and 8) we arrive at our conclusion $M_1 = M_2$ and $O_1 = O_2$.

If $s \equiv a[d] =^l e;\ s_1$ then we may proceed using similar reasoning as the previous case.

If $s \equiv while^l\ (a[d] \neq 0)\ s_1\ s_2$ then from Rule (53) we have 1) $p, E, R \vdash s_1 : M_1', O_1'$, 2) $p, E, O_1' \vdash s_2 : M_1'', O_1''$, 3) $M_1 = Lcross(l, O_1') \cup Scross_p(s_1, O_1') \cup M_1' \cup M_1''$, 4) $O_1 = O_1''$, 5) $p, E, R \vdash s_1 : M_2', O_2'$, 6) $p, E, O_2' \vdash s_2 : M_2'', O_2''$, 7) $M_2 = Lcross(l, O_2') \cup Scross_p(s_1, O_2') \cup M_2' \cup M_2''$ and 8) $O_2 = O_2''$. Let us apply the induction hypothesis on 1) and 5) to get 9) $M_1' = M_2'$ and 10) $O_1' = O_2'$. From 10) we are able to apply the induction hypothesis on 2) and 6) to get 11) $M_1'' = M_2''$ and 12) $O_1'' = O_2''$. Using substitution with 9),10),11) and 12) in 3),4),7) and 8) we get our conclusion $M_1 = M_2$ and $O_1 = O_2$.

If $s \equiv async^l\ s_1\ s_2$ then from Rule (54) we have
1) $p, E, Slabels_p(s_2) \cup R \vdash s_1 : M_1', O_1'$, 2) $p, E, Slabels_p(s_1) \cup R \vdash s_2 : M_1'', O_1''$, 3) $M_1 = Lcross(l, R) \cup M_1' \cup M_1''$, 4) $O_1 = O_1''$, 5) $p, E, Slabels_p(s_2) \cup R \vdash s_1 : M_2', O_2'$, 6) $p, E, Slabels_p(s_1) \cup R \vdash s_2 : M_2'', O_2''$, 7) $M_2 = Lcross(l, R) \cup M_2' \cup M_2''$ and 8) $O_2 = O_2''$. We may apply the induction hypothesis on 1) and 5) and 2) and 6) to get 9) $M_1' = M_2'$, 10) $O_1' = O_2'$, 11) $M_1'' = M_2''$ and 12) $O_1'' = O_2''$. Substituting 9),10),11) and 12) in 3),4),7) and 8) we get our conclusion $M_1 = M_2$ and $O_1 = O_2$.

If $s \equiv finish^l\ s_1\ s_2$ then we may proceed using similar reasoning as the previous case.

If $s \equiv f_i()^l\ k$ then from Rule (56) we have 1) $E(f_i) = (M_i, O_i)$, 2) $p, E, R \cup O_i \vdash s_1 : M_k', O_k'$, 3) $M_1 = Lcross(l, R) \cup symcross(Slabels_p(p(f_i), R) \cup M_i \cup M_k'$, 4) $O_1 = O_k'$, 5) $p, E, R \cup O_i \vdash s_k : M_k'', O_k''$, 6) $M_2 = Lcross(l, R) \cup symcross(Slabels_p(p(f_i), R) \cup M_i \cup M_k''$ and 7) $O_2 = O_k''$. From applying the induction hypothesis with 2) and 5) we get 8) $M_k' = M_k''$ and 9) $O_k' = O_k''$. Substituting 6) and 8) in 3) we get 10) $M_1 = M_2$. Substituting 7) and 9) in 4) we obtain 11) $O_1 = O_2$. From 10) and 11) we have our conclusion. □

The next lemma is similar to the previous lemma, but works for trees: any R and T uniquely determines M.

LEMMA 9. *If $p, E, R \vdash T : M$ and $p, E, R \vdash T : M'$ then $M = M'$.*

Proof. Let us perform induction on T. There are four cases.

If $T \equiv \surd$ then from Rule (49) we have 1) $M = \emptyset$ and 2) $M' = \emptyset$. It is obvious that $M = M'$.

If $T \equiv T_1 \rhd T_2$ then from Rule (46) we have 1) $p, E, R \vdash T_1 : M_1$, 2) $p, E, R \vdash T_2 : M_2$, 3) $M = M_1 \cup M_2$, 4) $p, E, R \vdash T_1 : M_1'$, 5) $p, E, R \vdash T_2 : M_2'$ and 6) $M' = M_1' \cup M_2'$. From the induction hypothesis applied to 1) and 4) and to 2) and 5) we get 7) $M_1 = M_1'$ and 8) $M_2 = M_2'$. From 3),6),7) and 8) we see $M = M'$.

If $T \equiv T_1 \parallel T_2$ then from Rule (47) we have
1) $p, E, Tlabels_p(T_2) \cup R \vdash T_1 : M_1$, 2) $p, E, Tlabels_p(T_1) \cup R \vdash T_2 : M_2$, 3) $M = M_1 \cup M_2$, 4) $p, E, Tlabels_p(T_2) \cup R \vdash T_1 : M_1'$, 5) $p, E, Tlabels_p(T_1) \cup R \vdash T_2 : M_2'$ and 6) $M' = M_1' \cup M_2'$. We use the induction hypothesis on 1) and 4) and on 2) and 5) to get 7) $M_1 = M_1'$ and 8) $M_2 = M_2'$. From 3),6),7) and 8) we get $M = M'$.

If $T \equiv \langle s \rangle$ then from Rule (48) we have 1) $p, E, R \vdash s : M_s, O_s$, 2) $M = M_s$, 3) $p, E, R \vdash s : M_s', O_s'$, 4) $M' = M_s'$. From Lemma (8) applied to 1) and 3) we have 5) $M_s = M_s'$. From 2),4) and 5) we have $M = M'$. □

8.3 Principal Typing

The following lemma shows that if a statement is typable with a set R, then it will also be typable with a set R'. This is convenient for showing the existence of a type when we perform induction in the proofs of later lemmas; once we have such a type, we can then use the unique-typing lemmas to relate the type to other types.

LEMMA 10. *If $p, E, R \vdash s : M, O$ then there exists M' and O' such that $p, E, R' \vdash s : M', O'$.*

Proof. Let us perform induction on s. This gives us seven cases to examine.

If $s \equiv skip^l$ then from Rule (50) we let $M = Lcross(l, R')$ and $O = R'$.

If $s \equiv skip^l\ s_1$ then from Rule (51) we have 1) $p, E, R \vdash s_1 : M_1, O_1$. Using the induction hypothesis with 1) we have that there exists M_1' and O_1' such that 2) $p, E, R' \vdash s_1 : M_1', O_1'$. Then by Rule (51) we let $M = Lcross(l, R') \cup M_1$ and $O = O_1'$.

If $s \equiv a[d] =^l e;\ s_1$ then we proceed using similar logic as the previous case.

If $s \equiv while^l\ (a[d] \neq 0)\ s_1\ s_2$ then by Rule (53) we have 1) $p, E, R \vdash s_1 : M_1, O_1$ and 2) $p, E, O_1 \vdash s_2 : M_2, O_2$. Using the induction hypothesis with 1) and 2) we have that there exists M_1',M_2',O_1' and O_2' such that 3) $p, E, R' \vdash s_1 : M_1', O_1'$ and 4) $p, E, O_1' \vdash s_2 : M_2', O_2'$. Then from Rule (53) we let $M = Lcross(l, R') \cup Scross_p(s_1, O_1') \cup M_1 \cup M_2$ and $O = O_2'$.

If $s \equiv async^l\ s_1\ s_2$ then by Rule (54) we have
1) $p, E, Slabels_p(s_2) \cup R \vdash s_1 : M_1, O_1$ and
2) $p, E, Slabels_p(s_1) \cup R \vdash s_2 : M_2, O_2$. We may use the induction hypothesis with 1) and 2) go get that there exists M_1',M_2',O_1' and O_2' such that 3) $p, E, Slabels_p(s_2) \cup R' \vdash s_1 : M_1', O_1'$ and 4) $p, E, Slabels_p(s_1) \cup R' \vdash s_2 : M_2', O_2'$. Then from Rule (54) we let $M = Lcross(l, R') \cup M_1' \cup M_2'$ and $O = O_2'$.

If $s \equiv finish^l\ s_1\ s_2$ then from Rule (55) we have 1) $p, E, R \vdash s_1 : M_1, O_1$ and 2) $p, E, R \vdash s_2 : M_2, O_2$. We use the induction hypothesis with 1) and 2) to get that there exists M_1',M_2',O_1' and O_2' such that 3) $p, E, R' \vdash s_1 : M_1', O_1'$ and 4) $p, E, R' \vdash s_2 : M_2', O_2$. Then from Rule (55) we let $M = Lcross(l, R') \cup M_1' \cup M_2'$ and $O = O_2'$.

If $s \equiv f_i()\ k$ then by Rule (56) we have 1) $E(f_i) = (M_i, O_i)$ and 2) $p, E, R \cup O_i \vdash k : M', O'$. Using the induction hypothesis with 2) we have that there exists M'' and O'' such that 3) $p, E, R' \cup O_i \vdash k : M'', O''$. With 1) and 3) we may apply Rule (56) with $M' = Lcross(l, R') \cup symcross(Slabel_p(p(f_i)), R') \cup M_i \cup M''$ and $O' = O''$ to reach our conclusion. □

Again, we need a similar lemma for execution trees.

LEMMA 11. *If $p, E, R \vdash T : M$ then there exists M' such that $p, E, R' \vdash T : M'$.*

Proof. Let us perform induction on T. There are four cases.

If $T \equiv \surd$ then by Rule (49) we let $M' = \emptyset$.

If $T \equiv T_1 \rhd T_2$ then from Rule (46) we have 1) $p, E, R \vdash T_1 : M_1$ and 2) $p, E, R \vdash T_2 : M_2$. We may use the induction hypothesis with 1) and 2) to get that there exists M_1' and M_2' such that 3) $p, E, R' \vdash T_1 : M_1'$ and 4) $p, E, R' \vdash T_2 : M_2'$. By Rule (46) we let $M' = M_1' \cup M_2'$.

If $T \equiv T_1 \parallel T_2$ the we may use similar logic as with the previous case.

If $T \equiv \langle s \rangle$ then from Rule (48) we have 1) $p, E, R \vdash s : M, O$. We use Lemma (10) with 1) and we have that there exists M'' and O'' such that $p, E, R' \vdash s : M'', O''$. Then by Rule (48) we let $M' = M''$. □

The following lemma is our principal typing lemma for statements. Intuitively, we have a mapping π from a typing to a set of typings, and if we produce a typing $\mathcal{T}$ for a statement s with $R = \emptyset$, then $\pi(\mathcal{T})$ are exactly all the possible typings of s. Our mapping π consists simply of creating appropriate set unions. The idea is that for a judgment $p, E, R \vdash s : M, O$, the statements with labels in R may still be running when s terminates so if we have a judgment $p, E, \emptyset \vdash s : M', O'$, then O must be the union of R and O'. Also, those statement with labels in R may run in parallel with any statement in s, hence M is the union of $Scross_p(s, R)$ and M'.

LEMMA 12. *$p, E, R \vdash s : M, O$ if and only if there exists M' and O' such that $p, E, \emptyset \vdash s : M', O'$ and $M = Scross_p(s, R) \cup M'$ and $O = R \cup O'$.*

Proof. $\Rightarrow$) We may use Lemma (10) with the premise to get that there exists M' and O' such that $p, E, \emptyset \vdash s : M', O'$. We next perform induction on s. We have seven cases to examine and show that $M = Scross_p(s, R) \cup M'$ and $O = R \cup O'$.

If $s \equiv skip^l$ then by Rule (50) and using the definition of $Lcross()$ we have 1) $M' = \emptyset$, 2) $O' = \emptyset$, 3) $M = symcross(\{l\}, R)$, and 4) $O = R$. Using Rule (15) and the definition of $Scross()$ we have 5) $M = symcross(Slabels(s), R) = Scross_p(s, R)$. We can now easily see from 1),2),4) and 5) that $M = Scross_p(s, R) \cup M'$ and $O = R \cup O'$.

If $s \equiv skip^l\ s_1$ then by Rule (51) and the definition of $Lcross()$ we have 1) $p, E, \emptyset \vdash s_1 : M_1', O_1'$, 2) $M' = Lcross(l, \emptyset) \cup M_1' = M_1'$, 3) $O' = O_1'$ 4) $p, E, R \vdash s_1 : M_1, O_1$, 5) $M = Lcross(l, R) \cup M_1$, and 6) $O = O_1$. Using the induction hypothesis on 1) and 4) we get 7) $M_1 = Scross_p(s_1, R) \cup M_1'$ and 8) $O_1 = R \cup O_1'$. Let us substitute 7) in 5) to get 9) $M = Lcross(l, R) \cup Scross_p(s_1, R) \cup M_1'$. By using Rule (16), Lemma (7.16) and 2) we get $M = Scross_p(s, R) \cup M_1' = Scross_p(s, R) \cup M'$. Finally using 3), 6), and 8) we may perform substitution to get $O = O_1 = R \cup O_1' = R \cup O'$ and thus we have our conclusion.

