SX10: A Language for Parallel Programming with Information Security

A thesis presented by

Stefan Muller

to

Computer Science

in partial fulfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

March 30, 2012

Abstract

Many developers of concurrent programs would like to have strong security guarantees about their programs. One such guarantee is *noninterference*, the notion that high-security inputs to a program can have no influence on public outputs. Noninterference is commonly guaranteed through type systems which track information flow through a program. Unfortunately, many type systems for noninterference in concurrent programs require strict conditions, such as complete observational determinism, or track the security level of information at a fine granularity, making it difficult for programmers to reason about security policies.

In this thesis, we introduce SX10, a practical parallel language based on X10. X10 contains the concurrency abstraction of *places*, which allow computation and data to be separated across computation nodes. SX10 uses places as a security abstraction as well, with security policies specified at the granularity of places. Flows of information between places correspond to flows of information between security levels, making it easier for the type system to locate potential information leaks and determine what parts of a program must meet strong conditions. We suspect these features will also allow programmers to more easily specify security policies and write secure code. We describe a prototype implementation and two case studies which demonstrate uses of the language.

Acknowledgments

This thesis would not have been possible without the help of a number of people, all of whom I would like to thank here, including those I will inevitably forget to mention on this page.

It would be impossible to list in this space all of the reasons for which I thank my advisor, Steve Chong. Among many other reasons, I owe tremendous gratitude to you for your calm and patient advising, your help in pointing me toward resources and giving me suggestions when I was stuck at particularly tricky points, and for demonstrating to me how to get started extending Polyglot.

In a less immediate but no less real way, I could not have reached this point without the help of the numerous Computer Science faculty members who helped me rediscover my love of computing and provided me with the background without which I could not have dreamed of doing this work. In particular, I would like to thank Greg Morrisett and Dan Grossman for introducing me to the principles of programming languages.

Finally, thank you to my friends for keeping me sane throughout this process, especially Anna and Riva for somehow agreeing to read this thesis despite having work of their own to do, and, of course, to my family for providing the support that you always have.

Contents

Chapter 1

Introduction

Every day, computer systems are tasked with handling massive amounts of data. Some of this data is sensitive or secret, and some is not. Often, secret data is handled side-by-side with public data on the same system. For example, an online shopper submits a form containing his or her name, which may be public, and credit card number, which should be kept secret. Occasionally, more than two levels of security are involved. A user of a social networking website may, in rapid succession, enter a secret password, share a photograph with his or her friends and post a note for anyone to see. The security of a tremendous amount of private information depends on the ability to provide a strong guarantee that the data believed to be secret by the user of a system remains confidential. Providing such guarantees, however, proves to be difficult in many cases and has been the subject of a great deal of research.

One particular area of this general study of *information security* is *information flow*, which studies how information about high-security data within a program can subtly influence the results of computations on low-security data. If information flows in this way, an attacker can, simply by observing ostensibly public data and program outputs, receive information about secret data. Information can flow over explicit channels, such as direct assignment, or covert channels, such as information provided by the timing or power consumption of computations. Covert channels, which will be discussed in more detail in section 2.2, can be exceedingly difficult to find. In the setting of a concurrent program, the scheduling of running threads provides a new set of covert channels. Leaks of information through these channels are especially difficult to prevent, partially due to the non-determinism inherent in the scheduling of threads. Just as a bug in a concurrent program may manifest itself very rarely, an information leak may be tied to the most minute difference in scheduling behavior between one execution of a program and the next.

Despite this difficulty, it is important to provide strong security guarantees for concurrent programs, since concurrent programs are becoming increasingly common. More of the world's data, including websites, scientific data, and incoming requests on a server, are being processed by concurrent computations, and the feasibility of these computations requires a way to handle data securely without overly sacrificing throughput.

The contribution presented by this thesis is SX10, a language which allows for natural concurrent programming and enforces secure information flow. The syntax and semantics of SX10 are based on the X10 language [23], which will be described in fuller detail in section 2.1. The feature of the X10 language that makes it an appealing choice for this purpose is the notion of *places*. A place in X10 represents a node in a (potentially distributed) computation. A program can contain many threads operating concurrently at the same or different places. Since places may, at runtime, represent distinct machines, the language is designed such that synchronization between places is deliberate and rarely needed. In SX10, several places may be executing on a single physical machine, but the concept of places is used to separate computations operating at different security levels. Each place is assigned a security level, and is authorized to handle data at or below that security level. Since all data at a place has a certain security level, we do not need to consider covert channels within a place, leading to coarser-grained, more natural security policies. The discipline enforced by X10 regarding communication between places is used to ensure that it is clear when information may flow, either directly through data, or implicitly through control flow, from places at high levels to places at lower levels.

