Towards a Practical Secure Concurrent Language

Stefan Muller
and
Stephen Chong

TR-05-12

Computer Science Group
Harvard University
Cambridge, Massachusetts

Towards a Practical Secure Concurrent Language [*]

Stefan Muller
Carnegie Mellon University
smuller@cs.cmu.edu

Stephen Chong
Harvard University
chong@seas.harvard.edu

Abstract

We demonstrate that a practical concurrent language can be extended in a natural way with information security mechanisms that provably enforce strong information security guarantees. We extend the X10 concurrent programming language with coarse-grained information-flow control. Central to X10 concurrency abstractions is the notion of a *place*: a container for data and computation. We associate a security level with each place, and restrict each place to store only data appropriate for that security level. When places interact only with other places at the same security level, then our security mechanisms impose no restrictions. When places of differing security levels interact, our information security analysis prevents potentially dangerous information flows, including information flow through covert scheduling channels. The X10 concurrency mechanisms simplify reasoning about information flow in concurrent programs.

We present a static analysis that enforces a noninterference-based extensional information security condition in a calculus that captures the key aspects of X10's place abstraction and async-finish parallelism. We extend this security analysis to support many of X10's language features, and have implemented a prototype compiler for the resulting language.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.4.6 [*Operating Systems*]: Security and Protection—Information flow controls

Keywords Language-based security; information-flow control; X10.

1. Introduction

Enforcement of strong information security guarantees for concurrent programs poses both a challenge and an opportunity. The challenge is that, given current hardware trends towards increased parallelism, and the large number of computer systems that handle data of varying sensitivity, it is increasingly important to reason about and enforce information security guarantees in the presence of concurrency. Although progress has been made towards this end, there are not yet practical enforcement mechanisms and usable implementations. The opportunity is to adapt new and existing language abstractions for concurrency to reason precisely about information security in concurrent programs. Information security, like concurrency, is intimately connected to notions of dependency [1]. As such, there is potential for synergy between language mechanisms for concurrency, and enforcement mechanisms for information security in concurrent programs.

The X10 programming language [10] is an object-oriented language with abstractions to support fine-grained concurrency. Central to X10 concurrency abstractions is the notion of a *place*. A place is a computational unit that contains computation and data. For example, each core of a single machine or each machine within a distributed system might be represented by a different place. Data held at a place, and computation running at a place, are said to be "local" to the place. Places are first-class in X10. Multiple threads of execution, which in X10 are known as *activities*, may execute concurrently within a place. Activities at the same place share memory, and an activity may only access data at the place where it is located. Places may communicate using message passing, but X10 is designed to discourage excessive communication between places, since this reduces concurrency.

We extend X10 language abstractions for concurrency with information security mechanisms, and call the resulting language SX10 (for Secure X10). Specifically, in SX10 each place is associated with a security level, and a (completely static) security analysis ensures that each place stores only data appropriate for that security level. Thus, all computation within a place is on data at the same security level. In the case where places communicate only with other places at the same security level, then our security mechanisms do not impose any restrictions on programs. Communication between places with different security levels may pose security concerns, but because message-passing communication is used between places, it is relatively simple to restrict such communication: the security analysis ensures that data may be sent to another place only when the security level of the destination is sufficiently high. Interaction between places may

[*] This technical report is an extended version of the paper of the same title that appeared in OOPSLA'12. This technical report provides additional proof details.

influence the scheduling of activities at a place, leading to potential covert information channels; our security analysis tracks and controls these covert channels.

We believe that this coarse-grained approach to providing information security in concurrent programs is simple, practical, and useful. All data at a place is at the same security level, which both provides simplicity of reasoning for the programmer, and allows a high degree of concurrency within a place without compromising security.

There are many highly concurrent systems that compute with data of varying sensitivity that fit naturally into such a model. The following are some examples.

- **Machine learning** A service such as Pandora processes a large amount of public data, which is then used to make recommendations to individual users based on their private usage data. All public data is at the same security level and processing it is highly parallel; data from many users are processed in parallel.
- **Social networks** Users specify that some posts are visible to all other users, and some are visible only to friends. Many users may use the system concurrently.
- **Shopping carts** An online shopping cart collects information about items ordered, which may appear in low-security logs, and credit card information, which must remain secure. Many customers may use the system concurrently.

Motivating example Reasoning about information security in the presence of concurrency can be subtle. Consider Program 1, which exhibits a timing channel.

```
at Low {
    async {
        // Activity 1
        if (hi > 0) longComputation();
        output “pos”;
    }
    // Activity 2
    mediumComputation(); output “nonpos”;
}
```

Program 1.

Assume that memory location hi contains high-security (e.g., secret) data. Instruction at Low s indicates that s is executed at a place called Low, which we assume is allowed to handle only low-security data. Outputs at place Low should not reveal high-security information. Instruction async s creates a new activity to execute statement s, and the current activity continues with the following statement. Thus, the if statement and subsequent output of the string “pos” execute concurrently with the output “nonpos” statement. If high-security memory location hi is positive, then Activity 1 outputs “pos” after a long time; otherwise it outputs “pos” immediately. Activity 2 computes for a medium amount of time, and outputs “nonpos”. It is likely that the order of outputs will reveal secret information, and the program is thus insecure. This is an example of an *internal timing channel* [47], where the order of publicly observable events depends upon high-security information.

Program 1 is not an SX10 program: low-security place Low is not allowed to hold high-security data, such as that stored in memory location hi. Suppose, however, that High is a place that is permitted to hold secret information. Then Program 2 is a SX10 program and exhibits a similar timing channel. (It is correctly rejected by our security analysis.)

```
at Low {
    async {
        // Activity 1
        at High { // activity 1 moves to High
            if (hi > 0) longComputation();
        }
        // activity 1 moves back to Low
        output “pos”;
    }
    // Activity 2
    mediumComputation(); output “nonpos”;
}
```

Program 2.

In this example, Activity 1 moves to place High in order to perform computation on high-security data, before returning to place Low to perform the output of the string “pos”.

We assume that the scheduling of activities at each place depends only on the activities at that place, an assumption that holds in the X10 2.2 runtime [14]. Nonetheless, in Program 2, the scheduling of Activity 2 at place Low depends on whether Activity 1 is running at Low, which in turn depends on how long the computation at place High takes. Thus, the scheduling of output “pos” and output “nonpos” may be influenced by high-security data. Our security analysis detects this potential information flow, and rejects the program.

Program 2 is inherently nondeterministic: it could perform the two outputs in either order. The scheduler resolves the nondeterminism, but in doing so, may reveal high-security information—a form of refinement attack [38]. One way to prevent such refinement attacks is to require that any observable behavior be deterministic [6, 48]. Our security analysis requires such observational determinism when the resolution of nondeterminism may reveal high-security information.

It is, however, possible to allow some observable nondeterminism within a secure concurrent program. Intuitively, if the resolution of the nondeterminism does not depend on high-security information, then observable nondeterminism is secure [30]. If place P does not communicate with any other places, then, since scheduling is performed per-place, resolution of nondeterminism at P will not reveal high-security information. In some cases it is also possible to allow nondeterminism at place P even if P interacts with places of higher security levels. Consider Program 3: Activity 1 executes at place Low concurrently with Activity 2's execution at place High. The finish s instruction executes statement s, and waits until s and all activities spawned by s have terminated before continuing execution. Thus, Activity 3 and Activity 4 execute concurrently at place Low after Activities 1 and 2 have finished. Although the order of output “B” and output “C” is nondeterministic, resolution of this nondeterminism is not influenced by how long the high-security computation takes, and so does not reveal high-security information.

```
at Low {
  finish {
    async {
      // Activity 1
      output “A”;
    }
    // Activity 2
    at High {
      if (hi > 0) longComputation();
    }
  }
  async {
    // Activity 3
    output “B”;
  }
  // Activity 4
  output “C”;
}
```

Program 3.

Contributions The key contribution of this paper is to demonstrate that practical and useful concurrency mechanisms can be extended in a natural way with information security mechanisms that provably enforce strong information security guarantees. We enforce coarse-grained information-flow control [38], requiring that every place can store only data at a single security level. If places interact only with other places at the same security level, then our security mechanisms do not restrict concurrency nor require determinism for security. When places of differing security interact, our information security analysis prevents potentially dangerous information flows by using X10's concurrency mechanisms to reason both about data sent between places, and about how the scheduling of activities at a place may depend on high-security information.

In Section 2 we present a calculus, based on Featherweight X10 [22], that captures key aspects of the X10 place abstraction, and its async-finish parallelism. We define a knowledge-based noninterference semantic security condition [2, 12] for this calculus in Section 3, and present a security analysis that provably enforces it. The language SX10 is the result of extending this analysis to handle many of the language features of X10. We have implemented a prototype compiler for SX10 by modifying the X10 compiler, and this is described in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2. FSX10: a secure parallel calculus

In this section, we introduce the calculus FSX10, based on Featherweight X10 [22]. Like Featherweight X10, this calculus captures X10's async-finish parallelism, but adds places and interaction with the external environment via input and output instructions.

2.1 Syntax

The abstract syntax of FSX10 is presented in Figure 1. A *place* P is a container for data and activities. In FSX10, as in X10, every memory location r and every activity is associated with a place. In FSX10, however, places are simply identifiers and are not first-class values. Function $Place(\cdot)$ describes how memory locations are mapped to places: memory location r is held at place $Place(r)$, and only code executing at that place is allowed to access the location.

For simplicity, we restrict values in the calculus to integers. Expressions e consist of integer constants v, variables x, memory reads $!r$ (where r is a memory location), and total binary operations over expressions $e_1 \oplus e_2$.

Statements s are sequences of instructions. Every instruction is labeled with a program point. For example, a store instruction $r :=^p e$ has program point p. For convenience we write s^p to indicate that program point p is the program point of the first instruction of statement s. When the program point of an instruction is irrelevant, we omit it.

Instructions include no-ops (skip), selection (if e then s else s), and iteration (while e do s). Instruction $r := e$ evaluates expression e and updates memory location r with the result. Instruction let $x = e$ in s evaluates expression e to a value, and uses that value in place of variable x in the evaluation of statement s. Once defined, variable x is immutable. A variable defined at place P may be used at a different place P', which can be thought of as P sending the value of the variable to P'.

Instruction async s creates a new activity that starts executing statement s, and the current activity continues executing the next instruction. Instruction at $P\,s$ executes statement s at place P. Note that at $P\,s$ does not create a new activity: execution of the next instruction waits until s has terminated. That is, given the statement at $P\,s; r := 42;$ skip, the assignment of 42 to memory location r will not occur until after statement s has finished execution. Instruction backat Ps does not appear in source programs, but is used by the operational semantics to track when control will return back to place P as a result of finishing an at $P'\,s'$ instruction. Finally, instruction finish s will block until statement s, and all activities created by s, have terminated.

Metavariables

P	ranges over places
p	ranges over program points
v	ranges over integer constants
x	ranges over program variables
r	ranges over memory locations
$\oplus$	ranges over total binary integer functions

Expressions

e	$::=$	v	Integer constant
	$\mid$	x	Variable
	$\mid$	$!r$	Memory read
	$\mid$	$e \oplus e$	Binary operation

Statements

s	$::=$	skip^p	No-op
	$\mid$	$i; s$	Sequence

Instructions

i	$::=$	skip^p	No-op
	$\mid$	$r :=^p e$	Memory write
	$\mid$	$\mathsf{if}^p\ e\ \mathsf{then}\ s\ \mathsf{else}\ s$	Conditional
	$\mid$	$\mathsf{while}^p\ e\ \mathsf{do}\ s$	Iteration
	$\mid$	$\mathsf{let}^p\ x = e\ \mathsf{in}\ s$	Let
	$\mid$	$\mathsf{output}^p\ e$	Output
	$\mid$	$\mathsf{input}^p\ r$	Input
	$\mid$	$\mathsf{async}^p\ s$	Asynchronous
	$\mid$	$\mathsf{at}^p\ P\ s$	At
	$\mid$	$\mathsf{finish}^p\ s$	Finish
	$\mid$	$\mathsf{backat}^p\ P$	Back at

Trees

T	$::=$	$\langle P, s \rangle$	Activity
	$\mid$	$T \,\|\, T$	Parallel
	$\mid$	$T \triangleright \langle P, s \rangle$	Join
	$\mid$	$\checkmark$	Done

Figure 1. FSX10 syntax

FSX10 programs can communicate with the external environment via input and output instructions. We assume, without loss of generality, that every place has a single communication channel. Instruction $\mathsf{output}\ e$, when executed at place P, will evaluate expression e to a value, and output that value on P's channel. Similarly, instruction $\mathsf{input}\ r$, when executed at P, will input a value from P's channel, and store the result in location r. We assume that there is always data available for input on a channel, and thus input instructions are non-blocking.

Concurrently executing activities in FSX10 are represented using *trees*. Tree $\langle P, s \rangle$ is an activity at place P executing statement s. Tree $T_1 \,\|\, T_2$ represents trees T_1 and T_2 executing concurrently. Tree $\checkmark$ indicates a terminated activity, and tree $T \triangleright \langle P, s \rangle$ indicates that activity $\langle P, s \rangle$ is blocked until all activities in T have terminated.

2.2 Events, traces, and input strategies

As a program executes, it generates input and output events. Input event $i(v, P)$ is generated when an input instruction accepts value v from P's channel. Output event $o(v, P)$ is generated when an output instruction outputs value v on P's channel.

A *trace* t is a (possibly empty) sequence of input, output and location assignment events. Other events are not tracked. We write ϵ for the empty trace. We write $t\restriction_P$ for the subsequence of events of t that occur at place P. More formally, we have

$$\epsilon\restriction_P = \epsilon$$

$$(t \cdot \alpha)\restriction_P = \begin{cases} (t\restriction_P) \cdot \alpha & \text{if } \mathit{Place}(\alpha) = P \\ t\restriction_P & \text{otherwise} \end{cases}$$

where function $\mathit{Place}(\alpha)$ is the place at which α occurred:

$$\mathit{Place}(i(v, P)) = P$$
$$\mathit{Place}(o(v, P)) = P.$$

We model input from the external environment with *input strategies* [30]. Input strategy ω is a function from places and traces to values, such that given trace t, value $\omega(P, t\restriction_P)$ is the next value to input on the channel for P. Note that the choice of the next value that will be input on a channel can depend on the previous outputs of the channel. In Section 3, where we consider the security of FSX10 programs, we will be concerned with ensuring that low-security attackers are unable to learn about the inputs to high-security places.

2.3 Scheduling

Since program execution, and information security, depends on scheduling, we model the scheduler in FSX10. We explicitly refine the nondeterminism inherent in scheduling using *refiners* [30] to represent the decisions made by the scheduler. Essentially, all nondeterminism in program execution is encapsulated in a refiner; once a refiner has been chosen, program execution is deterministic.

In X10, a place represents a distinct computational node with a distinct scheduler [14]. In accordance with this model, we assume that scheduling decisions are made on a per-place basis, and the choice of which activity to run at a given place depends only on the set of activities currently executing at that place.

We model these assumptions by representing a refiner R as a pair (Ps, Sch), where Ps is a stream of places indicating the order in which places take steps, and Sch is a function from places to streams chs of *scheduling functions*. A scheduling function ch takes a set of program points (representing the set of activities currently executing at the

Place

$$\frac{\begin{array}{cc} \mathit{Sch}(P) = \mathit{chs} \cdot \mathit{ch} & \mathsf{PointsRunning}(T, P) \neq \emptyset \\ \mathit{ch}(\mathsf{PointsRunning}(T, P)) = p & (H;\, \omega;\, t;\, T) \xrightarrow{p} (H';\, \omega;\, t';\, T') \end{array}}{(H; \omega; (P \cdot Ps, \mathit{Sch}); t; T) \rightarrow (H'; \omega; (Ps, \mathit{Sch}[P \rightarrow \mathit{chs}]); t'; T')}$$

IdlePlace

$$\frac{\mathsf{PointsRunning}(T, P) = \emptyset}{(H; \omega; (P \cdot Ps, \mathit{Sch}); t; T) \rightarrow (H; \omega; (Ps, \mathit{Sch}); t; T)}$$

$$\begin{aligned} \mathsf{PointsRunning}(\checkmark, P) &= \emptyset \\ \mathsf{PointsRunning}(\langle P', s^p \rangle, P) &= \begin{cases} \{p\} & \text{if } P' = P \\ \emptyset & \text{if } P' \neq P \end{cases} \\ \mathsf{PointsRunning}(T_1 \,||\, T_2, P) &= \mathsf{PointsRunning}(T_1, P) \cup \mathsf{PointsRunning}(T_2, P) \\ \mathsf{PointsRunning}(T \triangleright \langle P', s \rangle, P) &= \mathsf{PointsRunning}(T, P) \end{aligned}$$

Figure 2. Program semantics $(H; \omega; R; t; T) \rightarrow (H'; \omega; R'; t'; T')$

place), and returns an element of that set (representing which of the activities should be scheduled). We write $P \cdot Ps$ for a stream with first element P and remaining elements Ps.