If $s \equiv a[d] =^l e;\ s_1$ we may proceed using similar reasoning as the previous case.

If $s \equiv while^l\ (a[d] \neq 0)\ s_1\ s_2$ then by Rule (53) we have 1) $p, E, \emptyset \vdash s_1 : M_1', O_1'$, 2) $p, E, O_1' \vdash s_2 : M_2', O_2'$, 3) $M' = Lcross(l, O_1') \cup Scross_p(s_1, O_1') \cup M_1' \cup M_2'$, 4) $O' = O_2'$, 5) $p, E, R \vdash s_1 : M_1, O_1$, 6) $p, E, O_1 \vdash s_2 : M_2, O_2$, 7) $M = Lcross(l, O_1) \cup Scross_p(s_1, O_1) \cup M_1 \cup M_2$, and 8) $O = O_2$. From the induction hypothesis and Lemma (8) applied to 2),5) and 6) we have 9) $p, E, \emptyset \vdash s_2 : M_2'', O_2''$, 10) $M_2' = Scross_p(s_2, O_1') \cup M_2''$, 11) $O_2' = O_1' \cup O_2''$, 12) $M_1 = Scross_p(s_1, R) \cup M_1'$, 13) $O_1 = R \cup O_1'$, 14) $M_2 = Scross_p(s_2, O_1) \cup M_2''$ and 15) $O_2 = O_1 \cup O_2''$. Substituting 10) in 3); 12),13) and 14) in 7); 11) in 4); and 13) and 15) in 8) yields 16) $M' = Lcross(l, O_1') \cup Scross_p(s_1, O_1') \cup M_1' \cup Scross_p(s_2, O_1') \cup M_2''$, 17) $M = Lcross(l, R \cup O_1') \cup Scross_p(s_1, R \cup O_1') \cup Scross_p(s_1, R) \cup M_1' \cup Scross_p(s_2, R \cup O_1') \cup M_2''$, 18) $O' = O_1' \cup O_2''$ and 19) $O = R \cup O_1' \cup O_2''$. Using Lemma (7.4) and (7.5) on 17) we have 20) $M = Lcross(l, R) \cup Scross_p(s_1, R) \cup Scross_p(s_2, R) \cup Lcross(l, O_1') \cup Scross_p(s_1, O_1') \cup Scross_p(s_2, O_1') \cup M_1' \cup M_2''$.
Using Lemma (7.17) with Rule (18) on 20) we get 21) $M = Scross_p(s, R) \cup Lcross(l, O_1') \cup Scross_p(s_1, O_1') \cup Scross_p(s_2, O_1') \cup M_1' \cup M_2''$. We may substitute 16) in 21) to get $M = Scross_p(s, R) \cup M'$ and then substitute 18) in 19) to get $O = R \cup O'$.

If $s \equiv async^l\ s_1\ s_2$ then by Rule (54) we have
1) $p, E, Slabels_p(s_2) \vdash s_1 : M_1', O_1'$, 2) $p, E, Slabels_p(s_1) \vdash s_2 : M_2', O_2'$, 3) $M' = Lcross(l, \emptyset) \cup M_1' \cup M_2' = M_1' \cup M_2'$, 4) $O' = O_2'$, 5) $p, E, Slabels_p(s_2) \cup R \vdash s_1 : M_1, O_1$, 6) $p, E, Slabels_p(s_1) \cup R \vdash s_2 : M_2, O_2$, 7) $M = Lcross(l, R) \cup M_1 \cup M_2$, and 8) $O = O_2$. We may apply the induction hypothesis and Lemma (8) to 1),2),5) and 6) to get 9) $p, E, \emptyset \vdash s_1 : M_1'', O_1''$,
10) $p, E, \emptyset \vdash s_2 : M_2'', O_2''$,
11) $M_1' = Scross_p(s_1, Slabels_p(s_2)) \cup M_1''$,
12) $M_2' = Scross_p(s_2, Slabels_p(s_1)) \cup M_2''$,
13) $O_2' = Slabels_p(s_1) \cup O_2''$,
14) $M_1 = Scross_p(s_1, Slabels_p(s_2) \cup R) \cup M_1''$, 15) $M_2 = Scross_p(s_2, Slabels_p(s_1) \cup R) \cup M_2''$ and
16) $O_2 = Slabels_p(s_1) \cup R \cup O_2''$. We now substitute 11) and 12) in 3); 14) and 15) in 7) to get 17) $M' = Scross_p(s_1, Slabels_p(s_2)) \cup M_1'' \cup Scross_p(s_2, Slabels_p(s_1)) \cup M_2''$ and
18) $M = Lcross(l, R) \cup Scross_p(s_1, Slabels_p(s_2) \cup R) \cup M_1'' \cup Scross_p(s_2, Slabels_p(s_1) \cup R) \cup M_2''$. Using Lemma (7.5) on 18) then substituting in 17) we get 19) $M = Lcross(l, R) \cup Scross_p(s_1, R) \cup Scross_p(s_2, R) \cup M'$. We now apply
Lemma (7.17) with Rule (19) to get $M = Scross_p(s, R) \cup M'$. Finally from 4),8),13) and 16) we have $O = R \cup O'$.

If $s \equiv finish^l\ s_1\ s_2$ then by Rule (55) we have 1) $p, E, \emptyset \vdash s_1 : M_1', O_1'$, 2) $p, E, \emptyset \vdash s_2 : M_2', O_2'$, 3) $M' = Lcross(l, \emptyset) \cup M_1' \cup M_2' = M_1' \cup M_2'$, 4) $O' = O_2'$, 5) $p, E, R \vdash s_1 : M_1, O_1$, 6) $p, E, R \vdash s_2 : M_2, O_2$, 7) $M = Lcross(l, R) \cup M_1 \cup M_2$, and 8) $O = O_2$. Let us apply the induction hypothesis with Lemma (8) on 5) and 6) to get 9) $M_1 = Scross_p(s_1, R) \cup M_1'$, 10) $O_1 = R \cup O_1'$, 11) $M_2 = Scross_p(s_2, R) \cup M_2'$, and 12) $O_2 = R \cup O_2'$. We may substitute 3),9) and 11) in 7) to get 13) $M = Lcross(l, R) \cup Scross_p(s_1, R) \cup Scross_p(s_2, R) \cup M'$. Using Lemma (7.17) with Rule (20) we get $M = Scross_p(s, R) \cup M'$. Finally we see from 4),8) and 12) we have $O = R \cup O'$.

If $s \equiv f_i()\ k$ then by Rule (56) we have 1) $E(f_i) = (M_i, O_i)$, 2) $p, E, O_i \vdash k : M'_k, O'_k$, 3) $M' = Lcross(l, \emptyset) \cup symcross(Slabels_p(p(f_i)), \emptyset) \cup M_i \cup M'_k = M_i \cup M'_k$, 4) $O' = O'_k$, 5) $p, E, R \cup O_i \vdash k : M_k, O_k$, 6) $M = Lcross(l, R) \cup symcross(Slabels_p(p(f_i)), R) \cup M_i \cup M_k$ and 7) $O = O_k$. We may apply the induction hypothesis with the premise and 2) and 5) to get that there exists M''_k,M'''_k,O''_k and O'''_k such that 8) $p, E, \emptyset \vdash k : M''_k, O''_k$, 9) $M'_k = Scross_p(k, O_i) \cup M''_k$, 10) $O'_k = O_i \cup O''_k$, 11) $p, E, \emptyset \vdash k : M'''_k, O'''_k$, 12) $M_k = Scross_p(k, R \cup O_i) \cup M'''_k$ and 13) $O_k = R \cup O_i \cup O'''_k$. We apply Lemma (8) with 8) and 11) to get 14) $M''_k = M'''_k$ and 15) $O''_k = O'''_k$. We substitute 9) in 3) to get 16) $M' = M_i \cup Scross_p(k, O_i) \cup M''_k$. Substituting 12) and 14) and $p(f_i) = s_i$ in 6) gives us 17) $M = Lcross(l, R) \cup symcross(Slabels_p(s_i), R) \cup Scross_p(k, R \cup O_i) \cup M_i \cup M''_k$. Applying the definition of $Scross_p()$ and Lemma (7.5 to 17) gives us 18) $M = Lcross(l, R) \cup Scross_p(s_i, R) \cup Scross_p(k, R) \cup Scross_p(k, O_i) \cup M_i \cup M''_k$. We substitute 16) in 18) to get 19) $M = Lcross(l, R) \cup Scross_p(s_i, R) \cup Scross_p(k, R) \cup M'$. Applying Lemma (7.17) with Rule (21) on 19) to get 20) $M = Scross_p(s, R) \cup M'$. Substituting 10) and 15) in 13) gives us 21) $O_k = R \cup O'_k$. We substitute 4) and 7) in 21) to get 22) $O = R \cup O'$. With 20) and 22) we have our conclusion.

$\Leftarrow$) From Lemma (10) and the premise there exists M'' and O'' such that $p, E, R \vdash s : M'', O''$. If we show $M'' = M$ and $O'' = O$ then we will have our conclusion that $p, E, R \vdash s : M, O$. We now will perform induction on s and examine the seven cases and show that $M'' = M$ and $O'' = O$.

If $s \equiv skip^l$ then from Rule (50) we have
1) $M' = Lcross(l, \emptyset) = \emptyset$, 2) $O' = \emptyset$, 3) $M'' = Lcross(l, R)$ and 4) $O'' = R$. Substituting 1) and 2) in the premise gives us 5) $M = Scross_p(s, R)$ and 6) $O = R$. Unfolding the definition of $Scross_p()$ in 5) we have 7) $M = symcross(Slabels_p(s), R)$. From the definitions $Slabels_p()$ and $Lcross()$ applied to 7) we get 8) $M = Lcross(l, R)$. From 3),4),6) and 8) we have $M'' = M$ and $O'' = O$.

If $s \equiv skip^l\ s_1$ then from Rule (51) we have 1) $p, E, \emptyset \vdash s_1 : M'_1, O'_1$, 2) $M' = Lcross(l, \emptyset) \cup M'_1 = M'_1$, 3) $O' = O'_1$, 4) $p, E, R \vdash s_1 : M''_1, O''_1$, 5) $M'' = Lcross(l, R) \cup M''_1$ and 6) $O'' = O''_1$. Let 7) $M_1 = Scross_p(s_1, R) \cup M'_1$ and 8) $O_1 = R \cup O'_1$. Then using the induction hypothesis we have 9) $p, E, R \vdash s_1 : M_1, O_1$. From Lemma (8) on 4) and 9) we have 10) $M_1 = M''_1$ and 11) $O_1 = O''_1$. Substituting 7) and 10) in 5) and 8) and 11) in 6) yields 12) $M'' = Lcross(l, R) \cup Scross_p(s_1, R) \cup M'_1$ and 13) $O'' = R \cup O'_1$. Using Lemma (7.16) with Rule (16) on 12) gives us 14) $M'' = Scross_p(s, R) \cup M'_1$. Substituting 2) and 3) in the premise gives us 15) $M = Scross_p(s, R) \cup M'_1$ and 16) $O = R \cup O'_1$. From 13),14),15) and 16) we see $M'' = M$ and $O'' = O$.

If $s \equiv a[d] =^l e;\ s_1$ then we may use similar reasoning as the previous case.