Chapter 2 introduces the problem of information flow and summarizes prior work in this area and analyses and frameworks on which our analysis is built. The remainder of the thesis presents our contribution, the SX10 language. Chapter 3 describes Featherweight SX10, or FSX10, a small subset of SX10 which will be used as the model for proving security. The chapter presents the security type system for FSX10 and proves its soundness. In Section 4.1, we describe the implementation of SX10 and its type system through modification of the X10 compiler. In Section 4.2, we present two case studies which model realistic distributed computations that can be programmed efficiently and securely in our language. We conclude in chapter 5.

Chapter 2

Background and Previous Work

This chapter describes the prior work that forms the background of this thesis. First, we present the X10 language, upon which our language, SX10, is based. We also describe an analysis for X10 programs unrelated to information flow which is important to the SX10 type system. The problem of information flow itself will be described in section 2.2, followed by a brief summary of previous work that has been done in understanding and limiting information flow.

2.1 The X10 Language

The X10 Language [23, 8, 24] was developed by IBM as part of the DARPA High Productivity Computer Systems program. The goal of the project was a language for writing high-performance parallel code. The creators of X10 envisioned the future of parallel computing to be simultaneous computation over large sets of data on multiple cores with non-uniform memory hierarchies. To facilitate such programming, X10 uses a Partitioned Global Address Space (PGAS) model, in which multiple processors share a common address space, but data is considered local to a particular process. A language based on a PGAS model may strictly enforce this locality or provide mechanisms for processors to access data that is local to other processors.

X10 makes these ideas explicit with the notion of *places*. All data in X10 is local to the place at which it was declared. Data may only be modified at that place, but data can be copied to other places using `at` statements and expressions. The statement `at P s`, where `P` is a place and `s` is a closure body, executes `s` at place `P`. X10 also provides expressions `at P e`, where `P` is a place and `e` is an expression to be evaluated at place `P`.

Any variables captured in the body of an `at` are serialized and copied to the place `P`. If these values are modified within `s`, only the copy is modified and these changes do not modify the copy at the other place. To permit computations to modify values at other places, X10 provides `GlobalRef` objects, which wrap a reference to an object at one place and can be passed to another place.

Running X10 programs consist of many *activities* running concurrently at a number of places. Activities are lighter-weight than threads and many may be running at a given place at a time. The implementation [11] uses a constant pool of threads, each of which maintains a queue of running activities. Each thread executes one of its currently running activities, and activities are redistributed among threads if necessary. In the current X10 implementation, all scheduling decisions are made individually at each place.

Activities may be created using the statement `async s`, which spawns a new activity to execute statement `s`. Activities can be joined using `finish s`, which waits for all activities spawned in `s` to complete before proceeding. The `async` and `finish` statements provide the main constructs for concurrency and synchronization in X10, and these are the constructs on which we will focus in the majority of this thesis. The language, however, also provides other constructs for concurrency such as clocks, futures and `atomic` blocks. Some of these features will be discussed briefly in Section 5.2.

2.1.1 May-Happen-In-Parallel Analysis for X10

The type system for SX10 assumes the existence of a static analysis tool to determine, for X10 programs, what statements may be executed in parallel. The results of this analysis are assumed to be available to the SX10 analysis when it begins analyzing a program. Such a may-happen-in-parallel analysis is presented by Lee and Palsberg [12], who model a subset of X10 which they call FX10 or Featherweight X10. Among other analyses on FX10, they present a static analysis that produces a set of pairs of program points that may happen in parallel. An FX10 program s is represented by a sequence of instructions i^p, where each instruction is annotated with a label p.[1] For a program s, the set $M(s)$ contains all pairs of labels (p_1, p_2) such that the instructions labeled with p_1 and p_2 may run simultaneously during execution of the program. For example, consider the following program.

$$a[0] :=^{p_1} 1;\ async^{p_2}\{a[1] :=^{p_3} 1;\ skip^{p_4}\}\ skip^{p_5}$$

For the program above, $M = \{(p_3, p_5), (p_4, p_5)\}$ (up to reordering of labels in a pair.) This is because p_3 and p_4 take place in the body of an *async*, so these instructions will be executed in a new activity running concurrently with the main activity executing p_5. The analysis is conservative but will, for most programs, give a fairly precise estimate of the instructions that may happen in parallel.