Thus, if the refiner is $(P \cdot Ps, \mathit{Sch})$, then place P will take a step next, and if $\mathit{Sch}(P) = \mathit{ch} \cdot \mathit{chs}$ (where ch is the first element of the stream of scheduling function, and chs is the remainder of the stream), then scheduling function ch will be used to determine which of the current activities at P will be scheduled. Note that each time a place takes a step, it may use a different scheduling function. However, the sequence of scheduling functions at a given place must be decided in advance, and may not depend on the history of computation at the place.

The use of a stream of scheduling functions per place allows our model to capture many realistic scheduling algorithms, such as round robin, shortest remaining time, and fixed priority. Scheduling algorithms that depend on the history of computation at a place (such as the work-stealing scheduling algorithm used in the X10 runtime [14, 15]) cannot be directly represented in this model. However, we believe the security guarantees still hold for the X10 runtime; we further discuss the security of the X10 scheduler in Section 4.

2.4 Operational semantics

A program configuration is a 5-tuple $(H; \omega; R; t; T)$. Heap H maps locations r to values, and is updated as the program executes. Input strategy ω is used to determine values input on channels; the input strategy does not change during execution, but we include it in the program configuration for notational convenience. Refiner R is used to determine scheduling, and is updated during execution. Trace t is the trace (of input, output, and location assignment events) produced so far by the program's execution. Tree T is the tree of currently executing activities.

The small-step operational semantics relation

$$(H; \omega; R; t; T) \rightarrow (H'; \omega; R'; t'; T')$$

describes how a program configuration changes as a result of execution. Due to the use of refiners, the operational semantics is deterministic. Inference rules for this relation are given in Figure 2.

Const

$$\frac{}{P; H; v \Downarrow v}$$

Read

$$\frac{H(r) = v \qquad \mathit{Place}(r) = P}{P; H; !r \Downarrow v}$$

Op

$$\frac{P; H; e_1 \Downarrow v_1 \qquad P; H; e_2 \Downarrow v_2}{P; H; e_1 \oplus e_2 \Downarrow v_1 \oplus v_2}$$

Figure 4. Expression semantics $P; H; e \Downarrow v$

Rule Place uses the refiner to select a place P to execute, and to select a scheduling function ch to schedule an activity at P. Set $\mathsf{PointsRunning}(T, P)$ is the set of program points of running activities located at P (also defined in Figure 2), which is given to scheduling function ch to select an activity. Rule IdlePlace handles the case where the refiner has selected place P to execute, but P does not have any currently running activities. Judgment

$$(H;\, \omega;\, t;\, T) \xrightarrow{p} (H';\, \omega;\, t';\, T')$$

is used to indicate that tree configuration $(H;\, \omega;\, t;\, T)$ executes the instruction at program point p to produce tree configuration $(H';\, \omega;\, t';\, T')$. Tree configurations are similar to program configurations, but omit the refiner, since the refiner is used only to determine which activity to execute.

Inference rules for $(H;\, \omega;\, t;\, T) \xrightarrow{p} (H';\, \omega;\, t';\, T')$ are given in Figure 3. Rules ParaLeft, ParaRight, ParaLeftDone, ParaRightDone, Join, and JoinDone navigate through the tree structure to find the appropriate activity to execute. Rule Skip1 reduces a skip statement to a terminated activity $\checkmark$. The remaining rules execute a single instruction.

Several of the rules for evaluating instructions evaluate expressions to values, using judgment $P; H; e \Downarrow v$, which

ParaLeft

$$\frac{(H;\,\omega;\,t;\,T_1) \xrightarrow{p} (H';\,\omega;\,t';\,T_1') \qquad T_1' \neq \checkmark}{(H;\,\omega;\,t;\,T_1 \,||\, T_2) \xrightarrow{p} (H';\,\omega;\,t';\,T_1' \,||\, T_2)}$$

ParaRight

$$\frac{(H;\,\omega;\,t;\,T_2) \xrightarrow{p} (H';\,\omega;\,t';\,T_2') \qquad T_2' \neq \checkmark}{(H;\,\omega;\,t;\,T_1 \,||\, T_2) \xrightarrow{p} (H';\,\omega;\,t';\,T_1 \,||\, T_2')}$$

ParaLeftDone

$$\frac{(H;\,\omega;\,t;\,T_1) \xrightarrow{p} (H';\,\omega;\,t';\,\checkmark)}{(H;\,\omega;\,t;\,T_1 \,||\, T_2) \xrightarrow{p} (H';\,\omega;\,t';\,T_2)}$$

ParaRightDone

$$\frac{(H;\,\omega;\,t;\,T_2) \xrightarrow{p} (H';\,\omega;\,t';\,\checkmark)}{(H;\,\omega;\,t;\,T_1 \,||\, T_2) \xrightarrow{p} (H';\,\omega;\,t';\,T_1)}$$

Join

$$\frac{(H;\,\omega;\,t;\,T) \xrightarrow{p} (H';\,\omega;\,t';\,T') \qquad T' \neq \checkmark}{(H;\,\omega;\,t;\,T \triangleright \langle P, s\rangle) \xrightarrow{p} (H';\,\omega;\,t';\,T' \triangleright \langle P, s\rangle)}$$

JoinDone

$$\frac{(H;\,\omega;\,t;\,T) \xrightarrow{p} (H';\,\omega;\,t';\,\checkmark)}{(H;\,\omega;\,t;\,T \triangleright \langle P, s\rangle) \xrightarrow{p} (H';\,\omega;\,t';\,\langle P, s\rangle)}$$

Skip1

$$\frac{}{(H;\,\omega;\,t;\,\langle P, \mathsf{skip}^p\rangle) \xrightarrow{p} (H;\,\omega;\,t;\,\checkmark)}$$

Skip2

$$\frac{}{(H;\,\omega;\,t;\,\langle P, \mathsf{skip}^p; s\rangle) \xrightarrow{p} (H;\,\omega;\,t;\,\langle P, s\rangle)}$$

Write

$$\frac{P; H; e \Downarrow v \qquad \mathit{Place}(r) = P}{(H;\,\omega;\,t;\,\langle P, r :=^p e; s\rangle) \xrightarrow{p} (H[r \mapsto v];\,\omega;\,t;\,\langle P, s\rangle)}$$

Let

$$\frac{P; H; e \Downarrow v \qquad s_1' = s_1\{v/x\}}{(H;\,\omega;\,t;\,\langle P, \mathsf{let}^p\, x = e\, \mathsf{in}\, s_1; s_2\rangle) \xrightarrow{p} (H;\,\omega;\,t;\,\langle P, s_1' \bullet s_2\rangle)}$$

Async

$$\frac{}{(H;\,\omega;\,t;\,\langle P, \mathsf{async}^p\, s_1; s_2\rangle) \xrightarrow{p} (H;\,\omega;\,t;\,\langle P, s_1\rangle \,||\, \langle P, s_2\rangle)}$$

Finish

$$\frac{}{(H;\,\omega;\,t;\,\langle P, \mathsf{finish}^p\, s_1; s_2\rangle) \xrightarrow{p} (H;\,\omega;\,t;\,\langle P, s_1\rangle \triangleright \langle P, s_2\rangle)}$$

At

$$\frac{}{(H;\,\omega;\,t;\,\langle P_1, \mathsf{at}^p\, P_2\, s_1; s_2\rangle) \xrightarrow{p} (H;\,\omega;\,t;\,\langle P_2, s_1 \bullet (\mathsf{backat}\, P_1; s_2)\rangle)}$$

BackAt

$$\frac{}{(H;\,\omega;\,t;\,\langle P_2, \mathsf{backat}^p\, P_1; s\rangle) \xrightarrow{p} (H;\,\omega;\,t;\,\langle P_1, s\rangle)}$$

Output

$$\frac{P; H; e \Downarrow v}{(H;\,\omega;\,t;\,\langle P, \mathsf{output}^p\, e; s\rangle) \xrightarrow{p} (H;\,\omega;\,t \cdot o(v, P);\,\langle P, s\rangle)}$$

Input

$$\frac{\mathit{Place}(r) = P \qquad \omega(P, t\restriction_P) = v}{(H;\,\omega;\,t;\,\langle P, \mathsf{input}^p\, r; s\rangle) \xrightarrow{p} (H[r \mapsto v];\,\omega;\,t \cdot i(v, P);\,\langle P, s\rangle)}$$

If1

$$\frac{P; H; e \Downarrow v \qquad v \neq 0}{(H;\,\omega;\,t;\,\langle P, \mathsf{if}^p\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2; s_3\rangle) \xrightarrow{p} (H;\,\omega;\,t;\,\langle P, s_1 \bullet s_3\rangle)}$$

If2

$$\frac{P; H; e \Downarrow v \qquad v = 0}{(H;\,\omega;\,t;\,\langle P, \mathsf{if}^p\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2; s_3\rangle) \xrightarrow{p} (H;\,\omega;\,t;\,\langle P, s_2 \bullet s_3\rangle)}$$

While

$$\frac{}{(H;\,\omega;\,t;\,\langle P, (\mathsf{while}^p\, e\, \mathsf{do}\, s_1); s_2\rangle) \xrightarrow{p} (H;\,\omega;\,t;\,\langle P, (\mathsf{if}\, e\, \mathsf{then}\, (s_1 \bullet \mathsf{while}\, e\, \mathsf{do}\, s_1; \mathsf{skip})\, \mathsf{else}\, \mathsf{skip}); s_2\rangle)}$$

Figure 3. Tree and statement semantics $(H;\,\omega;\,t;\,T) \xrightarrow{p} (H';\,\omega;\,t';\,T')$

is defined in Figure 4. Evaluation of expressions is standard, with the exception of memory read $!r$, which requires that memory location r is held at place P, the current place of the activity performing the read.

Rule SKIP2 handles the instruction skip—it is a no-op. Rule WRITE executes write instruction $r :=^p e$ by evaluating expression e to a value v and updating the heap to map location r to v. Note that location r must be stored at the place at which the activity is executing: $Place(r) = P$.

Rule LET executes let instruction $\mathsf{let}\, x = e \,\mathsf{in}\, s$ by evaluating expression e to value v, and substituting uses of variable x in s with v using capture-avoiding substitution $s\{v/x\}$. The rule uses the operation $\bullet$ to "stitch together" two statements into a single statement. This operation is defined recursively as follows.

$$(i; s_1) \bullet s_2 = i; (s_1 \bullet s_2)$$
$$\mathsf{skip} \bullet s_2 = s_2$$

Instruction $\mathsf{async}\, s_1$ creates a new activity to execute s_1, and the current activity continues with the next statement. Thus, rule ASYNC executes the activity $\langle P, \mathsf{async}\, s_1; s_2\rangle$ by reducing it to the tree $\langle P, s_1\rangle \,||\, \langle P, s_2\rangle$.

Statement $\mathsf{finish}\, s_1; s_2$ executes s_1, and waits until all activities spawned by s_1 have terminated before executing s_2. Rule FINISH transforms activity $\langle P, \mathsf{finish}\, s_1; s_2\rangle$ to the tree $\langle P, s_1\rangle \rhd \langle P, s_2\rangle$.

Statement $\mathsf{at}\, P\, s_1; s_2$ executes statement s_1 at place P, and then executes s_2 at the original place. Rule AT thus transforms activity $\langle P', \mathsf{at}\, P\, s_1; s_2\rangle$ to an activity at place P: $\langle P, s_1 \bullet (\mathsf{backat}\, P'; s_2)\rangle$. We insert the instruction $\mathsf{backat}\, P'$ to let us know both that execution of s_2 will be at place P', and that the movement of the activity to P' is the result of returning from a previous at instruction. Rule BACKAT for statement $\mathsf{backat}\, P'$ simply changes the place of the activity back to place P'.

Rule OUTPUT evaluates output instruction $\mathsf{output}\, e$ by evaluating e to value v, and appending event $o(v, P)$ to the program's trace, where P is the current place of the activity. Similarly, rule INPUT evaluates input instruction $\mathsf{input}\, r$ by inputting value v from P's communication channel, updating the heap to map location r to v, and appending event $i(v, P)$ to the program's trace. The value to input is determined by input strategy ω, and is equal to $\omega(P, t\restriction_P)$, where P is the current place of the activity, and $t\restriction_P$ is the program's trace so far restricted to events occurring at place P.

Rules IF1 and IF2 handle the conditional instruction $\mathsf{if}\, e\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2$ by reducing it to s_1 if e evaluates to a non-zero value, and reducing it to s_2 otherwise. Rule WHILE handles a $\mathsf{while}\, e\, \mathsf{do}\, s_1$ instruction by unrolling it into a conditional instruction.

2.5 Program execution

A program is an activity $\langle P, s\rangle$, that is, a statement s that is intended to start execution at place P. Program execution depends on an input strategy ω and a refiner R. The initial configuration of a program is $(H_{init}; \omega; R; \epsilon; \langle P, s\rangle)$, where H_{init} is a distinguished heap and ϵ is the empty trace.

For program $\langle P, s\rangle$, input strategy ω, and refiner R, we write

$$(\langle P, s\rangle, \omega, R)\ \mathsf{emits}\ t$$

to indicate that program execution can produce trace t. That is, there is some heap H', refiner R' and tree T' such that

$$(H_{init}; \omega; R; \epsilon; \langle P, s\rangle) \to^* (H'; \omega; R'; t; T')$$

where $\to^*$ is the reflexive transitive closure of the small-step relation $\to$.

3. Security

We are interested in enforcing strong information security in concurrent programs. Towards that end, in this section, we define a noninterference-based [12] definition of security for FSX10, and present a type system that enforces security while allowing many useful and highly concurrent programs.

3.1 Defining security

Intuitively, we want to ensure that a consumer of low-security information from a FSX10 program does not learn anything about high-security information. In our setting, consumers of low-security information are entities that can observe the communication channel of low-security places, and the high-security information that needs to be protected are the values input at high-security places.

We assume that there is a set of security levels $\mathcal{L}$ with a partial order $\sqsubseteq$ that describes relative restrictiveness of the security levels. We further assume that every place P is associated with a security level, denoted $\mathcal{L}(P)$. Intuitively, place P will be allowed to store and handle only data of security level $\mathcal{L}(P)$ and lower, and to send values to, and invoke code on, only places P' such that $\mathcal{L}(P) \sqsubseteq \mathcal{L}(P')$.

For a given security level $\ell \in \mathcal{L}$, a *low-security place* is any place P such that the level of the place is less than or equal to ℓ, that is, $\mathcal{L}(P) \sqsubseteq \ell$. All other places are *high-security places*, i.e., P is a high-security place if $\mathcal{L}(P) \not\sqsubseteq \ell$.

We define a semantic security condition based on *attacker knowledge* [2]. An attacker observes the communication channels of low-security places. The knowledge of an attacker is the set of input strategies that are consistent with the attacker's observations: the smaller the set, the more accurate the attacker's knowledge. The semantic security condition will require that at all times, the attacker's knowledge includes all possible input strategies for high-security places. That is, all possible input strategies for high-security places are consistent with the attacker's observations. For ease of presentation, we will use a slightly weaker semantic security condition, a *progress-insensitive* condition [3] that also allows the attacker to learn not only the input strategies for low security places, but also whether low-security output is produced.

Trace equivalence Given an attacker with security level $\ell \in \mathcal{L}$ (i.e., who can observe communication channels of places P such that $\mathcal{L}(P) \sqsubseteq \ell$), two executions of a program look the same to the attacker if the trace of inputs and outputs at low-security places are the same in both executions. We define this formally via ℓ-equivalence of traces.

Definition 1 (ℓ-equivalence of traces)**.** Let $\ell \in \mathcal{L}$. Traces t_0 and t_1 are *ℓ-equivalent*, written $t_0 \sim_\ell t_1$, if $t_0\lceil_\ell = t_1\lceil_\ell$, where

$$\epsilon\lceil_\ell = \epsilon$$

$$(t \cdot \alpha)\lceil_\ell = \begin{cases} (t\lceil_\ell) \cdot \alpha & \text{if } \mathcal{L}(\mathit{Place}(\alpha)) \sqsubseteq \ell \\ t\lceil_\ell & \text{otherwise} \end{cases}$$

Attacker knowledge For a given execution of a program, starting from program configuration $(H_{init}; \omega; R; \epsilon; \langle P, s\rangle)$, that produces trace t, the knowledge of an attacker with security level ℓ, written $k(\langle P, s\rangle, R, t, \ell)$, is the set of input strategies that could have produced a trace that is equivalent to what the attacker observed.