If $s \equiv while^l\ (a[d] \neq 0)\ s_1\ s_2$ then from Rule (53) we have 1) $p, E, \emptyset \vdash s_1 : M'_1, O'_1$, 2) $p, E, O'_1 \vdash s_2 : M'_2, O'_2$, 3) $M' = Lcross(l, O'_1) \cup Scross_p(s_1, O'_1) \cup M'_1 \cup M'_2$, 4) $O' = O'_2$, 5) $p, E, R \vdash s_1 : M''_1, O''_1$, 6) $p, E, O''_1 \vdash s_2 : M''_2, O''_2$, 7) $M'' = Lcross(l, O''_1) \cup Scross_p(s, O''_1) \cup M''_1 \cup M''_2$ and 8) $O'' = O''_2$. Let 9) $M_1 = Scross_p(s_1, R) \cup M'_1$ and 10) $O_1 = R \cup O'_1$. Then from the induction hypothesis we have 11) $p, E, R \vdash s_1 : M_1, O_1$. Using Lemma (8) on 5) and 11) results in 12) $M_1 = M''_1$ and 13) $O_1 = O''_1$. From Lemma (10) there exists M_2 and O_2 such that 14) $p, E, \emptyset \vdash s_2 : M_2, O_2$. Let 15) $M'''_2 = Scross_p(s_2, O'_1) \cup M_2$, 16) $O'''_2 = O'_1 \cup O_2$, 17) $M''''_2 = Scross_p(s_2, O''_1) \cup M_2$ and 18) $O''''_2 = O''_1 \cup O_2$. We use the induction hypothesis with 14),15) and 16) and 14),17) and 18) to get 19) $p, E, O'_1 \vdash s_2 : M'''_2, O'''_2$ and 20) $p, E, O''_1 \vdash s_2 : M''''_2, O''''_2$. Using Lemma (8) on 2) and 19) and 6) and 20) we get 21) $M'_2 = M'''_2$, 22) $O'_2 = O'''_2$, 23) $M''_2 = M''''_2$ and 24) $O''_2 = O''''_2$. Substituting 15) and 21) in 3) and 16) and 22) in 4) to get 25) $M' = Lcross(l, O'_1) \cup Scross_p(s_1, O'_1) \cup M'_1 \cup Scross_p(s_2, O'_1) \cup M_2$ and 26) $O' = O'_1 \cup O_2$. We apply Lemma (7.17) with Rule (18) on 25) to get 27) $M' = Scross_p(s, O'_1) \cup M'_1 \cup M_2$. We now substitute 26) and 27) in the premise and we get 28) $M = Scross_p(s, R) \cup Scross_p(s, O'_1) \cup M'_1 \cup M_2$ and 29) $O = R \cup O'_1 \cup O_2$. Applying Lemma (7.5) to 28) results in 30) $M = Scross_p(s, R \cup O'_1) \cup M'_1 \cup M_2$. Substituting 9),10),12),13),17) and 23) in 7) and gives us 31) $M'' = Lcross(l, R \cup O'_1) \cup Scross_p(s_1, R \cup O'_1) \cup Scross_p(s_1, R) \cup M'_1 \cup Scross_p(s_2, R \cup O'_1) \cup M_2$. From substituting 10),13),18) and 24) in 8) we get 32) $O'' = R \cup O'_1 \cup O_2$. Using Lemma (7.5) allows us to simplify 31) to 33) $M'' = Lcross(l, R \cup O'_1) \cup Scross_p(s_1, R \cup O'_1) \cup M'_1 \cup Scross_p(s_2, R \cup O'_1) \cup M_2$. Next we apply Lemma (7.17) with Rule (18) to get 34) $M'' = Scross_p(s, R \cup O'_1) \cup M'_1 \cup M_2$. From 30) and 34) we have $M'' = M$ and from 29) and 32) we have $O'' = O$.

If $s \equiv async^l\ s_1\ s_2$ then from Rule (54) we have
1) $p, E, Slabels_p(s_2) \vdash s_1 : M'_1, O'_1$, 2) $p, E, Slabels_p(s_1) \vdash s_2 : M'_2, O'_2$, 3) $M' = Lcross(l, \emptyset) \cup M'_1 \cup M'_2 = M'_1 \cup M'_2$, 4) $O' = O'_2$, 5) $p, E, Slabels_p(s_2) \cup R \vdash s_1 : M''_1, O''_1$, 6) $p, E, Slabels_p(s_1) \cup R \vdash s_2 : M''_2, O''_2$, 7) $M'' = Lcross(l, R) \cup M''_1 \cup M''_2$ and 8) $O'' = O''_2$. From Lemma (10) there exists M'''_1,M'''_2,O'''_1, and O'''_2 such that 9) $p, E, \emptyset \vdash s_1 : M'''_1, O'''_1$ and 10) $p, E, \emptyset \vdash s_2 : M'''_2, O'''_2$.
Let 11) $M_1 = Scross_p(s_1, Slabels_p(s_2)) \cup M'''_1$, 12) $O_1 = Slabels_p(s_2) \cup O'''_1$, 13) $M_2 = Scross_p(s_2, Slabels_p(s_1)) \cup M'''_2$, 14) $O_2 = Slabels_p(s_1) \cup O'''_2$,
15) $M''''_1 = Scross_p(s_1, Slabels_p(s_2) \cup R) \cup M'''_1$ 16) $O''''_1 = Slabels_p(s_2) \cup R \cup O'''_1$, 17) $M''''_2 = Scross_p(s_2, Slabels_p(s_1) \cup R) \cup M'''_2$ and 18) $O''''_2 = Slabels_p(s_1) \cup R \cup O'''_2$. We use the induction hypothesis applied to 9),11), and 12); 10),13) and 14); 9),15) and 16); and 10),17) and 18) to get 19) $p, E, Slabels_p(s_2) \vdash s_1 : M_1, O_1$, 20) $p, E, Slabels_p(s_1) \vdash s_2 : M_2, O_2$, 21) $p, E, Slabels_p(s_2) \cup R \vdash s_1 : M''''_1, O''''_1$ and
22) $p, E, Slabels_p(s_1) \cup R \vdash s_2 : M''''_2, O''''_2$. Using Lemma (8) on 1) and 19); 2) and 20); 5) and 21); and 6) and 22) we have 23) $M'_1 = M_1$, 24) $O'_1 = O_1$, 25) $M'_2 = M_2$, 26) $O'_2 = O_2$, 27) $M''_1 = M''''_1$, 28) $O''_1 = O''''_1$, 29) $M''_2 = M''''_2$ and 30) $O''_2 = O''''_2$. We substitute 11),13),23) and 25) in 3) to get 31) $M' = Scross_p(s_1, Slabels_p(s_2)) \cup Scross_p(s_2, Slabels_p(s_1)) \cup M'''_1 \cup M'''_2$. Substituting 15),17),27) and 29) in 7) gives us 32) $M'' = Lcross(l, R) \cup Scross_p(s_1, Slabels_p(s_2) \cup R) \cup Scross_p(s_2, Slabels_p(s_1) \cup R) \cup M''''_1 \cup M''''_2$. We may apply Lemma (7.5) and then substitute 31) in 32) to get 33) $M'' = Lcross(l, R) \cup Scross_p(s_1, R) \cup Scross_p(s_2, R) \cup M'$. Using Lemma (7.17) with Rule (19) on 33) gives us 34) $M'' = Scross_p(s, R) \cup M'$. From 4),8),14),18),26) and 30) we may perform substitutions to get 35) $O'' = R \cup O'$. By substituting the premise in 34) and 35) we get $M'' = M$ and $O'' = O$.

If $s \equiv finish^l\ s_1\ s_2$ then by Rule (55) we have 1) $p, E, \emptyset \vdash s_1 : M'_1, O'_1$, 2) $p, E, \emptyset \vdash s_2 : M'_2, O'_2$, 3) $M' = Lcross(l, \emptyset) \cup M'_1 \cup M'_2 = M'_1 \cup M'_2$, 4) $O' = O'_2$, 5) $p, E, R \vdash s_1 : M''_1, O''_1$, 6) $p, E, R \vdash s_2 : M''_2, O''_2$, 7) $M'' = Lcross(l, R) \cup M''_1 \cup M''_2$ and 8) $O'' = O''_2$. Let 9) $M_1 = Scross_p(s_1, R) \cup M'_1$, 10) $O_1 = R \cup O'_1$, 11) $M_2 = Scross_p(s_2, R) \cup M'_2$ and 12) $O_2 = R \cup O'_2$. From the induction hypothesis applied with 1),9) and 10) and 2),11) and 12) we get 13) $p, E, R \vdash s_1 : M_1, O_1$ and 14) $p, E, R \vdash s_2 : M_2, O_2$. Using Lemma (8) on 5) and 13) and on 6) and 14) we get 15) $M_1 = M''_1$, 16) $M_2 = M''_2$ and 17) $O_2 = O''_2$. We substitute 9),11),15) and 17) in 7) to get 18) $M'' = Lcross(l, R) \cup Scross_p(s_1, R) \cup Scross_p(s_2, R) \cup M'_1 \cup M'_2$. Using Lemma (7.17) with Rule (20) on 18) we get 19) $M'' = Scross_p(s, R) \cup M'_1 \cup M'_2$. Substituting 4),12) and 17) in 8) gives

us 20) $O'' = R \cup O'$. From the 3),19) and the premise we have $M'' = M$ and from 20) and the premise we have $O'' = O$.

If $s \equiv f_i()^l\ k$ then by Rule (56) we have 1) $E(f_i) = (M_i, O_i)$, 2) $p, E, O_i \vdash k : M'_k, O'_k$, 3) $M' = Lcross(l, \emptyset) \cup symcross(Slabels_p(p(f_i)), \emptyset) \cup M_i \cup M'_k = M_i \cup M'_k$, 4) $O' = O'_k$, 5) $p, E, R \cup O_i \vdash k : M''_k, O''_k$, 6) $M'' = Lcross(l, R) \cup symcross(Slabels_p(p(f_i)), R) \cup M_i \cup M''_k$ and 7) $O'' = O''_k$. Applying Lemma (10) with 2) we have that there exists M'''_k and O'''_k such that 8) $p, E, \emptyset \vdash k : M'''_k, O'''_k$. Let 9) $M''''_k = Scross_p(k, O_i) \cup M'''_k$, 10) $O''''_k = O_i \cup O'''_k$, 11) $M'''''_k = Scross_p(k, R \cup O_i) \cup M'''_k$ and 12) $O'''''_k = R \cup O_i \cup O'''_k$. We may apply the induction hypothesis with the premise, 8),9) and 10) to get 13) $p, E, O_i \vdash k : M''''_k, O''''_k$. We also use the induction hypothesis with the premise, 8),11) and 12) to get 14) $p, E, R \cup O_i \vdash k : M'''''_k, O'''''_k$. Using Lemma (8) with 2) and 13) and with 5) and 14) gives us 15) $M'_k = M''''_k$, 16) $O'_k = O''''_k$, 17) $M''_k = M'''''_k$ and 18) $O''_k = O'''''_k$. We apply Lemma (7.5) on 11) to get 19) $M'''''_k = Scross_p(k, R) \cup Scross_p(k, O_i) \cup M'''_k$. Substituting 9),15) and 17) in 19) gives us 20) $M''_k = Scross_p(k, R) \cup M'_k$. Substituting 10),16) and 18) in 12) gives us 21) $O''_k = R \cup O'_k$. We substitute 20) in 6) then apply the definition of $Scross_p()$ with $p(f_i) = s_i$ to get 22) $M'' = Lcross(l, R) \cup Scross_p(s_i, R) \cup Scross_p(k, R) \cup M_i \cup M'_k$. Applying Lemma (7.17) with Rule (21) to 22) gives us 23) $M'' = Scross_p(s, R) \cup M_i \cup M'_k$. We may substitute 3) in 24) to get 24) $M'' = Scross_p(s, R) \cup M'$. Substituting 4) and 7) in 22) gives us 25) $O'' = R \cup O'$. Substituting the premise in 24) and 25) gives us $M'' = M$ and $O'' = O$ as desired. □

Likewise, we need a version that applies to an execution tree.

LEMMA 13. *$p, E, R \vdash T : M$ if and only if there exists M' such that $p, E, \emptyset \vdash T : M'$ and $M = Tcross_p(T, R) \cup M'$.*

Proof. ⇒) From Lemma (11) there exists M' such that $p, E, \emptyset \vdash T : M'$. We perform induction on T and in each of the four cases we will show $M = Tcross_p(T, R) \cup M'$.

If $T \equiv \surd$ then from Rule (49) we have 1) $M = \emptyset$ and 2) $M' = \emptyset$. From Lemma (7.9) we have 3) $Tcross_p(\surd, R) = \emptyset$. From 1), 2) and 3) we can see that $M = Tcross_p(T, R) \cup M'$.

If $T \equiv T_1 \rhd T_2$ then by Rule (46) we have 1) $p, E, R \vdash T_1 : M_1$, 2) $p, E, R \vdash T_2 : M_2$, 3) $M = M_1 \cup M_2$, 4) $p, E, \emptyset \vdash T_1 : M'_1$, 5) $p, E, \emptyset \vdash T_2 : M'_2$ and 6) $M' = M'_1 \cup M'_2$. We may use the induction hypothesis on 1) and 2) to get 7) $p, E, \emptyset \vdash T_1 : M''_1$, 8) $M_1 = Tcross_p(T_1, R) \cup M'_1$, 9) $p, E, \emptyset \vdash T_2 : M''_2$ and 10) $M_2 = Tcross_p(T_2, R) \cup M''_2$. From Lemma (9) we have that 11) $M'_1 = M''_1$ and 12) $M'_2 = M''_2$. Let us substitute 8),10),11) and 12) in 3) to get 13) $M = Tcross_p(T_1, R) \cup Tcross_p(T_2, R) \cup M'_1 \cup M'_2$. From Rule (23) we may apply Lemma (7.19) on 13) to get 14) $M = Tcross_p(T, R) \cup M'_1 \cup M'_2$. Finally substituting 6) in 14) we get $M = Tcross_p(T, R) \cup M'$.