2.2 Information Flow

There are many ways in which information can flow through a program. The most direct is assignment. Assume we have two variables in a program, `l` and `h`. Variable `l` is assumed to contain low-security information and its value may be viewed by anyone. Variable

[1] FX10 uses different syntax than we use here. In particular, programs are notated p and program points are notated ℓ. To prevent confusion, we have used our notation here.

`h` is assumed to contain high-security information. We wish to prevent any information about the value of `h` from influencing the value of `l` at any point during the program. This property is referred to as *noninterference*. The following program clearly violates noninterference, as the value of `h` is directly transferred into `l`.

```
l := h
```

We can generalize this example to say that no expression which is assigned to `l` may contain `h`. This rule can extend to programs with more than two variables as well. Each variable is assigned a security level, low or high. No assignment to a low-security variable may involve a high-security variable. However, there are more subtle ways in which information about `h` can be transmitted to `l`. Consider the following program.

```
if (h = 0):
    l := 0
else:
    l := 1
```

In this program, while no assignment is made to `l` using an expression containing or derived from `h`, one bit of information about `h` is transferred to `l`, namely whether `h = 0`. Thus, this program violates noninterefererence. More complex programs can leak more information in similarly covert ways.

```
l := 0
while (l < h):
    l := l + 1
```

If `h` is a nonnegative integer at the start of the program, `l` will contain the value of `h` at the end of execution. The reason for this leak is that, while the expression assigned to `l` does not depend on `h`, the control flow of the program is dependent on the value of `h` and information is leaked to `l` through the fact that control flow has reached a certain assignment. Denning and Denning [10] call this type of leak an *implicit flow*. A solution to this problem is to track the level of information contained in the program counter. Within the body of a while loop or a branch of an if statement that depended on a high variable, the program counter depends on high-security information. At such locations, assignments to low-security variables are not allowed. Volpano, Smith and Irvine [28] showed a method of enforcing noninterference in the face of these types of leaks, and an equivalent version was presented by Sabelfeld and Myers [21].

Covert channels are, however, even more difficult to reason about in concurrent programs. Consider the following program which contains two threads running concurrently.

```
Thread 1        Thread 2
l := 0          while h > 0:
                    h := h - 1
                l := 1
```

The code in each of the two threads is secure according to the restrictions described so far. However, in this case, additional information is leaked to `l` by the actual timing of execution. The time at which the assignment `l := 1` is executed depends on the starting value of `h` and this, in turn, may affect which of the two threads assigns to `l` first. The final value of `l` is therefore dependent on the value of `h` and noninterference is not satisfied. This is indeed a security vulnerability: by observing only low-security outputs over multiple executions of the program, an attacker could fish out information about the high-security value `h`. Much of the remainder of this chapter is devoted to analyses for preventing such information leaks.

2.2.1 Security Lattices

Many applications require more complex security policies than the one described in the previous section. For example, some secrets fall into a hierarcy. If A, B and C are the sets of information that three users of a system are privileged to access, we may have $A \subset B \subset C$, that is, C may access anything that B may access and more, and B may access anything that A may access and more. Or, perhaps, if A, B and C are users of a web application, there may be some set of information that all three may access but each has another set of information that neither of the others may access.

The generalization of the two-level security policy to multiple levels and more complex relations is a *security lattice* [9]. A lattice consists of a set $\mathcal{L}$ of security levels together with the partial ordering $\sqsubseteq$, which is reflexive, transitive and antisymmetric. For any $\ell, \ell' \in \mathcal{L}$, ℓ and ℓ' have a least upper bound, denoted by $\ell \sqcup \ell'$. The scenario discussed in the previous section can be described by a two-point lattice, where $\mathcal{L} = \{L, H\}$, $L \sqsubseteq H, L \sqsubseteq L, H \sqsubseteq H, H \not\sqsubseteq L$. While we will, in the description of SX10 and the proof of the security condition, deal with these general lattices, most examples will involve the simple two-point lattice.

2.3 Types of Enforcement Methods

Several methods are used to enforce noninterference and other conditions on information flows. Some authors have used dynamic mechanisms to track the dependencies of data at runtime and halt the program if data that depends on high-security information is assigned to a low-security variable. Other authors use static methods, which analyze the text of a program for information flow. Most of these use a type system to reject programs that do not meet the security condition being studied. Still others use a hybrid between these two mechanisms.