Definition 2 (Attacker knowledge)**.** For any $\ell \in \mathcal{L}$, program $\langle P, s\rangle$, trace t, and refiner R, the attacker's knowledge is:

$$k(\langle P, s\rangle, R, t, \ell) = \{\omega \mid \exists t'. \, (\langle P, s\rangle, \omega, R) \text{ emits } t' \wedge t \sim_\ell t'\}$$

We define what information an attacker with security level ℓ is permitted to learn about input strategies by defining ℓ-equivalence of input strategies. Intuitively, if two strategies are ℓ-equivalent, then they provide the exact same inputs for all low-security places, and an attacker with security level ℓ should not be able to distinguish them.

Definition 3 (ℓ-equivalence of input strategies)**.** Let $\ell \in \mathcal{L}$. Input strategies ω_0 and ω_1 are *ℓ-equivalent*, written $\omega_0 \sim_\ell \omega_1$, if for all places P such that $\mathcal{L}(P) \sqsubseteq \ell$, and for all traces t, we have $\omega_0(P, t\lceil_P) = \omega_1(P, t\lceil_P)$.

Relation $\sim_\ell$ is an equivalence relation, and we write $[\omega]_\ell$ for the equivalence class of ω under the relation $\sim_\ell$.

Given a program configuration $(H_{init}; \omega; R; \epsilon; \langle P, s\rangle)$ that produces trace t, *progress knowledge* [3] is the set of input strategies that could have produced a trace that is ℓ-equivalent to t, *and* could produce at least one more observable event. We will use progress knowledge as a lower bound on the allowed knowledge of an attacker. That is, we will explicitly allow the attacker to learn whether a program will produce another observable event. This means that the attacker may be permitted to learn the termination behavior of statements that depend on high-security information.

Definition 4 (Progress knowledge)**.** For any $\ell \in \mathcal{L}$, program $\langle P, s\rangle$, trace t, and refiner R, progress knowledge is:

$$k^+(\langle P, s\rangle, R, t, \ell) = \{\omega \mid \exists t', \alpha. \, (\langle P, s\rangle, \omega, R) \text{ emits } (t' \cdot \alpha) \wedge t \sim_\ell t' \wedge \mathcal{L}(\mathit{Place}(\alpha)) \sqsubseteq \ell\}$$

Our security condition requires that, for all attackers, and all executions, for each event the attacker can observe, the attacker learns no more than the input strategy for low-security places, and the fact that another event was produced.

Definition 5 (Security)**.** Program $\langle P, s\rangle$ is secure if for all $\ell \in \mathcal{L}$, traces $t \cdot \alpha$, refiners R, and input strategies ω such that

$$(\langle P, s\rangle, \omega, R) \text{ emits } (t \cdot \alpha)$$

we have

$$k(\langle P, s\rangle, R, t \cdot \alpha, \ell) \quad \supseteq \quad [\omega]_\ell \cap k^+(\langle P, s\rangle, R, t, \ell).$$

Recall that the attacker's knowledge is set of input strategies that are consistent with the attacker's observations: a smaller set means more precise knowledge. Security requires that there are lower bounds to the precision of the attacker's knowledge. That is, there is information that the attacker is not permitted to learn. Thus, security requires that attacker's knowledge is a superset of the knowledge it is permitted to learn.

According to this definition of security, Program 2 from the Introduction is insecure (assuming that memory location hi is initialized from an input from place High), since there exists a refiner and a strategy that will produce a trace that allows an observer of low-security outputs to learn something about the high-security input strategy. Indeed, our definition of security rules out internal timing channels [47], in which the order of low-security events (here, input, output, and accesses to memory locations) depends upon high-security information. Program 3 does not exhibit an internal timing channel, and is secure.

This definition of security is *progress insensitive* [3], as it permits the attacker to learn that program execution makes progress, and produces another observable output. This definition can be strengthened in a straightforward way to a *progress sensitive* security condition. While the type system of Section 3.2 enforces progress insensitive security, it can be modified using standard techniques (to conservatively reason about termination of loops) to enforce progress sensitive security [28]. We refrain from doing so to simplify the presentation of the type system.

3.2 Enforcing security

We enforce security using a security type system. The type system ensures that each place P stores and handles only data input from places P' such that $\mathcal{L}(P') \sqsubseteq \mathcal{L}(P)$. However, as noted in the Introduction, it is possible for the scheduling of activities at place P to be influenced by input from a place P' such that $\mathcal{L}(P') \not\sqsubseteq \mathcal{L}(P)$. Our type system tracks and controls information flow through this covert channel through program point contexts.

A *program point context* Δ is a function from program points to security levels such that $\Delta(p)$ is an upper bound on the level of information that may influence the scheduling of program point p. More precisely, it is an upper bound on information that may affect the presence or absence of activities that may run concurrently with p at the same place.

Each program point is statically associated with a place, and we write $\mathit{Place}(p)$ for the place at which program point p will execute. Intuitively, since program point p is executed at place $\mathit{Place}(p)$, and $\mathit{Place}(p)$ handles data at security level $\mathcal{L}(\mathit{Place}(p))$, we would expect that $\Delta(p)$ is at least as restrictive as $\mathcal{L}(\mathit{Place}(p))$. Indeed, the type system ensures for all p that $\mathcal{L}(\mathit{Place}(p)) \sqsubseteq \Delta(p)$.

It is often the case that $\mathcal{L}(\mathit{Place}(p))$ is also an upper bound of $\Delta(p)$. That is, the scheduling of p does not depend on any high-security information. However, if p may happen immediately after a computation at a high-security place finishes (as with the output "pos" instruction in Program 2), or in parallel with another program point at the same place whose scheduling depends on high-security information, then it is possible that $\Delta(p) \not\sqsubseteq \mathcal{L}(\mathit{Place}(p))$. In that case, in order to ensure that the scheduling decision at place $\mathit{Place}(p)$ does not leak high-security information, we require *observational determinism* [48] at $\mathit{Place}(p)$ during the scheduling of p. That is, for each memory location stored at $\mathit{Place}(p)$, there are no data races on that location, and the order of input and output at $\mathit{Place}(p)$ is determined.

Finally, *variable context* Γ maps program variables to the place at which the variable was created. The type system uses the variable context to ensure that if variable x was declared at place P, then x is used only at places P' such that $\mathcal{L}(P) \sqsubseteq \mathcal{L}(P')$.

May-happen-in-parallel analysis The type system relies on the results of a may-happen-in-parallel analysis, such as the one presented by Lee and Palsberg for Featherweight X10 [22]. The async-finish parallelism of X10 is amenable to a precise may-happen-in-parallel analysis. We write $\mathsf{MHPP}(p)$ for the set of program points that may happen in parallel with program point p at the same place (i.e., at $\mathit{Place}(p)$).

Typing expressions Judgment $p; \Gamma; \Delta \vdash e$ indicates that expression e, occurring at program point p is well typed under variable context Γ and program point context Δ. Inference rules for this judgment are given in Figure 5. Constants v are always well typed, and the use of variable x is well typed if the level of the place at which x is defined ($\mathcal{L}(\Gamma(x))$) is less than or equal to the level of the place at which x is used ($\mathcal{L}(\mathit{Place}(p))$). Expression $e_1 \oplus e_2$ is well typed if both e_1 and e_2 are well typed.

$$\begin{aligned}
\mathit{noWrite}(r,p) &= \forall p' \in \mathsf{MHPP}(p).\ \text{instruction at}\\
&\qquad p' \text{ does not write to } r\\[1ex]
\mathit{noReadWrite}(r,p) &= \mathit{noWrite}(r,p)\ \wedge\\
&\qquad \forall p' \in \mathsf{MHPP}(p).\ \text{instruction at}\\
&\qquad p' \text{ does not read } r.\\[1ex]
\mathit{noIO}(p) &= \forall p' \in \mathsf{MHPP}(p).\ \text{instruction at } p'\\
&\qquad \text{does not perform input or output.}
\end{aligned}$$

Figure 6. Predicate definitions

There are two different rules for reading memory location r. The first rule, rule T-READNONDET, handles the case where the scheduling of the expression's execution at place $\mathit{Place}(p)$ is influenced by information at most at level $\mathcal{L}(\mathit{Place}(p))$. In that case, there are no restrictions on when the read may occur: it may occur concurrently with activities at the same place that write to the location since the resolution of the data race will not be a covert information channel. (The existence of a data race may, however, be undesirable in terms of program functionality.)

The second rule, rule T-READDET, applies when the scheduling of the expression may be influenced by information that is not allowed to flow to level $\mathcal{L}(\mathit{Place}(p))$. In that case, the read is required to be observationally deterministic: predicate $\mathit{noWrite}(r,p)$ must hold, implying that the read of memory location r at program point p must not execute concurrently with any statement that may write to r. Predicate $\mathit{noWrite}(r,p)$ is defined in Figure 6.

Typing statements Judgment $\Gamma; \Delta \vdash s : \ell$ indicates that statement s is well typed in variable context Γ and program point context Δ, and that security level ℓ is an upper bound on the security level of information that may influence the scheduling of the last program point of s. Inference rules for the judgment are given in Figure 7.

Every inference rule for a statement s^p includes the premise $\forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p)$. Intuitively, the set $\mathsf{MHPA}(p)$ is the set of program points that may influence the presence or absence of activities running in parallel with p at the same place. Assuming that $\mathit{Place}(p) = P$, $\mathsf{MHPA}(p)$ contains the program points of backat P instructions that may happen in parallel with p, and the set of program points immediately following an $\mathsf{at}^{p'}\ P'\ s'$ instruction, where $\mathit{Place}(p') = P$ and p' may happen in parallel with p. The set $\mathsf{MHPA}(p)$ is a subset of the program points that may happen in parallel with p, and can easily be computed from the results of a may-happen-in-parallel analysis. Given this definition, the premise above requires that $\Delta(p)$, the upper bound on the scheduling of s, is at least as restrictive as the scheduling of any program point that may influence the presence or absence of activities running in parallel with p at the same place.

T-CONST
$$\frac{}{p;\Gamma;\Delta \vdash v}$$

T-VAR
$$\frac{\mathcal{L}(\Gamma(x)) \sqsubseteq \mathcal{L}(\mathit{Place}(p))}{p;\Gamma;\Delta \vdash x}$$

T-OP
$$\frac{\begin{array}{c} p;\Gamma;\Delta \vdash e_1 \\ p;\Gamma;\Delta \vdash e_2 \end{array}}{p;\Gamma;\Delta \vdash e_1 \oplus e_2}$$

T-READNONDET
$$\frac{\Delta(p) \sqsubseteq \mathcal{L}(\mathit{Place}(p))}{p;\Gamma;\Delta \vdash !r}$$

T-READDET
$$\frac{\begin{array}{c} \Delta(p) \not\sqsubseteq \mathcal{L}(\mathit{Place}(p)) \\ \mathit{noWrite}(r,p) \end{array}}{p;\Gamma;\Delta \vdash !r}$$

Figure 5. Expression typing judgment $p;\Gamma;\Delta \vdash e$

T-SKIP1
$$\frac{\forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p)}{\Gamma;\Delta \vdash \mathsf{skip}^p : \Delta(p)}$$

T-SKIP2
$$\frac{\begin{array}{c} \Gamma;\Delta \vdash s : \ell \qquad \Delta(p) \sqsubseteq \Delta(p_1) \\ \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{skip}^p; s^{p_1} : \ell}$$

T-LET
$$\frac{\begin{array}{c} p;\Gamma;\Delta \vdash e \qquad \Gamma[x \mapsto \mathit{Place}(p)];\Delta \vdash s_1 : \ell_1 \\ \Gamma;\Delta \vdash s_2 : \ell_2 \qquad \Delta(p) \sqsubseteq \Delta(p_1) \qquad \ell_1 \sqsubseteq \Delta(p_2) \\ \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{let}^p\ x = e\ \mathsf{in}\ s_1^{p_1}; s_2^{p_2} : \ell_2}$$

T-ASYNC
$$\frac{\begin{array}{c} \Gamma;\Delta \vdash s_1 : \ell_1 \qquad \Gamma;\Delta \vdash s_2 : \ell_2 \qquad \Delta(p) \sqsubseteq \Delta(p_1) \\ \Delta(p) \sqsubseteq \Delta(p_2) \qquad \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{async}^p\ s_1^{p_1}; s_2^{p_2} : \ell_2}$$

T-FINISH
$$\frac{\begin{array}{c} \Gamma;\Delta \vdash s_1 : \ell_1 \quad \Gamma;\Delta \vdash s_2 : \ell_2 \quad \Delta(p) \sqsubseteq \Delta(p_1) \quad \mathcal{L}(\mathit{Place}(p_2)) \sqsubseteq \Delta(p_2) \\ \forall p' \in \mathsf{MHPP}(p).\ \Delta(p') \sqsubseteq \Delta(p_2) \qquad \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{finish}^p\ s_1^{p_1}; s_2^{p_2} : \ell_2}$$

T-AT
$$\frac{\begin{array}{c} \Gamma;\Delta \vdash s_1 : \ell_1 \qquad \Gamma;\Delta \vdash s_2 : \ell_2 \qquad \Delta(p) \sqsubseteq \mathcal{L}(P) \\ \mathcal{L}(P) \sqsubseteq \Delta(p_1) \qquad \ell_1 \sqsubseteq \Delta(p_2) \qquad \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{at}^p\ P\ s_1^{p_1}; s_2^{p_2} : \ell_2}$$

T-BACKAT
$$\frac{\begin{array}{c} \Gamma;\Delta \vdash s : \ell \qquad \Delta(p) \sqsubseteq \Delta(p_1) \\ \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{backat}^p\ P; s^{p_1} : \ell}$$

T-WRITENONDET
$$\frac{\begin{array}{c} \Delta(p) \sqsubseteq \mathcal{L}(\mathit{Place}(p)) \\ \Gamma;\Delta \vdash s : \ell \qquad p;\Gamma;\Delta \vdash e \qquad \Delta(p) \sqsubseteq \Delta(p_1) \\ \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash r :=^p e; s^{p_1} : \ell}$$

T-WRITEDET
$$\frac{\begin{array}{c} \Delta(p) \not\sqsubseteq \mathcal{L}(\mathit{Place}(p)) \qquad \mathit{noReadWrite}(r,p) \\ \Gamma;\Delta \vdash s : \ell \qquad p;\Gamma;\Delta \vdash e \qquad \Delta(p) \sqsubseteq \Delta(p_1) \\ \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash r :=^p e; s^{p_1} : \ell}$$

T-OUTPUTNONDET
$$\frac{\begin{array}{c} \Delta(p) \sqsubseteq \mathcal{L}(\mathit{Place}(p)) \\ \Gamma;\Delta \vdash s : \ell \qquad p;\Gamma;\Delta \vdash e \qquad \Delta(p) \sqsubseteq \Delta(p_1) \\ \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{output}^p\ e; s^{p_1} : \ell}$$

T-OUTPUTDET
$$\frac{\begin{array}{c} \Delta(p) \not\sqsubseteq \mathcal{L}(\mathit{Place}(p)) \qquad \mathit{noIO}(p) \\ \Gamma;\Delta \vdash s : \ell \qquad p;\Gamma;\Delta \vdash e \qquad \Delta(p) \sqsubseteq \Delta(p_1) \\ \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{output}^p\ e; s^{p_1} : \ell}$$

T-INPUTNONDET
$$\frac{\begin{array}{c} \Delta(p) \sqsubseteq \mathcal{L}(\mathit{Place}(p)) \qquad \Gamma;\Delta \vdash s : \ell \\ \Delta(p) \sqsubseteq \Delta(p_1) \qquad \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{input}^p\ r; s^{p_1} : \ell}$$

T-INPUTDET
$$\frac{\begin{array}{c} \Delta(p) \not\sqsubseteq \mathcal{L}(\mathit{Place}(p)) \qquad \mathit{noReadWrite}(r,p) \qquad \mathit{noIO}(p) \\ \Gamma;\Delta \vdash s : \ell \qquad \Delta(p) \sqsubseteq \Delta(p_1) \qquad \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{input}^p\ r; s^{p_1} : \ell}$$