If $T \equiv T_1 \parallel T_2$ then by Rule (47) we have
1) $p, E, Tlabels_p(T_2) \cup R \vdash T_1 : M_1$, 2) $p, E, Tlabels_p(T_1) \cup R \vdash T_2 : M_2$, 3) $M = M_1 \cup M_2$, 4) $p, E, Tlabels_p(T_2) \vdash T_1 : M'_1$, 5) $p, E, Tlabels_p(T_1) \vdash T_2 : M'_2$ and 6) $M' = M'_1 \cup M'_2$. From the induction hypothesis applied to 1),2),4) and 5) we get 7) $p, E, \emptyset \vdash T_1 : M''_1$, 8) $M_1 = Tcross_p(T_1, Tlabels_p(T_2) \cup R) \cup M''_1$, 9) $p, E, \emptyset \vdash T_2 : M''_2$,
10) $M_2 = Tcross_p(T_2, Tlabels_p(T_1)) \cup R) \cup M''_2$, 11) $p, E, \emptyset \vdash T_1 : M'''_1$ 12) $M'_1 = Tcross_p(T_1, Tlabels_p(T_2)) \cup M'''_1$, 13) $p, E, \emptyset \vdash T_2 : M'''_2$ and 14) $M'_2 = Tcross_p(T_2, Tlabels_p(T_1)) \cup M'''_2$, From Lemma (9) applied to 7) and 11) and to 9) and 13) we get 15) $M''_1 = M'''_1$ and 16) $M''_2 = M'''_2$. We use Lemma (7.7) on 8) and 10) to get 17) $M_1 = Tcross_p(T_1, Tlabels_p(T_2)) \cup Tcross_p(T_1, R) \cup M''_1$ and
18) $M_2 = Tcross_p(T_2, Tlabels_p(T_1)) \cup Tcross_p(T_2, R) \cup M''_2$. We substitute 12) and 15) in 17) and substitute 14) and 16) in 18) to get 19) $M_1 = Tcross_p(T_1, R) \cup M'_1$ and 20) $M_2 = Tcross_p(T_2, R) \cup M'_2$. Substituting 6),21) and 22) in 3) yields 21) $M = Tcross_p(T_1, R) \cup Tcross_p(T_2, R) \cup M'$. Finally we use Lemma (7.19) with Rule (24) on 21) to get our conclusion $M = Tcross_p(T, R) \cup M'$.

If $T \equiv \langle s \rangle$ then from Rule (48) we have 1) $p, E, R \vdash s : M_s, O_s$, 2) $M = M_s$, 3) $p, E, \emptyset \vdash s : M'_s, O'_s$ and 4) $M' = M'_s$. Using Lemma (12) with the premise on 1) we get 5) $p, E, \emptyset \vdash s : M''_s, O''_s$ and 6) $M_s = Scross_p(s, R) \cup M''_s$. From Lemma (9) we have that 7) $M'_s = M''_s$. Using Lemma (7.18) on 6) we get 9) $M_s = Tcross_p(T, R) \cup M''_s$. We substitute 4),7) and 8) in 2) and we get $M = Tcross_p(T, R) \cup M'$.

⇐) From Lemma (11) there exists M'' such that $p, E, R \vdash T : M''$. We also have $\vdash p : E$ from the premise. Using induction on T we will examine the four cases and show that $M'' = M$ which will give us our conclusion that $p, E, R \vdash T : M$.

If $T \equiv \surd$ then from Rule (49) we have 1) $M'' = \emptyset$ and 2) $M' = \emptyset$. From Lemma (7.9) we have 3) $Tcross_p(\surd, R) = \emptyset$. Let us substitute 2) and 3) in the premise to get 4) $M = \emptyset$. From 1) and 4) we see $M'' = M$.

If $T \equiv T_1 \rhd T_2$ then from Rule (46) we have 1) $p, E, R \vdash T_1 : M''_1$, 2) $p, E, R \vdash T_2 : M''_2$, 3) $M'' = M''_1 \cup M''_2$, 4) $p, E, \emptyset \vdash T_1 : M'_1$, 5) $p, E, \emptyset \vdash T_2 : M'_2$ and 6) $M' = M'_1 \cup M'_2$. Let 7) $M'''_1 = Tcross_p(T_1, R) \cup M'_1$ and 8) $M'''_2 = Tcross_p(T_2, R) \cup M'_2$. Using the induction hypothesis with 4) and 7) and with 5) and 8) we obtain 9) $p, E, R \vdash T_1 : M'''_1$ and 10) $p, E, R \vdash T_2 : M'''_2$. Using Lemma (9) on 1) and 9) and on 2) and 10) we get 11) $M''_1 = M'''_1$ and 12) $M''_2 = M'''_2$. Substituting 6),7),8),11) and 12) in 3) gives us 13) $M'' = Tcross_p(T_1, R) \cup Tcross_p(T_2, R) \cup M'$. We may use Lemma (7.19) with Rule (23) on 13) to get 14) $M'' = Tcross_p(T, R) \cup M'$. Comparing 14) to the premise gives us $M'' = M$.

If $T \equiv T_1 \parallel T_2$ then from Rule (47) we have
1) $p, E, Tlabels_p(T_2) \cup R \vdash T_1 : M''_1$, 2) $p, E, Tlabels_p(T_1) \cup R \vdash T_2 : M''_2$, 3) $M'' = M''_1 \cup M''_2$, 4) $p, E, Tlabels_p(T_2) \vdash T_1 : M'_1$, 5) $p, E, Tlabels_p(T_1) \vdash T_2 : M'_2$ and 6) $M' = M'_1 \cup M'_2$, From Lemma (11) there exist M_1 and M_2 such that 7) $p, E, \emptyset \vdash T_1 : M_1$ and 8) $p, E, \emptyset \vdash T_2 : M_2$. Let 9) $M'''_1 = Tcross_p(T_1, Tlabels_p(T_2) \cup R) \cup M_1$,
10) $M'''_2 = Tcross_p(T_2, Tlabels_p(T_1) \cup R) \cup M_2$, 11) $M''''_1 = Tcross_p(T_1, Tlabels_p(T_2)) \cup M_1$ and
12) $M''''_2 = Tcross_p(T_2, Tlabels_p(T_1)) \cup M_2$.
Applying Lemma (7.7) on 9) and 10) which gives us 13) $M'''_1 = Tcross_p(T_1, R) \cup Tcross_p(T_1, Tlabels_p(T_2)) \cup M_1$ and 14) $M'''_2 = Tcross_p(T_2, R) \cup Tcross_p(T_2, Tlabels_p(T_1)) \cup M_2$. Substituting 11) and 12) in 13) and 14), respectively, yields 15) $M'''_1 = Tcross_p(T_1, R) \cup M''''_1$ and 16) $M'''_2 = Tcross_p(T_2, R) \cup M''''_2$. Applying the induction hypothesis to 7) and 9); 8) and 10); 7) and 11); and 8) and 12) to get 17) $p, E, Tlabels_p(T_2) \cup R \vdash T_1 : M'''_1$, 18) $p, E, Tlabels_p(T_1) \cup R \vdash T_2 : M''''_2$, 19) $p, E, Tlabels_p(T_2) \vdash T_1 : M''''_1$ and 20) $p, E, Tlabels_p(T_1) \vdash T_2 : M''''_2$. From Lemma (9) applied to 1) and 17); 2) and 18); 4) and 19); and 5) and 20) which gives us 21) $M'''_1 = M''''_1$, 22) $M''''_2 = M''''_2$, 23) $M''''_1 = M''''_1$ and 24) $M''''_2 = M''''_2$. Substituting 6),15),16),21),22),23) and 24) in 3) yields 26) $M'' = Tcross_p(T_1, R) \cup Tcross_p(T_2, R) \cup M'$. We use Lemma (7.19) with Rule (24) to get 27) $M'' = Tcross_p(T, R) \cup M'$. With 27) and the premise we see that $M'' = M$.

If $T \equiv \langle s \rangle$ then from Rule (48) we have 1) $p, E, R \vdash s : M''_s, O''_s$, 2) $M'' = M''_s$, 3) $p, E, \emptyset \vdash s : M'_s, O'_s$ and 4) $M' = M'_s$. Using Lemma (12) with the premise on 1) we get 5) $p, E, \emptyset \vdash s : M'''_s, O'''_s$ 6) $M''_s = Scross_p(s, R) \cup M'''_s$. From Lemma (8) applied to 3) and 5) we get 7) $M'_s = M'''_s$. We now will substitute 4),6) and 7) in 2) to get 8) $M'' = Scross_p(s, R) \cup M'$. Applying

Lemma (7.18) and on 8) we get 9) $M'' = Tcross_p(T, R) \cup M'$. Upon comparing the premise with 9) we see that $M'' = M$. $\square$

8.4 Preservation

In Rules (11) and (14) we use the . operator to combine statements so we need a way to type check such combined statements. The following lemma shows that a natural type rule for $s_a \ . \ s_b$ is admissible.

LEMMA 14. *If $p, E, R \vdash s_a : M_a, O_a$ and $p, E, O_a \vdash s_b : M_b, O_b$ and $p, E, R \vdash s_a \ . \ s_b : M, O$ then $M = M_a \cup M_b$ and $O = O_b$.*

Proof. Let $s = s_a \ . \ s_b$. We will perform induction on s_a. This gives us seven cases.

If $s_a \equiv skip^l$ then by the definition of . we have 1) $s = skip^l \ s_b$. From Rule (51) we have 2) $p, E, R \vdash s_b : M_b', O_b'$, 3) $M = Lcross(l, R) \cup M_b'$ and 4) $O = O_b'$. From Rule (50) we have 5) $M_a = Lcross(l, R)$ and 6) $O_a = R$. Substituting 6) in the premise gives us 7) $p, E, R \vdash s_b : M_b, O_b$. From Lemma (8) applied to 2) and 7) we get 8) $M_b = M_b'$ and 9) $O_b = O_b'$. Using substitution of 5) and 8) in 3) and 9) in 4) we have $M = M_a \cup M_b$ and $O = O_b$.

If $s_a \equiv skip^l \ s_1$ then by Rule (50) and the definition of . we have 1) $s = skip^l \ (s_1 \ . \ s_b)$. From Rule (51) we have 2) $p, E, R \vdash (s_1 \ . \ s_b) : M_k, O_k$, 3) $M = Lcross(l, R) \cup M_k$, 4) $O = O_k$, 5) $p, E, R \vdash s_1 : M_1, O_1$, 6) $M_a = Lcross(l, R) \cup M_1$ and 7) $O_a = O_1$. After substituting 7) in 5), we may use the induction hypothesis to get 8) $M_k = M_1 \cup M_b$ and 9) $O_k = O_b$. From 3),4),6) and 9) we arrive at our conclusion that $M = M_a \cup M_b$ and $O = O_b$.

If $s_a \equiv a[d] =^l e; \ s_1$ then we proceed using similar reasoning as with the previous case.

If $s_a \equiv while^l \ (a[d] \neq 0) \ s_1 \ s_2$ then from the definition of . we have 1) $s = while^l \ (a[d] \neq 0) \ s_1 \ (s_2 \ . \ s_b)$. From Rule (53) we have 2) $p, E, R \vdash s_1 : M_1, O_1$, 3) $p, E, O_1 \vdash (s_2 \ . \ s_b) : M_k, O_k$, 4) $M = Lcross(l, O_1) \cup Scross_p(s_1, O_1) \cup M_1 \cup M_k$, 5) $O = O_k$, 6) $p, E, R \vdash s_1 : M_1', O_1'$, 7) $p, E, O_1' \vdash s_2 : M_2', O_2'$, 8) $M_a = Lcross(l, O_1') \cup Scross_p(s_1, O_1') \cup M_1' \cup M_2'$ and 9) $O_a = O_2'$. From Lemma (8) applied to 2) and 6) we have that 10) $M_1 = M_1'$ and 11) $O_1 = O_1'$. Substituting 9) and 11) in 7) allows us to use the induction hypothesis to get 12) $M_k = M_2' \cup M_b$ and 13) $O_k = O_b$. From 4),5),8),12) and 13) we see that $M = M_a \cup M_b$ and $O = O_b$.