This thesis will be concerned with static enforcement mechanisms. Denning and Denning [10] present a static process for certifying that the flow of information in a program is secure. More recent analysis methods, however, such as that of Volpano, Smith and Irvine [28], use a security type system, in which only secure programs can be typed using the rules of the type system. These are the types of analysis that will be the focus of this thesis.

In a security type system, type information may include standard types such as int, but type information also contains security levels of expressions and program variables. The version of Volpano et al.'s type system presented in [21], for example, has judgements of the following form:

$$\Gamma; \mathsf{pc} \vdash C$$

where C is a command, pc is the security level of the information contained in the program counter (i.e. an upper bound on the level of information used to decide whether to execute the command in question) and Γ maps program variables to their security levels (this is analogous to more standard type systems which include a mapping from program variables to their types). For example, the command $l := h$ would not be typable with $\Gamma(l) = L, \Gamma(h) = H$ and any value of pc. The command $l := c$, for some constant c, would be typable with the same Γ and $\mathsf{pc} = L$, but not with $\mathsf{pc} = H$, since this might transfer information to l through the control flow of the program.

Static enforcement mechanisms must necessarily be conservative. For example, the following program would not be typable under the type system described above, even though no information is leaked.

```
if h = 0:
    l := 1
else:
    l := 1
```

Incremental refinements could be made to allow programs such as this, but any sound type system will reject some secure programs.

2.4 Observational Determinism

2.4.1 Security Through Observational Determinism

Studying information flow in concurrent programs is not a new idea. This issue was notably addressed by Zdancewic and Myers [30] who developed a calculus called λ_{SEC}^{PAR}. This calculus is based on the join calculus and so uses different mechanisms for concurrency and communications between threads than X10 does. The main mechanism for communication and synchronization is message passing: threads may send messages to each other, and may synchronize execution by blocking until messages are received from one or more other threads. Nevertheless, many of their observations are applicable to our situation.

The main contribution of Zdancewic and Myers is that noninterference can be enforced in concurrent programs by requiring *observational determinism*. Consider two configurations m and m' which represent the state of a computation at particular points, including the computer memory, program text and program counter. Suppose m and m' are *low-equivalent* (notated $m \sim_\ell m'$), that is, an observer viewing only the low-security state of the program, cannot distinguish between the two configurations. Let $m \Downarrow T$ denote that when configuration m executes to completion, it produces the execution trace T. Observational determinism is given by the following condition.

$$(m \Downarrow T \wedge m' \Downarrow T') \Rightarrow T \sim_\ell T'$$

That is, if m and m' execute, a low observer cannot distinguish between the execution traces they produce. Observational determinism is a strong property, but since it necessarily requires that the low-observable behavior of a system cannot depend on differences between configurations that are not low-observable, it can be shown that observational determinism implies noninterference. Note that observational nondeterminism does not necessarily leak information. Consider the following multithreaded program executing threads 1 and 2 concurrently.

```
Thread 1          Thread 2


l := 0            l := 1
```

This program contains no high-security information to leak, and so should intuitively satisfy noninterference. However, the program is not observationally deterministic because, depending on the scheduling of the two threads, the final value of `l` may differ on two executions of this program. Thus, comparing two executions beginning with *identical* configurations might result in execution traces that are not low-equivalent.

To enforce observational determinism, the analysis assumes that the program satisfies *race freedom*, and that this has been previously verified by a separate analysis. Race freedom is defined by Zdancewic and Myers to mean that reads and writes to locations may be arbitrarily interleaved. This is a very strong definition of race freedom and leads to a guarantee stronger than observational determinism as defined above. The previous program, for example, is not race-free. Subsequent work, which will be described in Section 2.4.2, relaxes this restriction.

The work of Zdancewic and Myers builds on that of Roscoe [17], which argues for determinism of low-observable events as a mechanism for enforcing noninterference, and further distinguishes two types of nondeterminism: probabilistic nondeterminism and nondeterminism due to omissions from the model. A program may be probabilistically nondeterministic if parts of the computation are based on random (or pseudo-random) events, and thus the events are chosen from some probability distribution. Nondeterminism may also arise based on processes like thread scheduling, which may in fact be deterministic but are not included in the model and therefore must be considered nondeterministic.