T-IF
$$\frac{\begin{array}{c} \Gamma;\Delta \vdash s_1 : \ell_1 \qquad \Gamma;\Delta \vdash s_2 : \ell_2 \qquad \Gamma;\Delta \vdash s_3 : \ell_3 \\ p;\Gamma;\Delta \vdash e \qquad \Delta(p) \sqsubseteq \Delta(p_1) \qquad \Delta(p) \sqsubseteq \Delta(p_2) \\ \ell_1 \sqsubseteq \Delta(p_3) \qquad \ell_2 \sqsubseteq \Delta(p_3) \qquad \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{if}^p\ e\ \mathsf{then}\ s_1^{p_1}\ \mathsf{else}\ s_2^{p_2}; s_3^{p_3} : \ell_3}$$

T-WHILE
$$\frac{\begin{array}{c} \Gamma;\Delta \vdash s_1 : \ell_1 \qquad \Gamma;\Delta \vdash s_2 : \ell_2 \qquad p_1;\Gamma;\Delta \vdash e \\ \Delta(p) \sqsubseteq \Delta(p_1) \qquad \ell_1 \sqsubseteq \Delta(p) \qquad \ell_1 \sqsubseteq \Delta(p_2) \\ \forall p' \in \mathsf{MHPA}(p).\ \Delta(p') \sqsubseteq \Delta(p) \end{array}}{\Gamma;\Delta \vdash \mathsf{while}^p\ e\ \mathsf{do}\ s_1^{p_1}; s_2^{p_2} : \ell_2}$$

Figure 7. Statement typing judgment $\Gamma;\Delta \vdash s : \ell$

Also, almost all inference rules for statements ensure that if program point p executes after p' (for example, because they are in sequence), then $\Delta(p') \sqsubseteq \Delta(p)$. The intuition here is that if information at level $\Delta(p')$ may influence the scheduling of p', and p follows in sequence after p', then information at level $\Delta(p')$ may influence the scheduling of p. For example, the typing rule for $\mathsf{if}^p\ e\ \mathsf{then}\ s_1^{p_1}\ \mathsf{else}\ s_2^{p_2}; s_3^{p_3}$ requires that $\Delta(p) \sqsubseteq \Delta(p_1)$ and $\Delta(p) \sqsubseteq \Delta(p_2)$, since the execution of s_1 and s_2 will occur only after the evaluation of the conditional guard. Similarly, since the execution of s_3 will follow the execution of either s_1 or s_2, the rule requires that $\ell_1 \sqsubseteq \Delta(p_3)$ and $\ell_2 \sqsubseteq \Delta(p_3)$, where ℓ_1 and ℓ_2 are upper bounds of the scheduling of the last program points of s_1 and s_2 respectively.

We discuss only the inference rules that have premises in addition to those common to all rules.

Statement $\mathsf{let}^p\ x = e\ \mathsf{in}\ s_1^{p_1}; s_2^{p_2}$ declares a variable x, and allows x to be used in the scope of statement s_1. Rule T-LET thus allows s_1 to be typed with a variable context that maps variable x to the place at which it was defined: $\mathit{Place}(p)$.

Statement $\mathsf{finish}^p\ s_1^{p_1}; s_2^{p_2}$ executes statement s_1, and waits until all activities spawned by s_1 have finished before executing s_2. Rule T-FINISH requires that $\Delta(p) \sqsubseteq \Delta(p_1)$ (since p_1 is executed after p) but notably does not require either $\Delta(p) \sqsubseteq \Delta(p_2)$ or $\ell_1 \sqsubseteq \Delta(p_2)$, despite the fact that p_2 is executed after p and p_1. The intuition is that because the scheduling behavior at place $P = \mathit{Place}(p)$ depends only on the current activities at P, by the time that p_2 is scheduled, program points p and p_1 (and all activities spawned by s_1) have finished execution, and do not influence the scheduling of p_2. In Program 3 in the Introduction, this reasoning is what permits us to conclude that the scheduling of output ''B'' and output ''C'' do not depend on high-security computation.

```
at Low {
    async {
        // Activity 1
        mediumComputation(); output ''nonpos'';
    }
    finish {
        // Activity 2
        at High {
            if (hi > 0) longComputation();
        }
    }
    // Activity 3
    output ''pos'';
}
```

Program 4.

However, it may be possible that scheduling of activities spawned by s_1 indirectly influences the scheduling of p_2. Consider Program 4, which contains a finish $s_1; s_2$ statement where $s_2 =$ output ''pos'', and s_1 invokes computation at high-security place High. There is an additional activity that executes concurrently with the finish statement: mediumComputation(); output ''nonpos''. The scheduling of this activity relative to s_2 will depend on the high-security computation. Indeed, this program is equivalent to Program 2, and both are insecure. Thus, typing rule T-FINISH requires that $\Delta(p_2)$ is at least as restrictive as $\Delta(p')$ for any program point p' that may execute in parallel with p at the same place. This ensures that insecure Program 4, and others like it, are rejected by the type system.

Statement $\mathsf{at}^p\ P\ s_1^{p_1}; s_2^{p_2}$ executes s_1 at place P, and then executes s_2 back at place $\mathit{Place}(p)$. Rule T-AT requires that the upper bound on the scheduling of the at instruction is permitted to flow to the level of place P ($\Delta(p) \sqsubseteq \mathcal{L}(P)$). Thus the type system restricts the creation of an activity at place P to reveal only information that is allowed to flow to level $\mathcal{L}(P)$. Also, because statement $s_1^{p_1}$ is executing at place P, information at level $\mathcal{L}(P)$ will influence the scheduling of p_1: $\mathcal{L}(P) \sqsubseteq \Delta(p_1)$. Finally, because statement $s_2^{p_2}$ is executed only after s_1, the scheduling of p_2 depends on when the last statement of s_1 is scheduled: $\ell_1 \sqsubseteq \Delta(p_2)$ where ℓ_1 is an upper bound on the scheduling of the last program point of s_1.

Similar to the typing rules for reading memory locations, there are two rules for writing memory locations: T-WRITENONDET and T-WRITEDET. As with the rules for reading memory, the first is for the case where the scheduling of the write is not influenced by high-security information, and there are thus no restrictions on when the write may occur. Rule T-WRITEDET applies when the scheduling of the write may be influenced by high-security information, and requires observational determinism via the predicate $\mathit{noReadWrite}(r, p)$, defined in Figure 6, which ensures that no reads or writes to the same memory location may happen in parallel.

The rules for input and output are similar to the rules for reading and writing memory locations: if the scheduling of input or output may depend on high-security information, the input or output must be observationally deterministic, which is achieved for output by requiring that there is no other input or output at that place that may happen in parallel (see predicate $\mathit{noIO}(p)$, defined in Figure 6). Since an input instruction writes to a memory location r, rule T-INPUTDET requires both that no input or output may happen at the place in parallel, and that no reads or writes to r may happen in parallel.

Typing trees Judgment $\Gamma; \Delta \vdash T$ means that tree T is well typed in variable context Γ and program point context Δ. Inference rules for the judgment are given in Figure 8. The rules require that all activities in the tree are well typed. Also, the rule for tree $T \triangleright \langle P, s^p \rangle$, T-JOIN, requires that $\Delta(p)$ is at least as restrictive as $\Delta(p')$ for any program point p' that may execute in parallel with p at place P, for similar reasons to the typing rule for finish statements, T-FINISH.

$$\textsc{T-Activity}\quad \frac{\Gamma;\Delta \vdash s : \ell \quad \mathcal{L}(P) \sqsubseteq \Delta(p)}{\Gamma;\Delta \vdash \langle P, s^p \rangle} \qquad \textsc{T-Para}\quad \frac{\Gamma;\Delta \vdash T_1 \quad \Gamma;\Delta \vdash T_2}{\Gamma;\Delta \vdash T_1 \| T_2}$$

$$\textsc{T-Join}\quad \frac{\Gamma;\Delta \vdash T \qquad \Gamma;\Delta \vdash \langle P, s^p \rangle \qquad \forall p' \in \mathsf{MHPP}(p).\ \Delta(p') \sqsubseteq \Delta(p)}{\Gamma;\Delta \vdash T \triangleright \langle P, s^p \rangle} \qquad \textsc{T-Done}\quad \frac{}{\Gamma;\Delta \vdash \checkmark}$$

Figure 8. Tree typing judgment $\Gamma; \Delta \vdash T$

Soundness of type system The type system enforces security. That is, if a program is well typed, then it is secure.

Theorem 1. If $\langle P, s\rangle$ is a program such that $\Gamma; \Delta \vdash \langle P, s\rangle$ for some variable context Γ and program point context Δ, then $\langle P, s\rangle$ is secure according to Definition 5.

We present a brief sketch of the proof here. A more detailed proof appears in Appendix A.

Outline of Proof. The proof uses a technique similar to that of Terauchi [44]. We first introduce the concept of an *erased configuration*. A configuration m erases to a configuration m' at security level ℓ if m', when executed, performs no computation at places with security level higher than ℓ but m and m' otherwise agree. Erased programs are defined similarly, with erased configurations containing erased programs.

Suppose we have a well-typed program $\langle P, s\rangle$, some security level ℓ, and two ℓ-equivalent input strategies ω_1 and ω_2. First, we erase the program $\langle P, s\rangle$ to program $\langle P, s'\rangle$ at level ℓ and consider side-by-side executions of these two programs with the same input strategy. Suppose the original program with input strategy ω_1 produces trace t_1. Then the erased program with input strategy ω_1 can produce a trace t_1' that is ℓ-equivalent. Similarly, if the original program with input strategy ω_2 produces trace t_2, then the erased program with input strategy ω_2 can produce a trace t_2' that is ℓ-equivalent.

Second, we consider the executions of the erased program with strategy ω_1 and strategy ω_2 that produced traces t_1' and t_2' respectively. Since the erased program performs no computation at high-security places, either t_1' is a prefix of t_2', or vice versa. Combining this with the previous result, if $(\langle P, s\rangle, \omega_1, R)$ emits t_1 and $(\langle P, s\rangle, \omega_2, R)$ emits t_2, then either the low-security events of t_1 are a prefix of the low-security events of t_2, or vice versa.

Knowledge-based security can then be shown as follows. Let ω be an input strategy, R a refiner, and ℓ a security level. Suppose that $(\langle P, s\rangle, \omega, R)$ emits $t \cdot \alpha$. Let ω' be another input strategy such that $\omega \sim_\ell \omega'$ and $(\langle P, s\rangle, \omega', R)$ emits $t' \cdot \alpha'$ such that $t \sim_\ell t'$ and $\mathcal{L}(Place(\alpha')) \sqsubseteq \ell$. The above result implies that either $\alpha = \alpha'$ or $\mathcal{L}(Place(\alpha)) \not\sqsubseteq \ell$. In either case, $t \cdot \alpha \sim_\ell t' \cdot \alpha'$, and so the inclusion required by Definition 5 is proven. □

4. SX10 prototype implementation

We have extended the principles of the security analysis of Section 3 to handle many of the language features of X10. The resulting language, called SX10, is a subset of X10. We have implemented a prototype compiler for SX10 by extending the open-source X10 compiler (version 2.1.2), which is implemented using the Polyglot extensible compiler framework [29], and is included in the X10 distribution. Our extension comprises approximately 8,500 lines of non-comment non-blank lines of Java code.

We do not modify the X10 run-time system: SX10 programs run using the standard X10 run-time system. We thus do not provide a performance comparison of SX10 with X10 or with other secure concurrent systems. Such a performance comparison is not directly useful, as it would evaluate the efficiency of the X10 runtime, not our enforcement technique, which is entirely static.

In this section, we describe how we extend the analysis to handle additional language features of X10 and present some example SX10 programs.

May-happen-in-parallel analysis We have implemented the may-happen-in-parallel (MHP) analysis of Lee and Palsberg [22] for SX10, which is a straightforward exercise. However, for additional precision in our security analysis, we implemented a *place-sensitive* MHP analysis. In our calculus FSX10, for every program point it is possible to statically determine which place the program point would execute on. In SX10, however, code for a given class may be executed at more than one place, since objects of the same class may reside at different places. Thus, if an activity at place P is executing code from program point p, our place-sensitive MHP analysis conservatively approximates the set $\mathsf{MHP}(p, P)$ such that if $(p', P') \in \mathsf{MHP}(p, P)$ then an activity at place P' may concurrently be executing code from program point p'.

Places We assume that all places are statically known, and that a security level is associated with each place. A configuration file specifies the set of security levels $\mathcal{L}$, the ordering $\sqsubseteq$ over the levels, and maps places to levels. Our prototype implementation does not currently support first-class places. If places are computed dynamically, then the choice of the place at which to execute a computation could be a covert channel, and would thus require the security analysis to track and control information flow through this channel. In this respect, first-class places are similar to first-class security levels (e.g., [13, 49]), and the security analysis could be extended to handle first-class places using similar techniques, such as dependent type systems.

As in FSX10, we restrict at statements to allow place P to invoke code on place P' only if $\mathcal{L}(P) \sqsubseteq \mathcal{L}(P')$.

In addition to at statements, X10 has at expressions: at P e evaluates expression e at place P. We allow at expressions, but only from place P to place P' where

$\mathcal{L}(P) = \mathcal{L}(P')$. If the security level of the places differed, then either data would be sent from a high-security place to a low-security place, or a high-security place would invoke code on a low-security place. Either way, a potentially dangerous information flow occurs, and must be ruled out.

Concurrency mechanisms The X10 async and finish statements are restricted similarly to their counterparts in FSX10. X10 provides additional synchronization mechanisms, including clocks (a form of synchronization barrier), futures, and atomic blocks. Our prototype implementation does not currently support these additional mechanisms. However, they can be incorporated in a straightforward manner by extending the MHP analysis to reason about them. Once the MHP analysis supports these constructs, our security analysis can be extended to add constraints similar to those for async and finish statements.

Objects Fields of objects can be mutable locations, and we enforce restrictions similar to those of other memory locations: we require determinism on accesses when scheduling may be influenced by high-security information. If an object is sent in a message from one place to another, the X10 runtime will create a copy of the object, thus ensuring that if an activity at a place attempts to update a field of an object, the memory location is local to the place. When objects are copied to send to another place, we impose restrictions similar to the use of variables: a copy of an object created at place P may be sent to place P' only if $\mathcal{L}(P) \sqsubseteq \mathcal{L}(P')$.

Control-flow constructs X10 has much richer control-flow constructs than the calculus FSX10. We support local-control-flow constructs, such as for loops and switch statements. We support dynamic dispatch of methods, using class information to conservatively over-approximate the set of possible callees at a method call site. We do not currently support exceptions, although they can be incorporated by extending the MHP analysis. Note that exceptions interact in an interesting way with the concurrency mechanisms, due to X10's *rooted exception model* [40].

Input and output The security analysis for SX10 restricts input and output from the system to enforce strong information security guarantees. We currently require methods that perform communication with the external environment to be explicitly annotated as such, but it is straightforward to infer where such methods are used, for example, detecting method calls to objects of classes x10.io.Printer, x10.io.Reader, etc. (Fields x10.io.Console.OUT and x10.io.Console.IN are instances of Printer and Reader, respectively.)

Arrays Our implementation supports local arrays, since these are simply objects of the class x10.array.Array[T], and elements of the array are stored at a single place. We do not currently support distributed arrays, which store elements over multiple places. Adding support for distributed arrays would require support for first-class places.

X10 runtime scheduler The X10 runtime scheduler uses a work-stealing algorithm to schedule activities within a place. This requires that threads maintain a double-ended queue of pending activities, and idle threads may steal activities from busy threads. Because the state of the queues may be influenced by which activities were or were not running at the place in the past, such work-stealing algorithms cannot be represented in our scheduling model in FSX10, which requires that scheduling functions do not depend on the history of computation at a place.

The type system relies on this requirement only in the rule for a finish $s_1; s_2$ statement, which allows the program point context of the first program point of s_2 to be lower than program point context of the last program point of s_1 when there are no other activities running at the place. However, in that situation there are no other activities to schedule other than s_1 and activities spawned by s_1: activity s_2 will not start execution until it is the only activity at the place. In that case, the state of the work queues for threads will be independent of the history of the computation up to that point. Thus, we expect the security guarantee to transfer to the actual scheduler used by the X10 runtime.

It is future work to extend the model of schedulers in FSX10 to include such work-stealing schedulers.