If $s_a \equiv async^l \ s_1 \ s_2$ then from the definition of . we have 1) $s = async^l \ s_1 \ (s_2 \ . \ s_b)$. From Rule (54) we have 2) $p, E, Slabels_p(s_2 \ . \ s_b) \cup R \vdash s_1 : M_1, O_1$, 3) $p, E, Slabels_p(s_1) \cup R \vdash (s_2 \ . \ s_b) : M_k, O_k$, 4) $M = Lcross(l, R) \cup M_1 \cup M_k$, 5) $O = O_k$, 6) $p, E, Slabels_p(s_2) \cup R \vdash s_1 : M_1', O_1'$, 7) $p, E, Slabels_p(s_1) \cup R \vdash s_2 : M_2', O_2'$, 8) $M_a = Lcross(l, R) \cup M_1' \cup M_2'$ and 9) $O_a = O_2'$. By substituting 9) in 7) we are able to apply the induction hypothesis and get 10) $M_k = M_2' \cup M_b$ and 11) $O_k = O_b$. Applying Lemma (12) to 2),6),7) and the $p, E, O_a \vdash s_b : M_b, O_b$ from the premise gives us 12) $p, E, \emptyset \vdash s_1 : M_w, O_w$ 13) $M_1 = Scross_p(s_1, Slabels_p(s_2 \ . \ s_b) \cup R) \cup M_w$, 14) $p, E, \emptyset \vdash s_1 : M_x, O_x$, 15) $M_1' = Scross_p(s_1, Slabels_p(s_2) \cup R) \cup M_x$, 16) $p, E, \emptyset \vdash s_2 : M_y, O_y$, 17) $O_2' = Slabels_p(s_1) \cup R \cup O_y$, 18) $p, E, \emptyset \vdash s_b : M_z, O_z$ and 19) $M_b = Scross_p(s_b, O_a) \cup M_z$. From Lemma (8) applied to 12) and 14) we get 20) $M_w = M_x$. We may substitute 9) and 17) in 19) to get 21) $M_b = Scross_p(s_b, Slabels_p(s_1) \cup R \cup O_y) \cup M_z$. Using Lemma (7.11) on 13) we get
22) $M_1 = Scross_p(s_1, Slabels_p(s_2) \cup Slabels_p(s_b) \cup R) \cup M_w$. Applying Lemma (7.5) to 21) and 22) yields 23) $M_b = Scross_p(s_b, Slabels_p(s_1)) \cup Scross_p(s_b, R \cup O_y) \cup M_z$ and 24) $M_1 = Scross_p(s_1, Slabels_p(s_b)) \cup Scross_p(s_1, Slabels_p(s_2) \cup R) \cup M_w$. Using Lemma (7.6) we have
25) $Scross_p(s_1, Slabels_p(s_b)) = Scross_p(s_b, Slabels_p(s_1))$. From 23) and 25) we see that 26) $Scross_p(s_b, Slabels_p(s_1)) \subseteq M_b$. We now substitute 15) and 20) in 24) to get 27) $M_1 = Scross_p(s_1, Slabels_p(s_b)) \cup M_1'$. Substituting 10) and 27) in 4) gives us 28) $M = Lcross(l, R) \cup Scross_p(s_1, Slabels_p(s_b)) \cup M_1' \cup M_2' \cup M_b$. From 27) we may simplify 28) to 29) $M = Lcross(l, R) \cup M_1' \cup M_2' \cup M_b$. Finally substituting 8) in 29) gives us $M = M_a \cup M_b$ and then substituting 5) in 11) gives us $O = O_b$.

If $s_a \equiv finish^l \ s_1 \ s_2$ then from the definition of . we obtain 1) $s = finish^l \ s_1 \ (s_2 \ . \ s_b)$. From Rule (55) we have 2) $p, E, R \vdash s_1 : M_1, O_1$, 3) $p, E, R \vdash (s_2 \ . \ s_b) : M_k, O_k$, 4) $M = Lcross(l, R) \cup M_1 \cup M_k$, 5) $O = O_k$, 6) $p, E, R \vdash s_1 : M_1', O_1'$, 7) $p, E, R \vdash s_2 : M_2', O_2'$, 8) $M_a = Lcross(l, R) \cup M_1' \cup M_2'$ and 9) $O_a = O_2'$. From Lemma (8) applied to 2) and 6) we have 10) $M_1 = M_1'$. By substituting 9) in 7) we may apply the induction hypothesis to get 11) $M_k = M_2' \cup M_b$ and 12) $O_k = O_b$. From 4),5),8),10),11) and 12) we have $M = M_a \cup M_b$ and $O = O_b$.

If $s_a \equiv f_i()^l \ k$ then from the definition of . we get 1) $s = f_i()^l \ (k \ . \ s_b)$. From Rule (56) we have 2) $E(f_i) = (M_i, O_i)$, 3) $p, E, R \cup O_i \vdash (k \ . \ s_b) : M_k', O_k'$, 4) $M = Lcross(l, R) \cup symcross(Slabels_p(p(f_i)), R) \cup M_i \cup M_k'$, 5) $O = O_k'$, 6) $p, E, R \cup O_i \vdash k : M_k, O_k$, 7) $M_a = Lcross(l, R) \cup symcross(Slabel_p(p(f_i)), R) \cup M_i \cup M_k$ and 8) $O_a = O_k$. Applying the induction hypothesis with the premise, 3) and 6) gives us 9) $M_k' = M_k \cup M_b$ and 10) $O_k' = O_b$. From 4),5),7),8),9) and 10) we have $M = M_a \cup M_b$ and $O = O_b$. $\square$

When we step by Rule (3) and (4) in the proof of Preservation, we will need this helper lemma.

LEMMA 15. *If $p, E, R \vdash T : M$ and $p, E, R' \vdash T : M'$ and $R' \subseteq R$ then $M' \subseteq M$.*

Proof. Using Lemma (13) on the premise we have 1) $p, E, \emptyset \vdash T : M_0$, 2) $M = Tcross_p(T, R) \cup M_0$, 3) $p, E, \emptyset \vdash T : M_0'$ and 4) $M' = Tcross_p(T, R') \cup M_0'$. Applying Lemma (9) to 1) and 3) gives us 5) $M_0 = M_0'$. We use Lemma (7.10) with the premise to get 6) $Tcross_p(T, R') \subseteq Tcross_p(T, R)$. From 2),4),5) and 6) it is easy to see that $M' \subseteq M$. $\square$

We are now ready to prove preservation.

LEMMA 16. *If $\vdash p : E$ and $p, E, \emptyset \vdash T : M$ and $(p, A, T) \rightarrow (p, A', T')$, then there exists M' such that $p, E, \emptyset \vdash T' : M'$ and $M' \subseteq M$.*

Proof. From Lemma (13) there exists M' such that 0) $p, E, \emptyset \vdash T' : M'$. We will now show $M' \subseteq M$. We perform induction on T and examine the four cases.

If $T \equiv \surd$ then T does not take a step.

If $T \equiv T_1 \rhd T_2$ then there are two rules by which we may take a step.

Suppose we step by Rule (1) we have that 1) $T' = T_2$. We may substitute 1) in 0) to get 2) $p, E, \emptyset \vdash T_2 : M'$ From Rule (46) we have 3) $p, E, \emptyset \vdash T_1 : M_1$, 4) $p, E, \emptyset \vdash T_2 : M_2$ and 5) $M = M_1 \cup M_2$. From Lemma (9) applied to 2) and 4) we have that 6) $M' = M_2$. We see the that $M' \subseteq M$ from 5) and 6).

Suppose we step by Rule (2) we have 1) $T' = T_1' \rhd T_2$ and 2) $(p, A, T_1) \rightarrow (p, A', T_1')$. Substituting 1) in 0) gives us 3) $p, E, \emptyset \vdash T_1' \rhd T_2 : M'$. From Rule (46) we have 4) $p, E, \emptyset \vdash T_1 : M_1$, 5) $p, E, \emptyset \vdash T_2 : M_2$, 6) $M = M_1 \cup M_2$, 7) $p, E, \emptyset \vdash T_1' : M_1'$, 8) $p, E, \emptyset \vdash T_2 : M_2'$ and 9) $M' = M_1' \cup M_2'$. From Lemma (9) applied to 5) and 8) we have 10) $M_2 = M_2'$. We may apply the induction hypothesis with 4) and 2) and get that there exists M_1'' such that 11) $p, E, \emptyset \vdash T_1' : M_1''$ and 12) $M_1'' \subseteq M_1$. Using Lemma (9) on 7) and 11) we get 13) $M_1' = M_1''$. From 6),9),10),12) and 13) we see that $M' \subseteq M$.

If $T \equiv T_1 \parallel T_2$ then there are four rules by which we may take a step.

Suppose we step by Rule (3) we then have 1) $T' = T_2$. We may substitute 1) in 0) to get 2) $p, E, \emptyset \vdash T_2 : M'$. From Rule (47) we have 3) $p, E, Tlabels_p(T_2) \vdash T_1 : M_1$, 4) $p, E, Tlabels_p(T_1) \vdash T_2 : M_2$ and 5) $M = M_1 \cup M_2$. We can immediately see that 6) $\emptyset \subseteq Tlabels(T_1)$. We also apply Lemma (15) on 2),4) and 6) to get 7) $M' \subseteq M_2$. We may see from 5) and 7) that $M' \subseteq M$.

Suppose we step by Rule (4) then we proceed using similar reasoning as the previous case.

Suppose we step by Rule (5) then we have 1) $T' = T_1' \parallel T_2$ and 2) $(p, A, T_1) \rightarrow (p, A', T_1')$. Substituting 1) in 0) yields 3) $p, E, \emptyset \vdash T_1' \parallel T_2 : M'$. From Rule (47) we have 4) $p, E, Tlabels_p(T_2) \vdash T_1 : M_1$, 5) $p, E, Tlabels_p(T_1) \vdash T_2 : M_2$, 6) $M = M_1 \cup M_2$, 7) $p, E, Tlabels_p(T_2) \vdash T_1' : M_1'$, 8) $p, E, Tlabels_p(T_1') \vdash T_2 : M_2'$ and 9) $M' = M_1' \cup M_2'$. Using Lemma (13) on 4),5),6) and 7) gives us 10) $p, E, \emptyset \vdash T_1 : M_1''$, 11) $M_1 = Tcross_p(T_1, Tlabels_p(T_2)) \cup M_1''$, 12) $p, E, \emptyset \vdash T_2 : M_2''$, 13) $M_2 = Tcross_p(T_2, Tlabels_p(T_1)) \cup M_2''$, 14) $p, E, \emptyset \vdash T_1' : M_1'''$, 15) $M_1' = Tcross_p(T_1', Tlabels_p(T_2)) \cup M_1'''$, 16) $p, E, \emptyset \vdash T_2 : M_2'''$ and 17) $M_2' = Tcross_p(T_2, Tlabels_p(T_1')) \cup M_2'''$. From using the induction hypothesis applied to 2),10) and 14) and using Lemma (9) we get 18) $M_1''' \subseteq M_1''$. We use Lemma (9) on 12) and 16) to get 19) $M_2'' = M_2'''$. Using Lemma (7.8) and substituting 11),13),15),17) and 19) in 6) and 9) results in 20) $M = Tcross_p(T_2, Tlabels_p(T_1)) \cup M_1'' \cup M_2''$ and 21) $M' = Tcross_p(T_2, Tlabels_p(T_1')) \cup M_1''' \cup M_2''$. We use Lemma (7.15) with 2) to get 22) $Tlabels_p(T_1') \subseteq Tlabels_p(T_1)$. We now use Lemma (7.10) with 22) to get 23) $Tcross_p(T_2, Tlabels_p(T_1')) \subseteq Tcross_p(T_2, Tlabels_p(T_1))$. From 18),20),21) and 23) we may get $M' \subseteq M$.

Suppose we step by Rule (6) the we may proceed using similar logic as the previous case.

If $T \equiv \langle s \rangle$ then we now perform induction on s which gives us an additional seven cases.

If $s \equiv skip^l$ then we take a step by Rule (7) and have 1) $T' = \surd$. We may substitute 1) in 0) to get 2) $p, E, \emptyset \vdash \surd : M'$. From Rule (49) we have 3) $M' = \emptyset$. From 3) we see that $M' \subseteq M$.

If $s \equiv skip^l\ s_1$ then we take a step by Rule (8) and have 1) $T' = \langle s_1 \rangle$. We may substitute 1) in 0) to get 2) $p, E, \emptyset \vdash s_1 : M'$. Using Rule (48) we have 3) $p, E, \emptyset \vdash s : M_s, O_s$, 4) $M = M_s$, 5) $p, E, \emptyset \vdash s_1 : M_s', O_s'$ and 6) $M' = M_s'$. From Rule (51) we have 7) $p, E, \emptyset \vdash s_1 : M_{s_1}, O_{s_1}$ and 8) $M_s = Lcross(l, \emptyset) \cup M_{s_1}$. We may use Lemma (8) on 5) and 7) to get 9) $M_s' = M_{s_1}$. From 4),6),8) and 9) we see that $M' \subseteq M$.

If $s \equiv a[d] =^l e;\ s_1$ then we step by Rule (9) then we may proceed using similar logic as the previous case.

If $s \equiv while^l\ (a[d] \neq 0)\ s_1\ s_2$ then there are two rules by which we may take a step.