Improvements to Analysis In implementing the SX10 compiler, we add an optimization that allows the type system to be more permissive without compromising security. If statement at P s is executed at place P' and no statement at P' is dependent on the termination of s, then the scheduling of activities at P' is independent of when s terminates, and we do not need to increase the program point context of statements that may happen in parallel. This corresponds to having a more precise definition of the set $\mathsf{MHPA}(p)$.

4.1 Example programs

Distributed Machine Learning Consider a music recommendation service, such as Pandora. Here, a large database of music information exists: the Music Genome Project. The service would like to process this data—perhaps running machine-learning algorithms on it—and then combine it with data from individual users to produce recommendations for users. We assume the database of music information is public, but the personal data from users is secret, and should not influence the results observed by other users.

We assume that the processing of the public data can be performed in parallel. We will process this data at a number of places pub0 through pubn, all with the same low security level L. Results from the processing will be sent back to the coordinator place, and collated into a value called results. We use the results of the processing of public data to compute recommendations for each user. We assume that each user U_i has its own place $\mathsf{P}i$, with a unique security level H_i such that $L \sqsubseteq H_i$.

A sketch of the code for this system appears in Program 5. Public data is supplied in the array pubData and we assume that place P_i already holds the data for user U_i.

```
at coordinator {
    finish {
        async at pub0 {
            val res = processPublicData(pubData(0));
            at coordinator { addResult(res); }
        }
        ...
        async at pubn {
            val res = processPublicData(pubData(n));
            at coordinator { addResult(res); }
        }
    }

    async { at P0 { processPrivateData(results); } }
    ...
    async { at Pn { processPrivateData(results); } }
}
```

Program 5. A distributed machine learning system

Note that no additional synchronization is required to make this program secure, and the secure program is in fact allowed to be highly concurrent. The translation of this program from X10 to SX10 does, however, require significant code duplication due to the lack of support for distributed arrays and first-class places. Natural X10 programming style would use a loop over places to execute a block of code at each place, rather than duplicate the code as we do here.

Online Shopping Following Tsai et al. [46], we consider a server running a shopping website. We assume that the server is concerned with keeping credit card data secure. Our example program models a multithreaded server accepting input from two web forms. On the first form, a user enters the item number they wish to purchase. This form is submitted along with the user's unique customer ID, which persists through the user's session. When this form is processed, the user's order is both saved on the server and output to a log for inventory purposes. The user is then presented with the next form, which requests his or her credit card number. This form is submitted with the same customer ID, and the credit card number is sent to an external service for processing.

This example contains two security levels. The customer ID and order are considered low-security, and the log is considered low-security output. The customer's credit card number is high-security, and so the action of exporting it should occur at a high place. We would like to ensure that no data from either the second form or the credit card processing service can leak to the log. The code for handling one customer's purchase is shown in Program 6.

Simpson's Rule Our final example program demonstrates that in programs, or sections of programs, in which all data is at the same security level, our analysis requires very few changes to the code for compilation. Thus, when a program operates on homogenous data, our security analysis does not significantly impact usability. The code for this example was taken from the Simpson's Rule example available on the X10 website[1]. The original program consists of approximately 200 non-blank non-comment lines of X10 code. Converting this program to SX10 required modifying seven lines of code, most of them trivially, and adding ten. The changes were as follows:

- Five statements producing console output were annotated as required by SX10.
- The original program uses all available places. Since SX10 requires static places, Place.MAX_PLACES, which in X10 is set to the number of places, was replaced with a new (arbitrary) constant, set to four for the purposes of this example. Identifiers representing these four places were declared.
- The code to start computation at each place was duplicated, since SX10 does not support loops over places. This required five additional lines of code.

Note that neither the number of lines modified nor the number added necessarily scales with the size of the code. Most required modifications were to input or output statements, and the number of lines of code added was proportional to the number of places used, not the size of the program.

Discussion of Example Programs The example programs demonstrate that it is possible to write realistic, highly concurrent programs in SX10. Note that the first two examples contain a high degree of nondeterminism. The order in which blocks of data are processed in Program 5 and the order in which entries are written to the log and credit card service in Program 6 are nondeterministic. This is secure because the resolution of this nondeterminism can in no way reveal high-security information. As will be discussed in Section 5, some previous security-type systems for noninterference in concurrent programs would rule out this nondeterminism and require additional synchronization and overhead.

The third example demonstrates that our analysis does not significantly prohibit the compilation and execution of programs that operate on a single security level.

The biggest restriction in SX10 is the lack of first-class places. As we gain more experience writing SX10 programs, we will identify and address further challenges to practical and secure concurrent programming.

5. Related work

This work seeks to provide strong language-based information security guarantees for concurrent programs. We discuss related work, focusing on recent work that controls information flow in concurrent settings.

Observational determinism Our security analysis ensures that if the scheduling of input, output, or memory accesses

[1] `http://x10-lang.org/`

```
async {
    at lowform {
        val custID = Int.parse(Console.IN.readLine().trim());
        // Input item number (low security)
        items(custID) = Int.parse(Console.IN.readLine().trim());
        at log Console.OUT.println(custID + "\t" + items(custID));
        at highform {
            // Input credit card number (high security)
            val card = Int.parse(Console.IN.readLine().trim());
            at cc Console.OUT.println(card + "\t" + costs(items(custID)));
        }
    }
}
```

Program 6. An online shopping cart

may leak sensitive information, then the order of such instructions must be deterministic. This approach is inspired by Zdancewic and Myers [48], who propose (following McLean [27] and Roscoe [32]) that there should be no nondeterminism (including thread scheduling) observable by a low-security observer. They present a semantic security condition that, for each observable memory location, requires determinism of the sequence of updates to that location. Huisman et al. [19] point out that this semantic security condition may reveal more information than intended, and propose that the sequence of updates to *all* observable memory locations should be deterministic. Terauchi [44] presents a type system that enforces such a semantic security condition in a shared-memory setting using fractional capabilities.

Mantel et al. [26] present semantic conditions that allow composition of concurrent programs. The semantic conditions use assume-guarantee reasoning to ensure that the composed program is free of data races, and thus is observationally deterministic.

Requiring observational determinism throughout a program is, however, overly restrictive. O'Neill et al. [30] note that low-observable nondeterminism is acceptable so long as its resolution depends only on low-observable information. We thus allow nondeterminism in the scheduling of activities at a place, provided that the resolution of the nondeterminism cannot leak sensitive information. Our model assumes that scheduling of activities at a place depends only on the activities at that place, and our security analysis exploits this assumption to allow non-determinism where possible.

Recent work on deterministic concurrency (e.g., [7, 45]) highlights functional benefits of determinism, and also allows some nondeterminism when it is safe to do so [8].

Scheduler independence Sabelfeld and Sands [37, 39] argue that the definition of security in multi-threaded programs should be *scheduler independent*, since the scheduler is typically outside of the language specification, and violations of scheduler assumptions may lead to vulnerabilities. By contrast, Boudol and Castellani [9] present a type system for schedulers and threads, and show that well-typed schedulers and threads satisfy a definition of security [4].

Barthe et al. [5] have developed a framework for security of multi-threaded programs that allows programs to be written without knowledge of the scheduler, i.e., in a scheduler-independent manner. Mechanisms to interact with the scheduler and secure timing channels are introduced during compilation, and enable a security-aware scheduler to enforce strong information security guarantees.

Mantel and Sabelfeld [24] show a scheduler-independent security property in a multi-threaded while language. Russo and Sabelfeld [34] suggest a model in which threads may increase and decrease their security levels and permitted scheduling decisions depend on the security levels of threads. Mantel and Sudbrock [25] prove a security property for programs consisting of threads with assigned security levels when these are run under any scheduler in a class of robust schedulers. Robust schedulers, such as round-robin schedulers, have the property that the probability that a particular low thread will be selected to run from among all low threads remains the same if high threads are removed. Our assumptions about scheduling in the X10 runtime imply that schedulers for places are robust, in that scheduling of activities at a place cannot depend upon the existence or non-existence of activities at higher-security places.

We do not provide scheduler independence. Our type-system and security proof assume that scheduling at a place depends only on the activities currently executing at a place. While this assumption enables greater concurrency while preserving security, it perhaps violates an abstraction boundary, as it makes assumptions about the behavior of the X10 runtime that are not necessarily intended as part of the runtime's specification.

Dynamic enforcement of concurrent information security Tsai et al. [46] extend work of Li and Zdancewic [23] and Russo and Sabelfeld [33] to encode information-flow control in Haskell with support for concurrency and side-effects. However, their mechanism relies on co-operative (i.e., nonpreemptive) scheduling, which may not be suitable for modern operating systems.

Stefan et al. [43] present a dynamic information-flow control system that eliminates termination and internal timing channels, and mitigates external timing channels, without relying on co-operative scheduling. Implemented as a Haskell library, their technique requires that the security level of a thread A that waits on a forked thread B must be at least as restrictive as the information that influences the control flow of thread B. A similar restriction is true of our static mechanism: the security level of a program point p that occurs after execution of program point p' is at least as restrictive as information that influences the scheduling of p'. However, our static analysis allows us to lower the security level of p in a particular situation: for a finish $s_1; s_2$ statement, the program point context of the first program point of s_2 can be lower than program point context of the last program point of s_1 if there are no other activities running at the place. The dynamic nature of the system of Stefan et al. allow them to be more precise than our static analysis in certain situations, highlighting the incomparability of static and dynamic flow-sensitive security [35].

Le Guernic [21] uses a hybrid execution monitor (which combines static and dynamic analyses) to enforce a strong security condition. The enforcement mechanism (similar to the type system of Smith and Volpano [42]) is restrictive: while loops cannot have high-security guards, and while loops are not permitted in branches of if commands with high-security guards. These restrictions are severe enough to rule out many useful programs.

Process calculi Focardi et al. [11] establish a link between language-based security for imperative programs, and process-algebraic frameworks of security properties. However, they consider only sequential imperative programs, and do not explore concurrency. Honda et al. [18] present a security-type system for the π-calculus (further developed by Honda and Yoshida [16, 17]) to address internal timing and progress channels. In their type system, channels are assigned security levels, and may be given linear types. Linear channels must statically have a single send and receive, which enables precise reasoning about synchronization between processes. Non-linear channels may have non-deterministic behavior, and processes cannot send low-security outputs after receiving high-security input on a non-linear channel, as the resolution of the non-determinism may be a covert channel. Pottier [31] presents a simpler type system (without linear channel types, and with a simpler proof) that also prevents low-security outputs after receiving high-security input.

Kobayashi [20] presents a type system for π-calculus that allows low-security output after high-security synchronization for a variety of synchronization mechanisms. It extends the idea of linear channels by using types to describe the *channel usage*, which permits precise reasoning about the information flow resulting from synchronization.

Security-type systems Other work concerned with information security in concurrent systems have also used security-type systems to enforce strong semantic security conditions (e.g., [5, 9, 36, 39, 41, 42, 44, 47, 48]). Some of these previous security-type systems are overly restrictive on synchronization between threads, either disallowing low-security output after synchronization with high-security threads or activities, or disallowing nondeterminism even when resolution of the nondeterminism is not influenced by high-security information.

The key difference in this work is that we integrate information-security guarantees with modern concurrency abstractions (i.e., X10 places). In doing so, we reason about information at coarser granularity than previous work, which simplifies reasoning about information flow (and thus, we believe, leads to increased practicality). Our type system does not track information flow on a per-location basis, but rather focuses on tracking how interaction with high-security places affects the scheduling of program points at a place.

6. Conclusion

We have extended the X10 concurrent programming language with coarse-grained information-flow control. The resulting language, SX10, provides information security for concurrent programs. Each place is associated with a security level, and may only handle data that is appropriate for the security level. We believe this language provides a better intuition for information flow than previous methods for controlling information flow, and will allow programmers to write secure programs more effectively.

The security analysis benefits from X10's abstractions for concurrency: potentially dangerous information flows correspond to interactions between places, which are relatively easy to detect, since communication between places is by message passing. Interaction between places may result in the scheduling of activities at a place being influenced by high-security information. Through a may-happen-in-parallel analysis for X10 [22], our security analysis will determine when this situation may arise, and requires *observational determinism* [48] to hold, which prevents activity scheduling from being a covert information channel. In the absence of interaction between places with different security levels, our security mechanism places no restrictions on the concurrent program. While some restrictions on concurrency necessarily remain, this allows a large class of useful programs to be written without burdensome synchronization between threads for the purposes of security.

This work highlights the opportunity for synergy between mechanisms for concurrency and mechanisms for information security: both rely on reasoning about dependencies within a program. We believe it is a promising step towards languages and tools for building secure concurrent systems.

Acknowledgments

We thanks Greg Morrisett, Eddie Kohler, and the anonymous reviewers for their helpful comments. This research is supported by the National Science Foundation under Grant No. 1054172.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Conference Record of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[2] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 207–221. IEEE Computer Society, 2007.

[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security*, Oct. 2008.

[4] G. Barthe and L. P. Nieto. Formally verifying information flow type systems for concurrent and thread systems. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 13–22, New York, NY, USA, 2004. ACM.

[5] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. *ACM Transactions on Information and System Security*, 13(3):21:1–21:32, July 2010.

[6] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[7] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 97–116, New York, NY, USA, 2009. ACM.

[8] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 535–548, New York, NY, USA, 2011. ACM.

[9] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, June 2002.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[11] R. Focardi, S. Rossi, and A. Sabelfeld. Bridging language-based and process calculi security. In *Foundations of Software Science and Computation Structure*, volume 3441 of *Lecture Notes in Computer Science*, pages 299–315, Edinburgh, UK, Apr. 2005. Springer-Verlag.

[12] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, Apr. 1982.

[13] R. Grabowski and L. Beringer. Noninterference with dynamic security domains and policies. In *13th Asian Computing Science Conference, Focusing on Information Security and Privacy*, 2009.

[14] D. Grove, O. Tardieu, D. Cunningham, B. Herta, I. Peshansky, and V. Saraswat. A performance model for x10 applications. In *Proceedings of The First X10 Workshop*, 2011.

[15] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[16] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Conference Record of the Twenty-Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 81–92, New York, NY, USA, Jan. 2002. ACM Press.

[17] K. Honda and N. Yoshida. Noninterference through flow analysis. *Journal of Functional Programming*, 15(2):293–349, Mar. 2005.

[18] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proceedings of the Ninth European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer, 2000.

[19] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, 2006.

[20] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.

[21] G. Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 218–232, 2007.

[22] J. K. Lee and J. Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 25–36, New York, NY, USA, January 2010. ACM.

[23] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 16–27, Washington, DC, USA, 2006. IEEE Computer Society.

[24] H. Mantel and A. Sabelfeld. A generic approach to the security of multi-threaded programs. In *Proceedings of the 14th*

IEEE Computer Security Foundations Workshop, page 126, Washington, DC, USA, 2001. IEEE Computer Society.

[25] H. Mantel and H. Sudbrock. Flexible scheduler-independent security. In *Proceedings of the 15th European Conference on Research in Computer Security*, pages 116–133, Berlin, Heidelberg, 2010. Springer-Verlag.

[26] H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF)*, pages 218–232. IEEE Computer Society, 2011.

[27] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992.

[28] S. Moore, A. Askarov, and S. Chong. Precise enforcement of progress-sensitive security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, New York, NY, USA, 2012. ACM Press.

[29] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[30] K. R. O'Neill, M. R. Clarkson, and S. Chong. Information-flow security for interactive programs. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 190–201. IEEE Computer Society, June 2006.

[31] F. Pottier. A simple view of type-secure information flow in the π-calculus. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 320–330, June 2002.

[32] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 114–127, Washington, DC, USA, 1995. IEEE Computer Society.

[33] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics*, volume 4378 of *Lecture Notes in Computer Science*, pages 474–480. Springer-Verlag, 2006.

[34] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, pages 177–189, 2006.

[35] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2010.

[36] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 225–239. Springer-Verlag, 2002.

[37] A. Sabelfeld. Confidentiality for multithreaded programs via bisimulation. In *Proceedings of the Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 260–273. Springer-Verlag, 2003.

[38] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[39] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society, July 2000.

[40] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. *X10 Language Specification: Version 2.1.2*, Feb. 2011. Available at `http://x10.sourceforge.net/\-documentation/\-languagespec/\-x10-212.pdf`.

[41] G. Smith. A new type system for secure information flow. In *Proceedings of the Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 115–125. IEEE Computer Society, June 2001.

[42] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 355–364, New York, NY, USA, Jan. 1998. ACM Press.

[43] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, June 2012. ACM Press.

[44] T. Terauchi. A type system for observational determinism. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 287–300, June 2008.

[45] T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *ACM Transactions on Programming Languages and Systems*, 30(5):27:1–27:30, Sept. 2008.