Suppose we step by Rule (10) then we have 1) $T' = \langle s_2 \rangle$. We substitute 1) in 0) to get 2) $p, E, \emptyset \vdash \langle s_2 \rangle : M'$. Let 3) $R = \emptyset$. From Rule (48) we have 4) $p, E, R \vdash \langle s \rangle : M_s$, 5) $M = M_s$, 6) $p, E, R \vdash \langle s_2 \rangle : M_s'$ and 7) $M' = M_s'$. From Rule (53) we have 8) $p, E, R \vdash s_1 : M_{s_1}, O_{s_1}$, 9) $p, E, O_{s_1} \vdash s_2 : M_{s_2}, O_{s_2}$ and 10) $M_s = Lcross(l, O_{s_1}) \cup Scross_p(s_1, O_{s_1}) \cup M_{s_1} \cup M_{s_2}$. Applying Lemma (12) to 6),8) and 9) we get 11) $p, E, \emptyset \vdash s_2 : M_s'', O_s''$, 12) $M_s' = Scross_p(s_2, R) \cup M_s''$, 13) $p, E, \emptyset \vdash s_1 : M_{s_1}'', O_{s_1}''$, 14) $O_{s_1} = R \cup O_{s_1}''$, 15) $p, E, \emptyset \vdash s_2 : M_{s_2}'', O_{s_2}''$ and 16) $M_{s_2} = Scross_p(s_2, O_{s_1}) \cup M_{s_2}''$. From Lemma (8) applied to 11) and 15) we get 17) $M_s'' = M_{s_2}''$. Substituting 14) and 17) in 16) gives us 18) $M_{s_2} = Scross_p(s_2, R \cup O_{s_1}'') \cup M_s''$. Using Lemma (7.5) on 18) results in 19) $M_{s_2} = Scross_p(s_2, O_{s_1}'') \cup Scross_p(s_2, R) \cup M_s''$. From 12),17) and 19) we have 20) $M_s' \subseteq M_{s_2}$. Finally from 5),7),10) and 20) we have $M' \subseteq M$.

Suppose we step by Rule (11) then we have 1) $T' = \langle s_1\ .\ s \rangle$. Substituting 1) in 0) gives us 2) $p, E, R \vdash s_1\ .\ s : M'$. Let 3) $R = \emptyset$. From Rule (48) we have 4) $p, E, R \vdash s : M_s, O_s$, 5) $M = M_s$, 6) $p, E, R \vdash s_1\ .\ s : M_s', O_s'$ and 7) $M' = M_s'$. From Lemma (10) there exists M_{s_1}',M_{ss}',O_{s_1}' and O_{ss}' such that 8) $p, E, R \vdash s_1 : M_{s_1}', O_{s_1}'$ and 9) $p, E, O_{s_1}' \vdash s : M_{ss}', O_{ss}'$. We may use Lemma (14) with 6),8) and 9) to get 10) $M_s' = M_{s_1}' \cup M_{ss}'$. From Rule (53) we have 11) $p, E, R \vdash s_1 : M_1, O_1$, 12) $p, E, O_1 \vdash s_2 : M_2, O_2$, 13) $M_s = Lcross(l, O_1) \cup Scross_p(s_1, O_1) \cup M_1 \cup M_2$, 14) $p, E, O_{s_1}' \vdash s_1 : M_1', O_1'$, 15) $p, E, O_1' \vdash s_2 : M_2', O_2'$ and 16) $M_{ss}' = Lcross(l, O_1') \cup Scross_p(s_1, O_1') \cup M_1' \cup M_2'$. Using Lemma (8) with 8) and 11) we have 17) $M_{s_1}' = M_1$ and 18) $O_{s_1}' = O_1$. Applying Lemma (12) to 11) and 14) gives us 19) $p, E, \emptyset \vdash s_1 : M_1'', O_1''$, 20) $M_1 = Scross_p(s_1, R) \cup M_1''$, 21) $O_1 = R \cup O_1''$, 22) $p, E, \emptyset \vdash s_1 : M_1''', O_1'''$, 23) $M_1' = Scross_p(s_1, O_{s_1}') \cup M_1'''$ and 24) $O_1' = O_{s_1}' \cup O_1'''$. We apply Lemma (8) to 19) and 22) to get 25) $M_1'' = M_1'''$ and 26) $O_1'' = O_1'''$. Let us substitute 18) and 26) in 24) to get 27) $O_1' = O_1 \cup O_1''$. We substitute 21) in 27) to get 28) $O_1' = R \cup O_1'' \cup O_1'' = R \cup O_1''$. From 21) and 28) we get 29) $O_1 = O_1'$. Substituting 29) in 15) we get 30) $p, E, O_1 \vdash s_2 : M_2', O_2'$. Using Lemma (8) on 12) and 30) yields 31) $M_2 = M_2'$. We now substitute 13),20) and 21) in 5) to get 32) $M = Lcross(l, R \cup O_1'') \cup Scross_p(s_1, R \cup O_1'') \cup Scross_p(s_1, R) \cup M_1'' \cup M_2$. Using Lemma (7.5) we may simplify 32) to 33) $M = Lcross(l, R \cup O_1'') \cup Scross_p(s_1, R \cup O_1'') \cup M_1'' \cup M_2$. Substituting 10),16),17),20),23),25) and 31) in 7) results in 34) $M' = Scross_p(s_1, R) \cup M_1'' \cup Lcross(l, R \cup O_1'') \cup Scross_p(s_1, R \cup O_1'') \cup Scross_p(s_1, R \cup O_1'') \cup M_1'' \cup M_2$. From 34) we use Lemma (7.5) and simplify to 35) $M' = Lcross(l, R \cup O_1'') \cup Scross_p(s_1, R \cup O_1'') \cup M_1'' \cup M_2$. From 33) and 35) we see $M' \subseteq M$.

If $s \equiv async^l\ s_1\ s_2$ then we take a step by Rule (12) and have 1) $T' = \langle s_1 \rangle \parallel \langle s_2 \rangle$. We substitute 1) in 0) to get 2) $p, E, \emptyset \vdash \langle s_1 \rangle \parallel \langle s_2 \rangle : M'$. Let 3) $T_1 = \langle s_1 \rangle$ and 4) $T_2 = \langle s_2 \rangle$. From Rule (47) we have 5) $p, E, Tlabels_p(T_2) \vdash \langle s_1 \rangle : M_1'$, 6) $p, E, Tlabels_p(T_1) \vdash \langle s_2 \rangle : M_2'$ and 7) $M' = M_1' \cup M_2'$. From Rule (48) we have 8) $p, E, \emptyset \vdash s : M_s, O_s$, 9) $M = M_s$, 10) $p, E, Tlabels_p(T_2) \vdash s_1 : M_{s_1}', O_{s_1}'$, 11) $M_1' = M_{s_1}'$, 12) $p, E, Tlabels_p(T_1) \vdash s_2 : M_{s_2}', O_{s_2}'$ and 13) $M_2' = M_{s_2}'$. From Rule (54) we have 14) $p, E, Slabels_p(s_2) \vdash s_1 : M_1, O_1$, 15) $p, E, Slabels_p(s_1) \vdash s_2 : M_2, O_2$ and 16) $M_s = M_1 \cup M_2$. From the definition of $Tlabels()$ we may simplify 10) and 12) to 17) $p, E, Slabels_p(s_2) \vdash s_1 : M_{s_1}', O_{s_1}'$ and 18) $p, E, Slabels_p(s_1) \vdash s_2 : M_{s_2}', O_{s_2}'$. We use Lemma (8) on 14) and 17) and on 15) and 18) to get 19) $M_{s_1}' = M_1$ and 20) $M_{s_2}' = M_2$. From 7),9),16),19) and 20) we have $M' \subseteq M$.

If $s \equiv finish^l\ s_1\ s_2$ then we step by Rule (13) which gives us 1) $T' = \langle s_1 \rangle \rhd \langle s_2 \rangle$. Substituting 1) in 0) results in 2) $p, E, \emptyset \vdash \langle s_1 \rangle \rhd \langle s_2 \rangle : M'$. From Rule (46) we have 3) $p, E, \emptyset \vdash \langle s_1 \rangle : M_1'$, 4) $p, E, \emptyset \vdash \langle s_2 \rangle : M_2'$ and 5) $M' = M_1' \cup M_2'$. From Rule (48) we have 6) $p, E, \emptyset \vdash s : M_s, O_s$, 7) $M = M_s$, 8) $p, E, \emptyset \vdash s_1 : M_{s_1}', O_{s_1}'$, 9) $M_1' = M_{s_1}'$, 10) $p, E, \emptyset \vdash s_2 : M_{s_2}', O_{s_2}'$ and 11) $M_2' = M_{s_2}'$. From Rule (55) we get 12) $p, E, \emptyset \vdash s_1 : M_{s_1}, O_{s_1}$, 13) $p, E, \emptyset \vdash s_2 : M_{s_2}, O_{s_2}$ and 14) $M_s = Lcross(l, \emptyset) \cup M_{s_1} \cup M_{s_2}$. Using Lemma (8) on 8) and 12) and on 10) and 13) gives us 15) $M_{s_1} = M_{s_1}'$ and 16) $M_{s_2} = M_{s_2}'$. Substituting 14),15) and 16) in 7) gives us 17) $M = Lcross(l, \emptyset) \cup M_{s_1}' \cup M_{s_2}'$. We substitute 9) and 11) in 5) to get 18) $M' = M_{s_1}' \cup M_{s_2}'$. From 17) and 18) we see $M' \subseteq M$.

If $s \equiv f_i()^l\ k$ then we step by Rule (14) which gives us 1) $p(f_i) = s_i$ and 2) $T' = \langle s_i\ .\ k \rangle$. From $\vdash p : E$ and Rule (45) we also have 3) $E(f_i) = (M_i, O_i)$ and 4) $p, E, \emptyset \vdash s_i : M_i, O_i$. Substituting 2) in 0) gives us 5) $p, E, \emptyset \vdash s_i\ .\ k : M'$. From Rule (48) we have 6) $p, E, \emptyset \vdash s : M_s, O_s$, 7) $M = M_s$, 8)

$p, E, \emptyset \vdash s_i \,.\, k : M'_s, O'_s$ and 9) $M' = M'_s$. Using Rule (56) on 6) gives us 10) $p, E, O_i \vdash k : M_k, O_k$, 11) $M_s = Lcross(l, \emptyset) \cup symcross(Slabels_p(s_i), \emptyset) \cup M_i \cup M_k = M_i \cup M_k$. Applying Lemma (14) with the premise, 4), 8) and 10) gives 12) $M'_s = M_i \cup M_k$. From 7),9),11) and 12) we get $M = M'$ and thus $M' \subseteq M$. □

8.5 Approximation

We now prove that our type system produces a label pair set M, such that if two statements can execute in parallel, then the pairing of their labels will appear in M.

LEMMA 17. *If* $p, E, \emptyset \vdash T : M$ *then* $parallel(T) \subseteq M$.

Proof. Let us perform induction on T. There are four cases.

If $T \equiv \surd$ then from Rule (49) we have 1) $M = \emptyset$. From the definition of $parallel()$, 2) $parallel(T) = \emptyset$. From 1) and 2) we see $parallel(T) \subseteq M$.

If $T \equiv T_1 \rhd T_2$ then from Rule (46) we have 1) $p, E, \emptyset \vdash T_1 : M_1$, 2) $p, E, \emptyset \vdash T_2 : M_2$ and 3) $M = M_1 \cup M_2$. From the definition of $parallel()$ we have 4) $parallel(T) = parallel(T_1)$. Using the induction hypothesis on 1) yields 5) $parallel(T_1) \subseteq M_1$ and From 3),4) and 5) we have $parallel(T) \subseteq M$.

If $T \equiv T_1 \parallel T_2$ then from Rule (47) we have
1) $p, E, Tlabels_p(T_2) \vdash T_1 : M_1$, 2) $p, E, Tlabels_p(T_1) \vdash T_2 : M_2$ and 3) $M = M_1 \cup M_2$. We apply Lemma (13) to 1) and 2) to get 4) $p, E, \emptyset \vdash T_1 : M'_1$, 5) $M_1 = Tcross_p(T_1, Tlabels_p(T_2)) \cup M'_1$, 6) $p, E, \emptyset \vdash T_2 : M'_2$ and 7) $M_2 = Tcross_p(T_2, Tlabels_p(T_1)) \cup M'_2$. Using Lemma (7.8) and substituting 5) and 7) in 3) gives us 8) $M = Tcross_p(T_1, Tlabels_p(T_2)) \cup M'_1 \cup M'_2$. Using the induction hypothesis on 4) and 6) yields 9) $parallel(T_1) \subseteq M'_1$ and 10) $parallel(T_2) \subseteq M'_2$. Unfolding the definition of $parallel()$ gives us 11) $parallel(T) = parallel(T_1) \cup parallel(T_2) \cup symcross(FTlabels(T_1), FTlabels(T_2))$. Using Lemma (7.14) gives us
12) $symcross(FTlabels(T_1), FTlabels(T_2)) \subseteq Tcross_p(T_1, Tlabels_p(T_2))$. From 8),9),10),11) and 12) we have $parallel(T) \subseteq M$.

If $T \equiv \langle s \rangle$ then from the definition of $parallel()$ we have $parallel(T) = \emptyset$ which makes $parallel(T) \subseteq M$ trivial. □

8.6 Soundness

We are now ready to prove Theorem 2, which we restate here:

> **Theorem (Soundness)** If $\vdash p : E$, $p, E, \emptyset \vdash \langle s_0 \rangle : M$ and $(p, A_0, \langle s_0 \rangle) \rightarrow^* (p, A, T)$ then $parallel(T) \subseteq M$.

Proof. We will first show that:

> Claim A: If $\vdash p : E$, $p, E, \emptyset \vdash \langle s_0 \rangle : M$ and $(p, A_0, \langle s_0 \rangle) \rightarrow^* (p, A, T)$, then there exists M' such that $p, E, \emptyset \vdash T : M'$ and $M' \subseteq M$.

It is sufficient to show that:

> Claim B: For all i: if $\vdash p : E$, $p, E, \emptyset \vdash \langle s_0 \rangle : M$ and $(p, A_0, \langle s_0 \rangle) \rightarrow^i (p, A, T)$, then there exists M' such that $p, E, \emptyset \vdash T : M'$ and $M' \subseteq M$.

We proceed by induction on i. In the base case of $i = 0$, we have $\langle s_0 \rangle = T$ and we can choose $M' = M$. From $p, E, \emptyset \vdash \langle s_0 \rangle : M$ and $\langle s_0 \rangle = T$ and $M' = M$, we immediately have $p, E, \emptyset \vdash T : M'$ and $M' \subseteq M$. In the induction step, suppose we have Claim B for a particular i, and consider $(p, A_0, \langle s_0 \rangle) \rightarrow^i (p, A, T) \rightarrow (p, A', T')$. From the induction hypothesis, we have M' such that $p, E, \emptyset \vdash T : M'$ and $M' \subseteq M$. From $\vdash p : E$, $p, E, \emptyset \vdash T : M'$ and $(p, A, T) \rightarrow (p, A', T')$ and Lemma (16), we have that there exists M'' such that $p, E, \emptyset \vdash T' : M''$ and $M'' \subseteq M'$. Finally, from $M'' \subseteq M'$ and $M' \subseteq M$, we have $M'' \subseteq M$. This completes the proof of Claim B and therefore the proof of Claim A.

To prove the soundness theorem itself, suppose $\vdash p : E$, $p, E, \emptyset \vdash \langle s_0 \rangle : M$ and $(p, A_0, \langle s_0 \rangle) \rightarrow^* (p, A, T)$. From $\vdash p : E$, $p, E, \emptyset \vdash \langle s_0 \rangle : M$ and $(p, A_0, \langle s_0 \rangle) \rightarrow^* (p, A, T)$ and Claim A, we have that there exists M' such that $p, E, \emptyset \vdash T : M'$ and $M' \subseteq M$. From $p, E, \emptyset \vdash T : M'$ and Lemma (17) we have $parallel(T) \subseteq M'$. Since $M' \subseteq M$, we have $parallel(T) \subseteq M$, as desired. □

Appendix C: Proof of Theorem 4

Let φ, ψ be valuations of the set variables in two, possibly different, constraints systems We say that φ, ψ *agree on their common domain,* if for all $v \in \text{dom}(\varphi) \cap \text{dom}(\psi) : \varphi(v) = \psi(v)$. If φ, ψ agree on their common domain, then we define

$$\varphi \cup \psi \quad = \quad \lambda v \in \text{dom}(\varphi) \cup \text{dom}(\psi). \begin{cases} \varphi(v) & \text{if } v \in \text{dom}(\varphi) \\ \psi(v) & \text{otherwise} \end{cases}$$

LEMMA 18. *$p, E, R \vdash s : M, O$ if and only if there exists a solution φ to $C(s)$ where $\varphi(r_s) = R$ and $\varphi(o_s) = O$ and $\varphi(m_s) = M$ and φ extends E.*

Proof. $\Leftarrow$) Let us now perform induction on s and examine the seven cases.

If $s \equiv skip^l$ then from constraints (60-61) we have 1) $\varphi(r_s) = \varphi(o_s)$ and 2) $\varphi(m_s) = Lcross(l, \varphi(r_s))$. Substituting the premise in 1) and 2) gives us 3) $R = O$ and 4) $M = Lcross(l, R)$. We may apply Rule (50) with 3) and 4) to get $p, E, R \vdash s : M, O$.

If $s \equiv skip^l\ s_1$ then from constraints (62-64) we have 1) $\varphi(r_s) = \varphi(r_{s_1})$, 2) $\varphi(o_s) = \varphi(o_{s_1})$ and 3) $\varphi(m_s) = Lcross(l, \varphi(r_s)) \cup \varphi(m_{s_1})$. Let 4) $\varphi(m_{s_1}) = M_1$. Substituting the premise and 4) in 1),2) and 3) gives us 5) $R = \varphi(r_{s_1})$, 6) $O = \varphi(o_{s_1})$ and 7) $M = Lcross(l, R) \cup M_1$. From the definition of $C(s)$ we have $C(s_1) \subseteq C(s)$. We see that since φ is a solution to $C(s)$, φ is also a solution to $C(s_1)$. Since φ is a solution to $C(s_1)$ and extends E we may use the induction hypothesis with 4),5) and 6) to get 8) $p, E, R \vdash s_1 : M_1, O$. Using 7) and 8) we may use Rule (51) to get $p, E, R \vdash s : M, O$.

If $s \equiv a[d] =^l e;\ s_1$ then we proceed using similar logic as the previous case.

If $s \equiv while^l\ (a[d] \neq 0)\ s_1\ s_2$ then from constraints (68-71) we have 1) $\varphi(r_s) = \varphi(r_{s_1})$, 2) $\varphi(r_{s_2}) = \varphi(o_{s_1})$, 3) $\varphi(o_s) = \varphi(o_{s_2})$ and 4) $\varphi(m_s) = Lcross(l, \varphi(o_{s_1})) \cup Scross_p(s_1, \varphi(o_{s_1})) \cup \varphi(m_{s_1}) \cup \varphi(m_{s_2})$. Let 5) $\varphi(o_{s_1}) = O_1$, 6) $\varphi(m_{s_1}) = M_1$ and 7) $\varphi(m_{s_2}) = M_2$. Substituting the premise, 5), 6) and 7) in 1),2),3), and 4) gives us 8) $R = \varphi(r_{s_1})$, 9) $\varphi(r_{s_2}) = O_1$, 10) $O = \varphi(o_{s_2})$ and 11) $M = Lcross(l, O_1) \cup Scross_p(s_1, O_1) \cup M_1 \cup M_2$. From the definition of $C(s)$ we have $C(s_1) \subseteq C(s)$ and $C(s_2) \subseteq C(s)$. Since φ is a solution to $C(s)$, φ is also a solution to both $C(s_1)$ and $C(s_2)$. Since φ is a solution to $C(s_1)$ and $C(s_2)$ and φ extends E we use the induction hypothesis with the 5),6),7),8),9) and 10) to get 12) $p, E, R \vdash s_1 : M_1, O_1$ and 13) $p, E, O_1 \vdash s_2 : M_2, O$. We may now apply Rule (53) with 11),12) and 13) to get $p, E, R \vdash s : M, O$.

If $s \equiv async^l\ s_1\ s_2$ then from constraints (72-75) we obtain 1) $\varphi(r_{s_1}) = Slabels_p(s_2) \cup \varphi(r_s)$, 2) $\varphi(r_{s_2}) = Slabels_p(s_1) \cup \varphi(r_s)$, 3) $\varphi(o_s) = \varphi(o_{s_2})$ and 4) $\varphi(m_s) = Lcross(l, \varphi(r_s)) \cup \varphi(m_{s_1}) \cup \varphi(m_{s_2})$. Let 5) $\varphi(m_{s_1}) = M_1$, 6) $\varphi(m_{s_2}) = M_2$ and 7) $\varphi(o_{s_1}) = O_1$. Substituting the premise, 5) and 6) in 1),2),3) and 4) gives us 8) $\varphi(r_{s_1}) = Slabels_p(s_2) \cup R$, 9) $\varphi(r_{s_2}) = Slabels_p(s_1) \cup R$, 10) $O = \varphi(o_{s_2})$ and 11) $M = Lcross(l, R) \cup M_1 \cup M_2$. From the definition of $C(s)$ we have $C(s_1) \subseteq C(s)$ and $C(s_2) \subseteq C(s)$. Since φ is a solution to $C(s)$, φ is also a solution to both $C(s_1)$ and $C(s_2)$. Since φ is a solution to $C(s_1)$ and $C(s_2)$ and φ extends E we use the induction hypothesis with the premise,5),6),7),8),9) and 10) to get 12) $p, E, Slabels_p(s_2) \cup R \vdash s_1 : M_1, O_1$ and 13) $p, E, Slabels_p(s_1) \cup R \vdash s_2 : M_2, O$. Using Rule (54) with 11),12) and 13) gives us $p, E, R \vdash s : M, O$.

If $s \equiv finish^l\ s_1\ s_2$ then from constraints (76-79) we get 1) $\varphi(r_{s_1}) = \varphi(r_s)$, 2) $\varphi(r_{s_2}) = \varphi(r_s)$, 3) $\varphi(o_{s_2}) = \varphi(o_s)$ and 4) $\varphi(m_s) = Lcross(l, \varphi(r_s)) \cup \varphi(m_{s_1}) \cup \varphi(m_{s_2})$. Let 5) $\varphi(m_{s_1}) = M_1$, 6) $\varphi(m_{s_2}) = M_2$ and 7) $\varphi(o_{s_1}) = O_1$. Substituting the premise, 5) and 6) in 1),2),3) and 4) results in 8) $\varphi(r_{s_1}) = R$, 9) $\varphi(r_{s_2}) = R$, 10) $\varphi(O_{s_2}) = O$ and 11) $M = Lcross(l, R) \cup M_1 \cup M_2$. From the definition of $C(s)$ we have $C(s_1) \subseteq C(s)$ and $C(s_2) \subseteq C(s)$. Since φ is a solution to $C(s)$, φ is also solution to both $C(s_1)$ and $C(s_2)$. Because φ is a solution to $C(s_1)$ and $C(s_2)$ and φ extends E we use the induction hypothesis with the premise,5),6),7),8),9) and 10) to get 12) $p, E, R \vdash s_1 : M_1, O_1$ and 13) $p, E, R \vdash s_2 : M_2, O$. We apply Rule (55) with 11),12) and 13) to get $p, E, R \vdash s : M, O$.

If $s \equiv f_i()\ k$ then from constraints (80-82) we have 1) $\varphi(r_k) = \varphi(r_s) \cup \varphi(o_i)$, 2) $\varphi(o_k) = \varphi(o_s)$ and 3) $\varphi(m_s) = Lcross(l, \varphi(r_s)) \cup symcross(Slabels_p(p(f_i)), \varphi(r_s)) \cup \varphi(m_i) \cup \varphi(m_k)$. Let 4) $\varphi(m_i) = M_i$ and 5) $\varphi(o_i) = O_i$. Let 6) $\varphi(m_k) = M_k$. Substituting the premise,4),5) and 6) in 1),2) and 3) gives us 7) $\varphi(r_k) = R \cup O_i$, 8) $\varphi(o_k) = O$ and 9) $M = Lcross(l, R) \cup symcross(Slabels_p(p(f_i)), R) \cup M_i \cup M_k$. From the definition of $C(s)$ we have $C(k) \subseteq C(s)$. Since φ is a solution $C(s)$, φ is also a solution to $C(k)$. Because φ is a solution to $C(k)$ and φ extends E, we may apply the induction hypothesis with 6),7) and 8) to get 10) $p, E, R \cup O_i \vdash k : M_k, O$. From the premise we have that φ extends E which gives us 11) $E(f_i) = (\varphi(m_i), \varphi(o_i))$. From 9),10) and 11) we may apply Rule (56) and obtain $p, E, R \vdash s : M, O$ as desired.

$\Rightarrow$) Let us perform induction on s and examine the seven cases.

If $s \equiv skip^l$ then by Rule (50) we have 1) $M = Lcross(l, R)$ and 2) $O = R$. Let us construct a solution φ that extends E and such that 3) $\varphi(r_s) = R$, 4) $\varphi(o_s) = O$ and 5) $\varphi(m_s) = M$. We substitute 1),2) and 3) in 4) and 5) to get 6) $\varphi(o_s) = R$ and 7) $\varphi(m_s) = Lcross(l, \varphi(r_s))$. From 8) and 9) we see constraints (60-61) are satisfied. From 3),4),5) we see that the other conditions of the conclusion are also satisfied.

If $s \equiv skip^l\ s_1$ then by Rule (51) we have 1) $p, E, R \vdash s_1 : M_1, O_1$, 2) $M = Lcross(l, R) \cup M_1$ and 3) $O = O_1$. From the induction hypothesis to 1) we have a solution φ_1 to $C(s_1)$ which extends E where 4) $\varphi_1(r_{s_1}) = R$, 5) $\varphi_1(o_{s_1}) = O_1$ and 6) $\varphi_1(m_{s_1}) = M_1$. Let 7) $\varphi = \varphi_1[r_s \mapsto R, m_s \mapsto M, o_s \mapsto O]$. From the definition of extension with 4),5) and 6) we have 8) $\varphi(r_s) = R$, 9) $\varphi(o_s) = O$, 10) $\varphi(m_s) = M$, 11) $\varphi(r_{s_1}) = R$, 12) $\varphi(o_{s_1}) = O_1$ and 13) $\varphi(m_{s_1}) = M_1$. From 8) and 11) we have 14) $\varphi(r_s) = \varphi(r_{s_1})$. From 3),9) and 12) we get 15) $\varphi(o_s) = \varphi(o_{s_1})$. From 2),8),10) and 13) we obtain 16) $\varphi(m_s) = Lcross(l, \varphi(r_s)) \cup \varphi(m_{s_1})$. Since φ extends φ_1, φ also extends E and is a solution to $C(s_1)$. With 14),15) and 16) we satisfy constraints (62-64) and thus φ is a solution to $C(s)$. From 8),9) and 10) we satisfy the additional conditions of the conclusion.