[46] T. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in haskell. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 187–202, Washington, DC, USA, 2007. IEEE Computer Society.

[47] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pages 34–45, Washington, DC, USA, 1998. IEEE Computer Society.

[48] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pages 29–43, Pacific Grove, California, June 2003. IEEE Computer Society.

[49] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *Formal Aspects in Security and Trust*, Toulouse, France, Aug. 2004.

A. Proof of Security Condition

We begin by proving preservation. That is, if a tree in a configuration is well-typed and that configuration takes a step, the resulting tree is also well-typed.

Lemma 1 (Preservation)**.** *For any $\ell \in \mathcal{L}$, and configurations $m = (H; \omega; R; t; T), m' = (H'; \omega; R'; t'; T')$ such that $\Gamma; \Delta \vdash T$, if $m \to m'$, then $\Gamma; \Delta \vdash T'$.*

We first prove the following claim, which states that the composition of two well-typed statements is well-typed and ends at the ending security level of the second statement.

Claim 1. *If $\Gamma; \Delta \vdash s_1 : \ell_1$ and $\Gamma; \Delta \vdash s_2^{p_2} : \ell_2$ and $\ell_1 \sqsubseteq \Delta(p_2)$, then $\Gamma; \Delta \vdash s_1 \bullet s_2 : \ell_2$.*

Proof. By induction on the length of s_1 and the definition of statement composition. □

Proof of Lemma 1. By case analysis on the derivation of $m \to m'$. Interesting cases are:

- ATSTMT Then $T = \langle P_1, \mathsf{at}\ P_2\ s_1; s_2^p\rangle$. By inversion of T-AT, $\Gamma; \Delta \vdash s_1 : \ell_1$, $\Gamma; \Delta \vdash s_2 : \ell_2$ and $\ell_1 \sqsubseteq \Delta(p_2)$. Applying T-BACKAT, $\Gamma; \Delta \vdash \mathsf{backat}\ P_1; s_2 : \ell_2$. Composing this with s_1 and applying T-ACTIVITY gives $\Gamma; \Delta \vdash T'$.
- WHILE Then $T = \langle P, \mathsf{while}^p\ e\ \mathsf{do}\ s_1^{p_1}; s_2^{p_2}\rangle$,
$T' = \langle P, \mathsf{if}\ e\ \mathsf{then}\ s_1 \bullet (\mathsf{while}\ e\ \mathsf{do}\ s_1); \mathsf{skip}\ \mathsf{else}\ \mathsf{skip}; s_2\rangle$. By inversion of T-WHILE, $\Gamma; \Delta \vdash s_1 : \ell_1$, $\ell_1 \sqsubseteq \Delta(p_1)$ and $\ell_1 \sqsubseteq \Delta(p_2)$. We have $\Gamma; \Delta \vdash \mathsf{while}^p\ e\ \mathsf{do}\ s_1^{p_1}; \mathsf{skip} : \ell_1$ where $\Delta(p) = \Delta(p_1)$. Composing gives $\Gamma; \Delta \vdash s_1 \bullet (\mathsf{while}^p\ e\ \mathsf{do}\ s_1); \mathsf{skip} : \ell_1$. We also have $\Gamma; \Delta \vdash \mathsf{skip} : \ell$, where $\ell \sqsubseteq \ell_1 \sqsubseteq \Delta(p_2)$. Thus, the resulting if statement can be typed with T-IF and applying T-ACTIVITY gives $\Gamma; \Delta \vdash T'$.
- LET Then $T = \langle P, \mathsf{let}\ x = e\ \mathsf{in}\ s_1^{p_1}; s_2^{p_2}\rangle$ and $T' = \langle P, s_1[v/x] \bullet s_2\rangle$. By inversion of rule T-LET, $\Gamma; \Delta \vdash s_1 : \ell_1$. Substitution gives $\Gamma; \Delta \vdash s_1[v/x] : \ell$ since rules T-VAR and T-CONST have the same conclusions and T-CONST has no conditions. Also by inversion, $\Gamma; \Delta \vdash s_2 : \ell_2$ and $\ell_1 \sqsubseteq \Delta(p_2)$. Composing statements by the claim and applying T-ACTIVITY gives $\Gamma; \Delta \vdash T'$.

□

We now define erased trees. We say that T erases to T' at level ℓ if $T \preceq_\ell T'$. The judgments for this relation are shown in Figure 9. Some of them require the helper judgments of the form $i \preceq_\ell i'$, which act on instructions. Rules E-LOWDONE, E-JOIN, E-PARA, E-SKIP2 and E-LOW are straightforward. Rules E-SKIP1, E-HIGHSEQ, E-HIGHPARAL and E-HIGHPARAR remove high parts of a tree. Rule E-AT removes an at statement to a high place. Rule E-HIGH erases high instructions from a statement until backat or a terminal skip is reached. Rule E-BACKAT stops erasing at a backat to a low place. To ease bookkeeping in the proof, we annotate ✓ trees with a place, to create the notation $\checkmark_P$. The semantic rule SKIP1 is modified as follows:

SKIP1

$$\frac{}{(H;\ \omega;\ t;\ \langle P, \mathsf{skip}^p\rangle) \xrightarrow{p} (H;\ \omega;\ t;\ \checkmark_P)}$$

At points in a program when, for a place P, either the scheduling of activities at P is affected by high-security information, or an activity at P is waiting for the termination of high computation, the type system enforces observational determinism. We first introduce notation to formalize this notion. The function $\mathsf{PointsWaiting}(T, P)$, defined formally in Figure 10, includes all points in $\mathsf{PointsRunning}(T, P)$, as well as any program points at P that will run immediately following currently executing computation at another place. Using this notation, observational determinism is enforced at P when for all $p \in \mathsf{PointsWaiting}(T, P)$, we have $\Delta(p) \not\sqsubseteq \mathcal{L}(P)$.

It will become important later in the proof to be able to reorder the execution of instructions at such points. This can be done because observational determinism ensures that the order of execution does not affect the order of observable events. However, when a configuration is erased, we need a method to track what parts of the program allow reordering, since it is impossible to tell from an erased configuration when the scheduling of threads might depend on high computation. A natural vehicle for this information is the refiner. We introduce underspecified refiners, which allow nondeterministic choice of the next instruction to execute. When a configuration is erased, certain scheduling functions in the refiner are replaced with $*$, allowing instructions that may happen in parallel to be executed in an arbitrary order at points where observational determinism applies and reordering is thus allowed. To facilitate this, we add a rule to the operational semantics:

$\boxed{T \preceq_\ell T}$

E-LowDone
$$\frac{\mathcal{L}(P) \sqsubseteq \ell}{\checkmark_P \preceq_\ell \checkmark_P}$$

E-Join
$$\frac{T_1 \preceq_\ell T_1' \qquad T_2 \preceq_\ell T_2'}{T_1 \triangleright T_2 \preceq_\ell T_1' \triangleright T_2'}$$

E-Para
$$\frac{T_1 \preceq_\ell T_1' \qquad T_2 \preceq_\ell T_2'}{T_1 \,||\, T_2 \preceq_\ell T_1' \,||\, T_2'}$$

E-Skip1
$$\frac{\mathcal{L}(P) \not\sqsubseteq \ell}{\langle P, \mathsf{skip}\rangle \preceq_\ell \checkmark_P}$$

E-Skip2
$$\frac{\mathcal{L}(P) \sqsubseteq \ell}{\langle P, \mathsf{skip}\rangle \preceq_\ell \langle P, \mathsf{skip}\rangle}$$

E-HighSeq
$$\frac{\mathcal{L}(P) \not\sqsubseteq \ell}{\checkmark_P \triangleright T \preceq_\ell T}$$

E-HighParaL
$$\frac{\mathcal{L}(P) \not\sqsubseteq \ell}{\checkmark_P \,||\, T \preceq_\ell T}$$

E-HighParaR
$$\frac{\mathcal{L}(P) \not\sqsubseteq \ell}{T \,||\, \checkmark_P \preceq_\ell T}$$

E-At
$$\frac{\mathcal{L}(P') \not\sqsubseteq \ell}{\langle P, \mathsf{at}\ P'\ s_1; s_2\rangle \preceq_\ell \langle P, s_2\rangle}$$

E-High
$$\frac{\begin{array}{c}\langle P, s\rangle \preceq_\ell \langle P', s'\rangle \\ \mathcal{L}(P) \not\sqsubseteq \ell \\ i \neq \mathsf{backat}\ P'\end{array}}{\langle P, i; s\rangle \preceq_\ell \langle P', s'\rangle}$$

E-Backat
$$\frac{\begin{array}{c}\langle P', s\rangle \preceq_\ell \langle P', s'\rangle \\ \mathcal{L}(P) \not\sqsubseteq \ell\end{array}}{\langle P, \mathsf{backat}\ P'; s\rangle \preceq_\ell \langle P', s'\rangle}$$

E-Low
$$\frac{\begin{array}{c}i \preceq_\ell i' \\ \langle P, s\rangle \preceq_\ell \langle P, s'\rangle \\ \mathcal{L}(P) \sqsubseteq \ell \\ i \neq \mathsf{at}\ P'\ s_1, P' \not\sqsubseteq \ell\end{array}}{\langle P, i; s\rangle \preceq_\ell \langle P, i'; s'\rangle}$$

$\boxed{i \preceq_\ell i'}$

$$\frac{}{\mathsf{skip} \preceq_\ell \mathsf{skip}} \qquad \frac{}{r := e \preceq_\ell r := e} \qquad \frac{}{\mathsf{output}\ e \preceq_\ell \mathsf{output}\ e} \qquad \frac{}{\mathsf{input}\ r \preceq_\ell \mathsf{input}\ r} \qquad \frac{}{\mathsf{backat}\ P \preceq_\ell \mathsf{backat}\ P} \qquad \frac{s \preceq_\ell s'}{\mathsf{finish}\ s \preceq_\ell \mathsf{finish}\ s'}$$

$$\frac{s \preceq_\ell s'}{\mathsf{async}\ s \preceq_\ell \mathsf{async}\ s'} \qquad \frac{s \preceq_\ell s'}{\mathsf{at}\ P\ s \preceq_\ell \mathsf{at}\ P\ s'} \qquad \frac{s \preceq_\ell s'}{\mathsf{let}\ x = e\ \mathsf{in}\ s \preceq_\ell \mathsf{let}\ x = e\ \mathsf{in}\ s'} \qquad \frac{s \preceq_\ell s'}{\mathsf{while}\ e\ \mathsf{do}\ s \preceq_\ell \mathsf{while}\ e\ \mathsf{do}\ s'}$$

$$\frac{s_1 \preceq_\ell s_1' \qquad s_2 \preceq_\ell s_2'}{\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \preceq_\ell \mathsf{if}\ e\ \mathsf{then}\ s_1'\ \mathsf{else}\ s_2'}$$

Figure 9. Judgments for erasure

$$\begin{aligned}
\mathsf{PointsWaiting}(\checkmark, P) &= \emptyset \\
\mathsf{PointsWaiting}(\langle P', i^p; s\rangle, P) &= \begin{cases} \{p\} & \text{if } P' = P \\ \mathsf{PointsWaiting}(\langle P', s\rangle, P) & \text{if } P' \neq P \text{ and } i \neq \mathsf{backat}\ P'' \\ \mathsf{PointsWaiting}(\langle P'', s\rangle, P) & \text{if } P' \neq P \text{ and } i = \mathsf{backat}\ P'' \end{cases} \\
\mathsf{PointsWaiting}(T_1 \,||\, T_2, P) &= \mathsf{PointsWaiting}(T_1, P) \cup \mathsf{PointsWaiting}(T_2, P) \\
\mathsf{PointsWaiting}(T \triangleright \langle P', s\rangle, P) &= \mathsf{PointsWaiting}(T, P)
\end{aligned}$$

Figure 10. Definition of $\mathsf{PointsWaiting}(T, P)$

PlaceNonDet
$$\frac{\begin{array}{c}(H;\, \omega;\, t;\, T) \xrightarrow{p} (H';\, \omega;\, t';\, T') \\ \mathsf{PointsRunning}(T, P) \neq \emptyset \quad Sch(P) = * \cdot chs \quad Place(p) = P\end{array}}{(H; \omega; (P \cdot Ps, Sch); t; \langle P, s\rangle) \to (H'; \omega; (Ps, Sch[P \to chs]); t'; T')}$$

The next several definitions present relations between specified and underspecified refiners, and then formalize when reordering of instructions is allowed in an erased configuration. We first define scheduling lists to be equivalent when corresponding elements are identical or one is $*$.

Definition 6. Let $E_0 \cdot chs_0$ and $E_1 \cdot chs_1$ be scheduling lists (recall that a scheduler Sch is a function from places to scheduling lists). These two lists are equivalent, written $E_0 \cdot chs_0 \sim E_1 \cdot chs_1$, if $chs_0 \sim chs_1$ and

- $E_0 = E_1$,
- $E_0 = ch$ and $E_1 = *$ or
- $E_0 = *$ and $E_1 = ch$.

Refiners are low-equivalent when the place lists are low equivalent and corresponding scheduling lists for each low place are equivalent.

Definition 7 (Low-equivalence of refiners)**.** For $\ell \in \mathcal{L}$, refiners $R_0 = (Ps_0, Sch_0)$ and $R_1 = (Ps_1, Sch_1)$ are low-equivalent, written $R_0 \sim_\ell R_1$, if $Ps_0\restriction_{LP} = Ps_1\restriction_{LP}$, where $LP = \{P \mid \mathcal{L}(P) \sqsubseteq \ell\}$ and for all $P \in LP$, $Sch_0(P) \sim Sch_1(P)$ (this relation was defined in Definition 6.)

A refiner $R = (Ps, Sch)$ such that $Sch(P)$ does not contain $*$ for any place P is referred to as a specified refiner. Otherwise, R is an underspecified refiner. A specified refiner erases to an underspecified refiner when the underspecified place list is the restriction of the specified place list to low places and each element of a scheduling list at a low place in the underspecified refiner either matches the corresponding element in the specified refiner or is $*$.

Definition 8 (Erasure of refiners)**.** For $\ell \in \mathcal{L}$, refiner $R_0 = (Ps_0, Sch_0)$ erases to refiner $R_1 = (Ps_1, Sch_1)$, written $R_0 \preceq_\ell R_1$, if

- $Ps_0 = P \cdot Ps'_0$, where $\mathcal{L}(P) \sqsubseteq \ell$, $Ps_1 = P \cdot Ps'_1$,
  $Sch_0(P) = ch \cdot chs_0$, $Sch_1(P) = ch \cdot chs_1$, and
  $(Ps'_0, Sch_0[P \mapsto chs_0]) \preceq_\ell (Ps'_1, Sch_1[P \mapsto chs_1])$
- $Ps_0 = P \cdot Ps'_0$, where $\mathcal{L}(P) \sqsubseteq \ell$, $Ps_1 = P \cdot Ps'_1$,
  $Sch_0(P) = ch \cdot chs_0$, $Sch_1(P) = * \cdot chs_1$, and
  $(Ps'_0, Sch_0[P \mapsto chs_0]) \preceq_\ell (Ps'_1, Sch_1[P \mapsto chs_1])$
- $Ps_0 = P \cdot Ps'_0$, where $\mathcal{L}(P) \not\sqsubseteq \ell$ and $(Ps'_0, Sch_0) \preceq_\ell R_1$

Note that this implies that if $R_0 \preceq_\ell R_1$, then Ps_1 contains no place P such that
$\mathcal{L}(P) \not\sqsubseteq \ell$.

As was stated above, execution may consume a $*$ from an underspecified refiner, and execute an arbitrary instruction, only if observational determinism was enforced at the corresponding point in the original configuration. We determine whether this holds by matching steps in an original (non-erased) configuration with the scheduler element they consume. An underspecified refiner with the property that $*$ is consumed when observational determinism applies is called *properly formed*.

Definition 9. Let $m_0 \to m_1 \to \ldots \to m_n$ be a sequence of configurations, where $m_0 = (H; \omega; R; t; T)$ is well-typed. Let E_i be the scheduler element consumed in the step $m_i \to m_{i+1}$ and let T_i be the tree for configuration m_i. An underspecified refiner R is *properly formed* with respect to the sequence of configurations $m_0 \to^* m_n$ if, for $i \in [0, n-1]$, $E_i = *$ if and only if $\Delta(p) \neq \mathcal{L}(P)$ for all $p \in \mathsf{PointsWaiting}(T_i, P)$. Note that the Δ values used here are preserved from the initial configuration.

The following lemma, which will be useful throughout the proof, gives conditions under which identical expressions will evaluate to identical values. Part a states that identical expressions will evaluate to identical values under heaps that are equivalent for any references which the expression reads. Part b states that identical expressions at a low place will evaluate to identical values under low-equivalent heaps.

Lemma 2.

(a) Let C be a (possibly empty) set of locations. If $H_1\restriction_C = H_2\restriction_C$, e is part of a well-typed configuration and dereferences only locations in the set C, and $P; H_1; e \Downarrow v$, then $P; H_2; e \Downarrow v$.

(b) If $H_1 \sim_\ell H_2$ for some $\ell \in \mathcal{L}$ such that $\mathcal{L}(P) \sqsubseteq \ell$, e is part of a well-typed configuration and $P; H_1; e \Downarrow v$, then $P; H_2; e \Downarrow v$.

Proof. By induction on the derivation of $P; H_1; e \Downarrow v$.

(a)
- CONST is trivial.
- REF By inversion, $H_1(r) = v$. By assumption, $r \in C$, so $H_2(r) = v$ and $P; H_2; !r \Downarrow v$.
- OP $e = e_1 \oplus e_2$. By inversion, $P; H_1; e_1 \Downarrow v_1$, $P; H_1; e_2 \Downarrow v_2$, and $v_1 \oplus v_2 = v$. e_1 and e_2 dereference only locations in C, so by induction, $P; H_2; e_1 \Downarrow v_1$ and $P; H_2; e_2 \Downarrow v_2$. Applying OP gives $P; H_2; e \Downarrow v$.

(b) is similar, noting that $H_1 \sim_\ell H_2$ implies that the condition for part a is met for all locations at low places P.

□

The following three lemmas deal with simulating an original configuration against its erasure. We first show that at points where reordering is not possible and observational determinism does not apply, the original and erased configurations must agree on the set of instructions currently running.

Lemma 3. *If* $\Gamma;\Delta \vdash T$*,* $T \preceq_\ell T'$ *and* $\Delta(p) \sqsubseteq \ell$ *for some* $p \in \mathsf{PointsWaiting}(T,P)$*, then* $\mathsf{PointsRunning}(T,P) = \mathsf{PointsRunning}(T',P)$*.*

Proof. Let $p \in \mathsf{PointsRunning}(T,P)$. $\mathcal{L}(P) \sqsubseteq \ell$ since $\mathcal{L}(P) \sqsubseteq \Delta(p)$, so by the definition of erasure, $p \in \mathsf{PointsRunning}(T',P)$. Now let $p \in \mathsf{PointsRunning}(T',P)$ and suppose $p \notin \mathsf{PointsRunning}(T,P)$. By inspection of the rules for erasure, since $\langle P', s'^p\rangle \in T'$, there exists a $\langle P, s^p\rangle$ in T such that $\langle P,s\rangle \preceq_\ell \langle P',s'\rangle$, since only E-AT, E-HIGH, E-BACKAT, and E-LOW apply, and all have $T = \langle P,s\rangle$. We have $P' \neq P$, since otherwise, by the erasure rules, s and s' would start with the same instruction, so we would have $p \in \mathsf{PointsRunning}(T,P)$. Again by inspection of the rules, we have $\mathcal{L}(P) \not\sqsubseteq \ell$ and, applying the case for that condition, T must contain $\langle P', s_1; \mathsf{backat}^{p_1}\, P; i^p; s_2\rangle$, and thus $p \in \mathsf{PointsWaiting}(T,P)$. However, by the typing rule for backat, $\mathcal{L}(P') \sqsubseteq \Delta(p)$, and so $\Delta(p) \not\sqsubseteq \ell$. We also have that for all $p' \in \mathsf{PointsRunning}(T,P)$, $p' \in \mathsf{MHPP}(p_1)$, and so $\mathcal{L}(P') \sqsubseteq \Delta(p')$ for all p', which is a contradiction. □

Lemmas 4 and 5 show the main result of this section of the proof. Lemma 4 runs a step-by-step simulation of an original configuration against its erasure. At each step, the original configuration may take a step at a high place (the first case of the lemma), the original configuration may take a step at a different high place, specified by a refiner that is low-equivalent to the original refiner used (the second case), or both configurations may take a step at a low place (the third case). In any of these cases, the resulting non-erased configuration still corresponds to the resulting erased configuration. Lemma 5 applies this inductively to show that configurations produce traces low-equivalent to those produced by their erasures.

Lemma 4 (Simulation). *Let* $\ell \in \mathcal{L}$*, and* $m_1 = (H_1;\omega;R_1;t_1;T_1)$ *and* $m_2 = (H_2;\omega;R_2;t_2;T_2)$ *be well-typed configurations. Let* T_1', H_1' *and* t_1' *be such that* $T_1 \preceq_\ell T_1'$*,* $H_1 \sim_\ell H_1'$*,* $t_1 \sim_\ell t_1'$*. If* $m_1 \to m_2$ *and* R_1 *is properly formed with respect to this step, then one of these cases holds:*

- $R_2 \sim_\ell R_1$*,* $H_2 \sim_\ell H_1'$*,* $t_2 \sim_\ell t_1'$*, and* $T_2 \preceq_\ell T_1'$
- *There exist* R, R', H_2', t_2' *and* T_2' *such that* $(H_1;\omega;R;t_1;T_1) \to (H_2';\omega;R';t_2';T_2')$*,* $R \sim_\ell R_1$*,* $H_2' \sim_\ell H_1'$*,* $t_2' \sim_\ell t_1'$*, and* $T_2' \preceq_\ell T_1'$
- *there exists a configuration* $m_2' = (H_2';\omega;R_2';t_2';T_2')$ *and a refiner* R_1' *such that* $(H_1';\omega;R_1';t_1';T_1') \to m_2'$ *with* R_1' *properly formed with respect to this step and* $R_1 \preceq_\ell R_1'$*. In addition,* $H_2 \sim_\ell H_2'$*,* $t_2 \sim_\ell t_2'$ *and* $T_2 \preceq_\ell T_2'$*.*

Proof. By cases on the derivation of $m_1 \to m_2$.

- If this transition instantiates JOINDONE, PARALEFTDONE or PARARIGHTDONE, then m_1' can take the same step to a configuration containing a new tree T_2'. Heaps and traces are not modified and $T_2 \preceq_\ell T_2'$ by induction, so the third case of the lemma applies.
- Suppose this transition instantiates IDLE with $R_1 = (P{\cdot}Ps, Sch)$ and $\mathsf{PointsRunning}(T_1',P) = \emptyset$. Then $(H_1';\omega;R_1';t_1';T_1') \to (H_1';\omega;R_2';t_1';T_1')$ for some R_1' such that $R_1 \preceq_\ell R_1'$, and this reduction instantiates IDLEPLACE. Since no instructions were executed, the third case of the lemma applies.
- If it instantiates IDLEPLACE with $R_1 = (P \cdot Ps, Sch)$ and $\mathsf{PointsRunning}(T_1',P) \neq \emptyset$, then there is some $p \in \mathsf{PointsWaiting}(T_1,P)$ waiting for return of control from a place $P', \mathcal{L}(P') \not\sqsubseteq \ell$. $(H_1;\omega;(P' \cdot P \cdot Ps, Sch);t_1;T_1) \to (H_2';\omega;R_1;t_2';T_2')$ for some H_2', t_2', T_2', ch. $(P' \cdot P \cdot Ps, Sch) \sim_\ell R_1$. The second case of the lemma applies.

  Otherwise, $T_1 = \langle P, i; s\rangle$. We proceed further in cases on the execution of this statement.
- Suppose $m_1 \to m_2$ executes an instruction at a place P, such that $\mathcal{L}(P) \not\sqsubseteq \ell$. Then this step must only modify high parts of the tree, heap and trace, so the first case of the lemma applies.
- Suppose $m_1 \to m_2$ executes an activity $\langle P, \mathsf{at}^p\, P'\, s_1; s_2\rangle$ such that $\mathcal{L}(P) \sqsubseteq \ell$ but $\mathcal{L}(P') \not\sqsubseteq \ell$. By rule E-AT, $T_1' = \langle P, s_2\rangle$, so $T_2 \preceq_\ell T_1'$ still holds and the first case of the lemma applies.
- Suppose $R_1 = (P \cdot Ps, Sch)$ such that $\mathcal{L}(P) \sqsubseteq \ell$ and $m_1 \to m_2$ executes a program point $p \in \mathsf{PointsRunning}(T_1,P)$. If for some $p \in \mathsf{PointsWaiting}(T_1,P)$ we have $\Delta(p) = \mathcal{L}(P) \sqsubseteq \ell$, then let $R_1' = (Ps' \cdot P, Sch')$, where $(Ps, Sch) \preceq_\ell (Ps', Sch')$. By Lemma 3, we have $\mathsf{PointsRunning}(T_1,P)$=$\mathsf{PointsRunning}(T_1',P)$, so instantiating PLACE on both configurations will execute the same program point p. Otherwise, let $R_1' = (P \cdot Ps', Sch'[P \mapsto * \cdot chs])$ such that $R_1 \preceq_\ell R_1'$. In

either case, there exists a configuration m'_2 such that $(H'_1; \omega; R'_1; t'_1; T'_1) \rightarrow m'_2$ instantiates PLACE or PLACENONDET and executes p. Straightforward case analysis on $i; s$ and $i'; s'$ gives the conclusions for the third case of the lemma.

□

Lemma 5. *For any $\ell \in \mathcal{L}$, and for all well-typed configurations $m_1 = (H_1; \omega; R_0; t_1; T_1)$, if $T_1 \preceq_\ell T'_1$, $H_1 \sim_\ell H'_1$, $t_1 \sim_\ell t'_1$ and (T_1, ω, R_0) emits t_0, then there exist t, t', R_1, R'_1 such that*

$$R_0 \sim_\ell R_1, \;\; R_1 \preceq_\ell R'_1$$

$$(T_1, \omega, R_1) \;\mathsf{emits}\; t$$

$$(T'_1, \omega, R'_1) \;\mathsf{emits}\; t'$$

$$t_0 \sim_\ell t \sim_\ell t'$$

and R'_1 is properly formed with respect to the derivation of (T'_1, ω, R'_1) emits t'.

Proof. This is an inductive application of Lemma 4. Handling the first case of the lemma is straightforward, and the second case of the lemma is straightforward keeping in mind the transitivity of low-equivalence of refiners. In the third case of the lemma, there exists a configuration $m'_2 = (H'_2; \omega; R'_2; t'_2; T'_2)$ and a refiner R'_0 such that $(H'_1; \omega; R'_0; t'_1; T'_1) \rightarrow m'_2$ with R'_0 properly formed with respect to this step and $R_0 \preceq_\ell R'_0$. In addition, $H_2 \sim_\ell H'_2$, $t_2 \sim_\ell t'_2$ and $T_2 \preceq_\ell T'_2$. By induction, there exist t_i, t'_i, R_i, R'_i such that $R_2 \sim_\ell R_i$, $R_i \preceq_\ell R'_i$, (T_2, ω, R_i) emits t_i, (T'_2, ω, R'_i) emits t'_i and R'_i is properly formed with respect to the latter derivation. Suppose $R_i = (Ps_i, Sch_i), R'_i = (Ps'_i, Sch'_i), R_0 = (Ps_0, Sch_0), R'_0 = (Ps'_0, Sch'_0)$. Suppose the step $m_1 \rightarrow m_2$ consumed a scheduler function ch at place P. Let $R_1 = (P \cdot Ps_i, Sch_i[P \mapsto ch \cdot Sch_i(P)])$. Similarly, suppose the step $(H'_1; \omega; R'_0; t'_1; T'_1) \rightarrow m'_2$ consumed a scheduler element E at place P (E may be a scheduler function, $*$ or ϵ indicating no element was consumed). Let $R'_1 = (P \cdot Ps'_i, Sch_i[P \mapsto E \cdot Sch'_i(P)])$. Since $R_i \preceq_\ell R'_i$, we have $R_1 \preceq_\ell R'_1$. Since $R_2 \sim_\ell R_i$, we have $R_0 \sim_\ell R_1$. Since R'_i was properly formed with respect to its derivation and R'_0 was properly formed with respect to the step $(H'_1; \omega; R'_0; t'_1; T'_1) \rightarrow m'_2$, R'_1 is properly formed with respect to this step, followed by the remainder of the derivation. □

The next section of the proof simulates two erased configurations against each other. The next definition formalizes the notion of an erased configuration. The proof then proceeds by showing the diamond property: if two erased configurations begin in agreement and take two different steps, they may each be made to take another step such that the resulting configurations agree. In particular, Lemmas 6 and 7 show that if a configuration executes p_0 followed by p_1 and an identical configuration executes p_1 followed by p_0, the resulting configurations will agree on their heaps, traces and trees. This does not require observational determinism throughout the program, because we are considering configurations with equivalent refiners.

Definition 10 (Erased Configuration)**.** A configuration $m = (H; \omega; R; t; T)$ is erased at level $\ell \in \mathcal{L}$ if there exists a T_0 such that $T_0 \preceq_\ell T$.

Lemma 6. *Let $\ell \in \mathcal{L}$ and $m_0 = (H_0; \omega_0; R_0; t_0; \langle P, i^p; s\rangle)$ and $m_1 = (H_1; \omega_1; R_1; t_1; \langle P, i^p; s\rangle)$ be well-typed erased configurations. Suppose that p reads only over channels in the set I and writes only over channels in O (either or both sets may be empty) and that $\omega_0 \sim_\ell \omega_1$, $H_0\restriction_I = H_1\restriction_I$, and $t_0\restriction_I = t_1\restriction_I$. If*
$m_0 \rightarrow (H'_0; \omega_0; R'_0; t'_0; T'_0)$ and
$m_1 \rightarrow (H'_1; \omega_1; R'_1; t'_1; T'_1)$,
then $H'_0\restriction_C = H'_1\restriction_C$ and $t'_0\restriction_C = t'_1\restriction_C$, where $C = I \cup O$, and $T'_0 = T'_1$.

Proof. Since the instruction executed is the same in both reductions, proceed in cases on i. Note that since these are erased configurations at ℓ, it must be the case that $\mathcal{L}(P) \sqsubseteq \ell$. Otherwise, no instruction would execute at P in these configurations. This allows us to apply Lemma 2 for the expressions of assignments and input and output instructions, which gives $H'_0\restriction_C = H'_1\restriction_C, t'_0\restriction_C = t'_1\restriction_C$, since the changes to heaps and traces depend only on the instruction itself and on the constant to which the expression evaluates. For all other instructions, heaps and traces are unchanged and these equalities are trivial. $T'_0 = T'_1$ follows from a straightforward analysis of each case. □

Lemma 7. *Let $\ell \in \mathcal{L}$ and $m_0 = (H;\, \omega_0;\, t;\, T)$ and $m_1 = (H;\, \omega_1;\, t;\, T)$ be well-typed erased configurations, with $\Gamma; \Delta \vdash T$. Suppose $p_0, p_1 \in \mathsf{PointsRunning}(T, P)$, $\Delta(p_0) \not\sqsubseteq \mathcal{L}(P)$ and $\Delta(p_1) \not\sqsubseteq \mathcal{L}(P)$ (using Δ from the original, non-erased configuration). If $m_0 \overset{p_0}{\rightarrow} m'_0 \overset{p_1}{\rightarrow} m''_0$ and $m_1 \overset{p_1}{\rightarrow} m'_1 \overset{p_0}{\rightarrow} m''_1$ to reach final configurations*
$m''_0 = (H''_0;\, \omega_0;\, t''_0;\, T''_0)$ and $m''_1 = (H''_1;\, \omega_1;\, t''_1;\, T''_1)$,
then $H''_0 = H''_1$, $t''_0 = t''_1$ and $T''_0 = T''_1$.

Proof. Let $m'_0 = (H'_0;\, \omega_0;\, t'_0;\, T'_0)$ be the intermediate step such that $m_0 \xrightarrow{p_0} m'_0$ and $m'_1 = (H'_1;\, \omega_1;\, t'_1;\, T'_1)$ be the intermediate step such that $m_1 \xrightarrow{p_1} m'_1$. Let $S_0 = \langle P, i_0^{p_0}; s_0\rangle, S_1 = \langle P, i_1^{p_1}; s_1\rangle$. S_0 and S_1 are both contained in T. Suppose

$$(H_0;\, \omega_0;\, t_0;\, S_0) \xrightarrow{p_0} (H_0;\, \omega_0;\, t_0;\, S'_0), (H_1;\, \omega_1;\, t_1;\, S_1) \xrightarrow{p_1} (H_1;\, \omega_1;\, t_1;\, S'_1)$$

By the reduction rules on trees, T'_0 is equal to T with S_0 replaced by S'_0 and T'_1 is T with S_1 replaced by S'_1.