If $s \equiv a[d] =^l e;\ s_1$ then we proceed using similar logic as the previous case.

If $s \equiv while^l\ (a[d] \neq 0)\ s_1\ s_2$ then from Rule (53) we have 1) $p, E, R \vdash s_1 : M_1, O_1$, 2) $p, E, O_1 \vdash s_2 : M_2, O_2$, 3) $M = Lcross(l, O_1) \cup Scross_p(s_1, O_1) \cup M_1 \cup M_2$ and 4) $O = O_2$. Applying the induction hypothesis to 1) yields a solution φ_1 to $C(s_1)$ that extends E and 5) $\varphi_1(r_{s_1}) = R$, 6) $\varphi_1(o_{s_1}) = O_1$ and 7) $\varphi_1(m_{s_1}) = M_1$. Applying the induction hypothesis to 2) yields a solution φ_2 to $C(s_2)$ that extends E such that 8) $\varphi_2(r_{s_2}) = O_1$, 9) $\varphi_2(o_{s_2}) = O_2$ and 10) $\varphi_2(m_{s_2}) = M_2$. Notice that φ_1 and φ_2 agree on their common domain. Let 11) $\varphi = \varphi_1 \cup \varphi_2$. We have that φ is a solution to both $C(s_1)$ and $C(s_2)$, and that φ extends E. From our definition of φ, we have 12) $\varphi(r_{s_1}) = R$, 13) $\varphi(o_{s_1}) = O_1$, 14) $\varphi(m_{s_1}) = M_1$, 15) $\varphi(r_{s_2}) = O_1$, 16) $\varphi(o_{s_2}) = O_2$ and 17) $\varphi(m_{s_2}) = M_2$. From 11) we have 18) $\varphi(r_s) = R$, 19) $\varphi(o_s) = O$ and 20) $\varphi(m_s) = M$. From 12) and 18) we have 21) $\varphi(r_s) = \varphi(r_{s_1})$. From 13) and 15) we get 22) $\varphi(o_{s_1}) = \varphi(r_{s_2})$. Combining 3),13),14),17) and 20) gives us 23) $\varphi(m_s) = Lcross(l, \varphi(o_{s_1})) \cup Scross(s_1, \varphi(o_{s_1})) \cup \varphi(m_{s_1}) \cup \varphi(m_{s_2})$. We use 4),16) and 19) to get 24) $\varphi(o_s) = \varphi(o_{s_2})$. We see from constraints (68-71) are satisfied by 21),22),23) and 24) and since φ satisfies $C(s_1)$ and $C(s_2)$ it is a solution to $C(s)$. From 18),19) and 20) we satisfy the other conditions of the conclusion.

If $s \equiv async^l\ s_1\ s_2$ then from Rule (54) we have
1) $p, E, Slabels_p(s_2) \cup R \vdash s_1 : M_1, O_1$, 2) $p, E, Slabels_p(s_1) \cup R \vdash s_2 : M_2, O_2$, 3) $M = Lcross(l, R) \cup M_1 \cup M_2$ 4) $O = O_2$. Applying the induction hypothesis to 1) yields a solution φ_1 to $C(s_1)$ that extends E and 5) $\varphi_1(r_{s_1}) = Slabels_p(s_2) \cup R$, 6) $\varphi_1(o_{s_1}) = O_1$ and 7) $\varphi_1(m_{s_1}) = M_1$. Applying the induction hypothesis to 2) yields a solution φ_2 to $C(s_2)$ that extends E and 8) $\varphi_2(r_{s_2}) = Slabels_p(s_1) \cup R$, 9) $\varphi_2(o_{s_2}) = O_2$ and 10) $\varphi_2(m_{s_2}) = M_1$. Notice that φ_1 and φ_2 agree on their common domain. Let 11) $\varphi = \varphi_1 \cup \varphi_2$. We have that φ is a solution to both $C(s_1)$ and $C(s_2)$, and that φ extends E. We will now show that φ is a solution to $C(s)$. From our definition of φ we have 12) $\varphi(r_{s_1}) = Slabels_p(s_2) \cup R$, 13) $\varphi(o_{s_1}) = O_1$, 14) $\varphi(m_{s_1}) = M_1$, 15) $\varphi(r_{s_2}) = Slabels_p(s_1) \cup R$, 16) $\varphi(o_{s_2}) = O_2$ and 17) $\varphi(m_{s_2}) = M_2$. From 11) we have 18) $\varphi(r_s) = R$, 19) $\varphi(o_s) = O$ and 20) $\varphi(m_s) = M$. From 12) and 18) we have 21) $\varphi(r_{s_1}) = Slabels_p(s_2) \cup \varphi(r_s)$. From 15) and 18) we get 22) $\varphi(r_{s_2}) = Slabels_p(s_1) \cup \varphi(r_s)$. Combining 3),14),17),18) and 20) gives us 23) $\varphi(m_s) = Lcross(l, \varphi(r_s)) \cup \varphi(m_{s_1}) \cup \varphi(m_{s_2})$. We use 4),16) and 19) to get 24) $\varphi(o_s) = \varphi(o_{s_2})$. We see from constraints (72-75) are satisfied by 21),22),23) and 24) and since φ satisfies $C(s_1)$ and $C(s_2)$ it is a solution to $C(s)$. From 18),19) and 20) we satisfy the other conditions of the conclusion.

If $s \equiv finish^l\ s_1\ s_2$ then from Rule (55) we get 1) $p, E, R \vdash s_1 : M_1, O_1$, 2) $p, E, R \vdash s_2 : M_2, O_2$, 3) $M = Lcross(l, R) \cup M_1 \cup M_2$ and 4) $O = O_2$. Applying the induction hypothesis to 1) yields a solution φ_1 to $C(s_1)$ that extends E and 5) $\varphi_1(r_{s_1}) = R$, 6) $\varphi_1(o_{s_1}) = O_1$ and 7) $\varphi_1(m_{s_1}) = M_1$. Applying the induction hypothesis to 2) yields a solution φ_2 to $C(s_2)$ that extends E and 8) $\varphi_2(r_{s_2}) = R$, 9) $\varphi_2(o_{s_2}) = O_2$ and 10) $\varphi_2(m_{s_2}) = M_2$. Notice that φ_1 and φ_2 agree on their common domain. Let 11) $\varphi = \varphi_1 \cup \varphi_2$. We have that φ is a solution to both $C(s_1)$ and $C(s_2)$, and that φ extends E. We will now show that φ is a solution to $C(s)$. From our definition of φ we have 12) $\varphi(r_{s_1}) = R$, 13) $\varphi(o_{s_1}) = O_1$, 14) $\varphi(m_{s_1}) = M_1$, 15) $\varphi(r_{s_2}) = R$, 16) $\varphi(o_{s_2}) = O_2$ and 17) $\varphi(m_{s_2}) = M_2$. From 11) we have 18) $\varphi(r_s) = R$, 19) $\varphi(o_s) = O$ and 20) $\varphi(m_s) = M$. From 12) and 18) we have 21) $\varphi(r_s) = \varphi(r_{s_1})$. From 15) and 18) we get 22) $\varphi(r_s) = \varphi(r_{s_2})$. Combining 3),14),17),18) and 20) gives us 23) $\varphi(m_s) = Lcross(l, \varphi(r_s)) \cup \varphi(m_{s_1}) \cup \varphi(m_{s_2})$. We use 4),16) and 19) to get 24) $\varphi(o_s) = \varphi(o_{s_2})$. We see from constraints (76-79) are satisfied by 21),22),23) and 24) and since φ satisfies $C(s_1)$ and $C(s_2)$ it is a solution to $C(s)$. From 18),19) and 20) we satisfy the other conditions of the conclusion.

If $s \equiv f_i()\ k$ then we have 1) $E(f_i) = (M_i, O_i)$, 2) $p, E, R \cup O_i \vdash k : M_k, O_k$,
3) $M = Lcross(l, R) \cup symcross(Slabels_p(p(f_i)), R) \cup M_i \cup M_k$ and 4) $O = O_k$. We may apply the induction hypothesis on 2) to get a solution φ_k to $C(k)$ that extends E and 5) $\varphi_k(r_k) = R \cup O_i$, 6) $\varphi_k(o_k) = O_k$ and 7) $\varphi_k(m_k) = M_k$. Let 8) $\varphi = \varphi_k[r_s \mapsto R, m_s \mapsto M, o_s \mapsto O]$. From the definition of extension with 5),6) and 7) we have 9) $\varphi(r_s) = R$, 10) $\varphi(o_s) = O$, 11) $\varphi(m_s) = M$, 12) $\varphi(r_k) = R \cup O_i$, 13) $\varphi(o_k) = O_k$ and 14) $\varphi(m_k) = M_k$. Since φ extends E we use the definition of extension with 1) to get 15) $\varphi(o_i) = O_i$ and 16) $\varphi(m_i) = M_i$. From 12) and 15) we get 17) $\varphi(r_k) = \varphi(r_s) \cup \varphi(o_i)$. Using 4),10) and 13) we obtain 18) $\varphi(o_s) = \varphi(o_k)$. Combining 3),7),8),9),11) and 16) we get 19) $\varphi(m_s) = Lcross(l, \varphi(r_s)) \cup symcross(Slabels_p(p(f_i)), \varphi(r_s)) \cup \varphi(m_i) \cup \varphi(m_k)$. We let φ be our solution as we see that it is a solution to $C(s_1)$ and satisfies constraints (80-82) and thus is a solution to $C(s)$. Additionally, from 10) we also see that φ also extends E. From 9),10) and 11) we satisfy the remaining conditions of the conclusion. □

We are now ready to prove Theorem 4, which we restate here:

Theorem (Equivalence) $\vdash p : E$ if and only if there exists a solution φ of $C(p)$ where φ extends E.

Proof. $\Leftarrow$) We have a solution φ of $C(p)$ where φ extends E. From constraints (57-59) we have for all f_i defined in p, 1) $\varphi(r_{s_i}) = \emptyset$, 2) $\varphi(o_i) = \varphi(o_{s_i})$ and 3) $\varphi(m_i) = \varphi(m_{s_i})$. Substituting the premise that φ extends E in 2) and 3) gives us 4) $\varphi(o_{s_i}) = O_i$ and 5) $\varphi(m_{s_i}) = M_i$. Since $C(s_i) \subseteq C(p)$, we see that φ is a solution to $C(s_i)$. Since φ is a solution to $C(s_i)$ and φ extends E then using Lemma (18) with 1),4),5) and the premise we get for each i, 6) $p, E, \emptyset \vdash s_i : M_i, O_i$. Since φ extends E then for all i 7) $E(f_i) = (\varphi(m_i), \varphi(o_i))$. Substituting 2),3),4) and 5) in 7) gives us 8) $E(f_i) = (M_i, O_i)$ which is we can rewrite as 9) $E = \{\ f_i \mapsto (M_i, O_i)\ \}$. From 6) and 9) we may use Rule (45) to get $\vdash p : E$ as desired.

$\Rightarrow$) From Rule (45) we have 1) $E = \{f_i \mapsto (M_i, O_i)\}$ and for all i 2) $p, E, \emptyset \vdash s_i : M_i, O_i$. We apply for each i Lemma (18) to 2) to get a solution φ_i to $C(s_i)$ that extends E and 3) $\varphi_i(r_{s_i}) = \emptyset$, 4) $\varphi_i(o_{s_i}) = O_i$ and 5) $\varphi_i(m_{s_i}) = M_i$. Notice that all the φ_i agree on their common domain. Let 6) $\varphi = \bigcup_i \varphi_i$, We will now show that φ is a solution to $C(p)$. We have that φ is a solution to each $C(s_i)$ and that φ extends E. All we must show then is that constraints (57-59) are satisfied. From 1) we see 7) $E(f_i) = (M_i, O_i)$. Since φ extends E and using the definition of extends we get 8) $E(f_i) = (\varphi(m_i), \varphi(o_i))$. From 7) and 8) we get 9) $\varphi(m_i) = M_i$ and 10) $\varphi(o_i) = O_i$. From 3) and 6) we get 11) $\varphi(r_{s_i}) = \emptyset$. Using 4),6) and 10) we have 12) $\varphi(o_{s_i}) = O_i$. From 5),6) and 9) we have 13) $\varphi(m_{s_i}) = M_i$. Substituting 10) in 12) gives us 14) $\varphi(o_{s_i}) = \varphi(o_i)$. We substitute 9) in 13) to get 15) $\varphi(m_{s_i}) = \varphi(m_i)$. From 11),14) and 15) we see that constraints (57-59) are satisfied and since φ extends E we have reached our conclusion. □