Suppose instruction i_0 reads only on channels in set I_0 and writes only on channels in set O_0, and that instruction i_1 reads only on channels in set I_1 and writes only on channels in set O_1. Since $p_0 \in \mathsf{MHPP}(p_1)$, the definitions of $\mathit{noWrite}(\cdot,\cdot)$, $\mathit{noReadWrite}(\cdot,\cdot)$ and $\mathit{noIO}(\cdot)$ ensure that

$$I_0 \cap O_1 = I_1 \cap O_0 = O_0 \cap (I_1 \cup O_1) = O_1 \cap (I_0 \cup O_0) = \emptyset$$

We now execute i_1 to produce the reduction $m'_0 \xrightarrow{p_1} m''_0$. Since executing i_0 could not alter any channel in I_1 (because the sets of channels are disjoint), we have

$$H'_0{\restriction}_{I_1} = H_1{\restriction}_{I_1} t'_0{\restriction}_{I_1} = t_1{\restriction}_{I_1}$$

The derivation of $m'_0 \xrightarrow{p_1} m''_0$ contains the reduction

$$(H'_0;\, \omega_0;\, t'_0;\, \langle P, i_1; s_1\rangle) \xrightarrow{p_1} (H''_0;\, \omega_0;\, t''_0;\, T)$$

so by Lemma 6, $H''_0{\restriction}_{I_1 \cup O_1} = H'_1{\restriction}_{I_1 \cup O_1}$ and $t''_0{\restriction}_{I_1 \cup O_1} = t'_1{\restriction}_{I_1 \cup O_1}$. Executing i_0 on m'_1 cannot alter any channel in $I_1 \cup O_1$, so we have

$$H''_0{\restriction}_{I_1 \cup O_1} = H''_1{\restriction}_{I_1 \cup O_1}, t''_0{\restriction}_{I_1 \cup O_1} = t''_1{\restriction}_{I_1 \cup O_1}$$

The symmetry of the situation gives

$$H''_1{\restriction}_{I_0 \cup O_0} = H''_0{\restriction}_{I_0 \cup O_0}, t''_1{\restriction}_{I_0 \cup O_0} = t''_0{\restriction}_{I_0 \cup O_0}$$

No other channels were modified in the execution of the two instructions, so $H''_0 = H''_1$ and $t''_0 = t''_1$.

Since $p_0 \in \mathsf{MHPP}(p_1)$, S_0 and S_1 are non-overlapping parts of the program tree T. Thus, after executing i_0, S_1 is still contained in T'_0. By Lemma 6, T''_1 is T'_1 with S_0 replaced by S'_0. By symmetry, T''_0 is T'_0 with S_1 replaced by S'_1. This makes both T''_0 and T''_1 equal to T with both of these replacements. □

Lemma 8 (Diamond Property). *Let $\ell \in \mathcal{L}$. Let*

$$m_0 = (H; \omega_0; R_0; t; T)$$

$$m_1 = (H; \omega_1; R_1; t; T)$$

$$m'_0 = (H'_0; \omega_0; R'_0; t'_0; T'_0)$$

$$m'_1 = (H'_1; \omega_1; R'_1; t'_1; T'_1)$$

be well-typed erased configurations such that $\omega_0 \sim_\ell \omega_1$, $R_0 \sim_\ell R_1$. If $m_0 \to m'_0$, $m_1 \to m'_1$ and R_0 and R_1 are properly formed with respect to their respective steps, then either

- *$t'_0 = t'_1, T'_0 = T'_1$ and $H'_0 = H'_1$ or*
- *there exist configurations $m''_0 = (H'; \omega_0; R'_0; t'; T')$ and $m''_1 = (H'; \omega_1; R'_1; t'; T')$ such that*

$$(H'_0; \omega_0; R''_0; t'_0; T'_0) \to m''_0, (H'_1; \omega_1; R''_1; t'_1; T'_1) \to m''_1$$

where $R''_0 \sim_\ell R''_1$ and these refiners are properly formed with respect to their respective steps.

Proof. By case analysis. Let $R_0 = (P \cdot Ps_0, Sch_0)$, $R_1 = (P \cdot Ps_1, Sch_1)$ (the same place must be at the front of both lists since $\mathcal{L}(P) \sqsubseteq \ell$ and$R_0 \sim_\ell R_1$).

- If $m_0 \to m'_0$ instantiates JoinDone, ParaLeftDone or ParaRightDone, then T contains $\checkmark \triangleright T'$, $\checkmark \,||\, T_2$ or $T_1 \,||\, \checkmark$ (the place attached to $\checkmark$ is not important, but must be a place whose level is below ℓ since T is erased.) The step $m_1 \to m'_1$ can instantiate the same rule. Since these rules do not depend on ω or R, we must have $T'_0 = T'_1$. Neither emits an event or modifies the heap, so so $t'_0 = t'_1 = t$ and $H'_0 = H'_1 = H$.

- Suppose $Sch_0(P) = ch \cdot chs_0$. Since R_0 is properly formed, it must be the case that $\Delta(p) \sqsubseteq \ell$ for some $p \in \mathsf{PointsWaiting}(T, P)$. Since R_1 is also properly formed with respect to a step from T and $R_0 \sim_\ell R_1$, $Sch_1(P) = ch \cdot chs_1$. Thus, $m_0 \to m_0'$ and $m_1 \to m_1'$ both instantiate PLACE with the same ch, so both execute the same instruction. The statement reduction rules only depend on H, t and $\omega(P, t\lceil_P)$, all of which are the same for both configurations, so $T_0' = T_1', H_0' = H_1'$ and $t_0' = t_1'$.
- Otherwise, $Sch_0(P) = * \cdot chs_0$. Since R_0 is properly formed, it must be the case that $\Delta(p) \not\sqsubseteq \ell$ for all $p \in \mathsf{PointsWaiting}(T, P)$. Since R_1 is also properly formed with respect to a step from T and $R_0 \sim_\ell R_1$, $Sch_1 = * \cdot chs_1$, and $m_0 \to m_0'$ and $m_1 \to m_1'$ instantiate PLACENONDET. If both execute the same instruction, then, as above, $T_0' = T_1', H_0' = H_1'$ and $t_0' = t_1'$.
- If the two configurations instantiate PLACENONDET with different instructions, then suppose $m_0 \overset{p_0}{\rightarrow} m_0'$ and $m_1 \overset{p_1}{\rightarrow} m_1'$. Since R_0 and R_1 were properly formed, $\Delta(p_0) \not\sqsubseteq \mathcal{L}(P)$ and $\Delta(p_1) \not\sqsubseteq \mathcal{L}(P)$. Let $R_0'' = R_0$, $R_1'' = R_1$. This allows the transitions
$$m_0 \to (H_0'; \omega_0; (P \cdot Ps_0, Sch_0[P \mapsto * \cdot chs_0]); t_0'; T_0') \to (H_0''; \omega_0; R_0'; t_0''; T_0'')$$
$$m_1 \to (H_1'; \omega_1; (P \cdot Ps1, Sch_1[P \mapsto * \cdot chs_1]); t_1'; T_1') \to (H_1''; \omega_1; R_1'; t_1''; T_1'')$$
where the first transition executes program point p_0 followed by p_1 and the second executes program point p_1 followed by p_0. By Lemma 7, $H_0'' = H_1''$, $t_0'' = t_1''$ and $T_0'' = T_1''$. Since R_0 and R_1 were properly formed, $\Delta(p) \neq \mathcal{L}(P)$ for all $p \in \mathsf{PointsWaiting}(T, P)$. Rules T-FINISH and T-JOIN ensure that if all activities at a place are tainted with high information, this will be the case until they all finish executing, so $\Delta(p) \neq \mathcal{L}(P)$ for all $p \in \mathsf{PointsWaiting}(T_0', P)$ and $p \in \mathsf{PointsWaiting}(T_1', P)$. Thus, $(P \cdot Ps_0, Sch_0[P \mapsto * \cdot chs_0])$ and $(P \cdot Ps_1, Sch_1[P \mapsto * \cdot chs_1])$ are properly formed with respect to their respective steps.

□

We are now prepared to show a confluence property similar to those of other proofs that use this technique [44, 45]. That is, if two erased configurations begin with identical heaps, trees and traces and low-equivalent refiners and input strategies, and each take an arbitrary numbers of steps, these configurations are confluent. Lemma 10 uses this to show that, at any given point, two erased configurations that began in agreement must have traces that agree, up to prefix.

Lemma 9 (Confluence of Erased Configurations)**.** *Let $\ell \in \mathcal{L}$ and*
$$m_0 = (H; \omega_0; R_0; t; T),$$
$$m_1 = (H; \omega_1; R_1; t; T),$$
$$m_0' = (H_0'; \omega_0; R_0'; t_0'; T_0'),$$
$$m_1' = (H_1'; \omega_1; R_1'; t_1'; T_1')$$
be well-typed erased configurations such that $\omega_0 \sim_\ell \omega_1$, $R_0 \sim_\ell R_1$. If $m_0 \to^* m_0'$, $m_1 \to^* m_1'$ and R_0 and R_1 are properly formed with respect to their respective derivations, then there exist configurations
$$m_0'' = (H'; \omega_0; R_0''; t'; T'),$$
$$m_1'' = (H'; \omega_1; R_1''; t'; T')$$
such that $(H_0'; \omega_0; \overline{R_0'}; t_0'; T_0') \to^* m_0''$, $(H_1'; \omega_1; \overline{R_1'}; t_1'; T_1') \to^* m_1''$, where $\overline{R_0'} \sim_\ell \overline{R_1'}$ and $\overline{R_0}$ and $\overline{R_1}$ are properly formed with respect to their respective derivations.

Proof. This is a standard induction on the diamond property (Lemma 8). □

Lemma 10 (Prefix-Equivalence of Erased Configurations)**.** *Let H_{init} be the default (empty) heap. For any $\ell \in \mathcal{L}$ and erased configurations*
$$m_0 = (H_{init}; \omega_0; R_0; \epsilon; T),$$
$$m_1 = (H_{init}; \omega_1; R_1; \epsilon; T)$$
such that $\Gamma; \Delta \vdash T$, $R_0 \sim_\ell R_1$, $\omega_0 \sim_\ell \omega_1$, if (T, ω_0, R_0) emits t_0, (T, ω_1, R_1) emits t_1 and R_0 and R_1 are properly formed with respect to their respective derivations, then for all P, either $t_0\lceil_P$ is a prefix of $t_1\lceil_P$ or $t_1\lceil_P$ is a prefix of $t_0\lceil_P$.

Proof. There exist configurations

$$m_0' = (H_0; \omega_0; R_0'; t_0; T_0),$$

$$m_1' = (H_1; \omega_1; R_1'; t_1; T_1)$$

such that $m_0 \to^* m_0'$ and $m_1 \to^* m_1'$. By Lemma 9, there exist configurations

$$m_0'' = (H'; \omega_0; R_0''; t'; T'),$$

$$m_1'' = (H'; \omega_1; R_1''; t'; T')$$

and refiners $\overline{R_0'}$ and $\overline{R_1'}$ such that $(H_0; \omega_0; \overline{R_0'}; t_0; T_0) \to^* m_0''$, and $(H_1; \omega_1; \overline{R_1'}; t_1; T_1) \to^* m_1''$. Thus, for all places P, $t_0\upharpoonright_P$ and $t_1\upharpoonright_P$ are both prefixes of $t'\upharpoonright_P$, and so $t_0\upharpoonright_P$ is a prefix of $t_1\upharpoonright_P$ or $t_1\upharpoonright_P$ is a prefix of $t_0\upharpoonright_P$. □

Theorem 2 combines the two major results by simulating original configurations against their corresponding erasures and the erasures against each other to show that the low restrictions of the traces produced by the original configurations agree up to prefix. If the original execution traces are t_0 and t_1 and the execution traces of the erased configurations are t_0'' and t_1'' respectively, the first simulation shows that t_0 and t_0'' are low-equivalent, as are t_1 and t_1''. The second simulation shows prefix-equivalence of t_0'' and t_1'' and transitivity gives the desired result.

Theorem 2. For any $\ell \in \mathcal{L}$ and for all configurations $m_0 = (H_{init}; \omega_0; R; \epsilon; \langle P, s\rangle)$ and $m_1 = (H_{init}; \omega_1; R; \epsilon; \langle P, s\rangle)$ such that $\Gamma; \Delta \vdash \langle P, s\rangle$, R is fully specified, and $\omega_0 \sim_\ell \omega_1$, if $(\langle P, s\rangle, \omega_0, R)$ emits t_0 and $(\langle P, s\rangle, \omega_1, R)$ emits t_1, then for all P such that $\mathcal{L}(P) \sqsubseteq \ell$, $t_0\upharpoonright_P$ is a prefix of $t_1\upharpoonright_P$ or $t_1\upharpoonright_P$ is a prefix of $t_0\upharpoonright_P$.

Proof. Let $\langle P, s'\rangle$ be such that $\langle P, s\rangle \preceq_\ell \langle P, s'\rangle$. If a program is well-typed, its erasure is well-typed with the same Δ and Γ, so $\Gamma; \Delta \vdash \langle P, s'\rangle$. By Lemma 5, there exist $t_0', t_1', t_0'', t_1'', R_0, R_1, R_0', R_1'$ such that

$$(\langle P, s\rangle, \omega_0, R_0) \text{ } \mathsf{emits} \text{ } t_0',$$

$$(\langle P, s\rangle, \omega_1, R_1) \text{ } \mathsf{emits} \text{ } t_1',$$

$$(\langle P, s'\rangle, \omega_0, R_0') \text{ } \mathsf{emits} \text{ } t_0'',$$

$$(\langle P, s'\rangle, \omega_1, R_1') \text{ } \mathsf{emits} \text{ } t_1'',$$

$$t_0 \sim_\ell t_0' \sim_\ell t_0'', t_1 \sim_\ell t_1' \sim_\ell t_1''.$$

R_0' and R_1' are properly formed with respect to their respective derivations, $R \sim_\ell R_0 \preceq_\ell R_0'$ and $R \sim_\ell R_1 \preceq_\ell R_1'$. Low-equivalence of refiners is transitive, so $R_0 \sim_\ell R_1$. Erasures of low-equivalent refiners are equivalent (this can be shown by induction on the rules for erasure), so $R_0' \sim_\ell R_1'$. Let P be a place with $\mathcal{L}(P) \sqsubseteq \ell$. By Lemma 10, either $t_0''\upharpoonright_P$ is a prefix of $t_1''\upharpoonright_P$ or $t_1''\upharpoonright_P$ is a prefix of $t_0''\upharpoonright_P$. Since $t_0''\upharpoonright_P = t_0\upharpoonright_P$ and $t_1''\upharpoonright_P = t_1\upharpoonright_P$, the theorem is proven. □

We now prove Theorem 1.

Proof of Theorem 1. Let P be a place such that $\mathcal{L}(P) \sqsubseteq \ell$. Take an initial configuration $(H_{init}; \omega; R; \epsilon; \langle P, s\rangle)$. If the set on the right side of the inclusion in Definition 5 is empty, the theorem is trivially satisfied. Otherwise, there exists an $\omega' \sim_\ell \omega$ such that $(\langle P, s\rangle, \omega', R)$ emits $t' \cdot \alpha'$ and $t'\upharpoonright_P = t\upharpoonright_P$. By Theorem 2, $(t \cdot \alpha)\upharpoonright_P$ is a prefix of $(t' \cdot \alpha')\upharpoonright_P$ or vice versa. If $\mathit{Place}(\alpha) = P$, then $(t \cdot \alpha)\upharpoonright_P$ and $(t' \cdot \alpha')\upharpoonright_P$ are the same length since $\mathit{Place}(\alpha') = P$ by definition and $t'\upharpoonright_P = t\upharpoonright_P$. Therefore, $(t \cdot \alpha)\upharpoonright_P = (t' \cdot \alpha')\upharpoonright_P$ and $\omega' \in k(\langle P, s\rangle, t \cdot \alpha, P, \ell)$. If $\mathit{Place}(\alpha) \neq P$, then $(t \cdot \alpha)\upharpoonright_P = t\upharpoonright_P = t'\upharpoonright_P$. Since $(\langle P, s\rangle, \omega', R)$ emits t', we have $\omega' \in k(\langle P, s\rangle, t \cdot \alpha, P, \ell)$. This shows the required inclusion. □