Many other authors have developed type systems for ensuring observational determinism. For example, Deterministic Parallel Java (DPJ) [4, 5] is an extension to Java requiring that programs be deterministic by default, and any nondeterminism must be specified. The authors argue that such requirements will increase efficiency and correctness and are not unreasonable since most useful parallel algorithms are deterministic. The early work on DPJ does not mention applications to information flow, but mechanisms similar to theirs could be used to guarantee noninterference in a practical object-oriented language. We now, however, turn to a different type system for enforcing observational determinism.

2.4.2 Observational Determinism Through Fractional Capabilities

Terauchi [26] defines a type system that enforces observational determinism but is less restrictive than the type system described in Section 2.4.1. The language model used, like that of Zdancewic and Myers, involves message passing, which adds channels for communications between threads. This type system uses the idea of fractional capabilities [6, 27]: each thread has a capability with respect to each memory location or communication channel, which determines whether that thread can read or write on that location or channel. Capabilities are a widely used mechanism, and have been used by Birgisson, Russo and Sabelfeld [3] to prevent information flow by denying write capability when a write would be tainted with high information. Terauchi, in contrast, uses fractional capabilities in multithreaded programs to prevent data races.

Let γ range over abstract channels (for example, memory locations). Terauchi introduces Ψ, a mapping from memory locations to rationals in the range $[0, \infty)$ (though for the analysis it is sufficient to consider rationals in the range $[0, 1]$ and we will further simplify this later.) The essential idea is that the total capability for each location γ is conserved during the run of a program. That is, suppose that a thread spawns another thread using the statement *spawn* $s; s_1$. After the *spawn* instruction is executed, a new thread will execute s and the existing thread will execute s_1. If *spawn* $s; s_1$ is typable with Ψ, then s is typable with Ψ_1 and s_1 is typable with $\Psi - \Psi_1$ (where $(\Psi - \Psi_1)(\gamma) = \Psi(\gamma) - \Psi_1(\gamma)$ for all γ). Thus, the total fractional capability is constant as new threads are created. Race freedom can be ensured by allowing reads of γ only when $\Psi(\gamma) > 0$ and writes of γ only when $\Psi(\gamma) \geq 1$. Assuming the program starts with a single thread and $\Psi(\gamma) = 1$ for all γ, this restriction will prevent any accesses to γ while one thread is performing a write to γ.

This is only a part of the analysis, and this part does not in itself ensure observational determinism, since the values of variables is affected by factors other than the resolution of data races. Despite the limitations of this part of the analysis considered on its own, we use the notation of fractional capabilities to describe restrictions on memory accesses in parts of a program. Given a may-happen-in-parallel analysis, the Ψ notation may be separated from Terauchi's analysis by defining $\Psi(\gamma) = 0$ for program points that may happen in parallel with a write on γ, $\Psi(\gamma) = 0.5$ for program points that may happen in parallel with a read on γ and $\Psi(\gamma) = 1$ for all other program points.

2.5 Other Mechanisms for Noninterference

O'Neill, Clarkson and Chong [16] explicitly introduce observational nondeterminism into programs through the addition of a probabilistic choice operator, which executes one statement with probability p and another with probability $1 - p$. They use a type system to prevent *refinement attacks* in which information can be leaked through the resolution of nondeterministic choices during the execution of a program. In their model they use *refiners*, which "factor out" nondeterminism in programs by encapsulating all information about the nondeterministic decisions made in a particular execution of a program. The execution of a configuration together with a refiner thus deterministically produces an execution trace, and so one can compare the traces produced by two executions with the same refiner, for all refiners.

Other authors make different assumptions about scheduling. Smith and Volpano [25] use a language with a model of concurrency that is entirely nondeterministic: progress may be made on any thread at any step. They present the surprising result that while acheiving noninterference is fairly straightforward with these semantics, this task becomes more difficult when considering schedulers that are not purely deterministic. This shows the need for *scheduler-independent* models, which were formalized by Sabelfeld and Sands [22, 20]. The authors observe that when a known deterministic or probabilistic scheduler is used, knowledge of the scheduler can give an attacker high information. To model this situation, a scheduler σ is defined as a function from computation histories and current state to a scheduling decision. The papers present proof techniques for scheduler-dependent noninterference, where a fixed σ is a parameter to the model, and scheduler-independent noninterference, which holds over all possible schedulers σ.

Mantel and Sabelfeld [13] show a scheduler-independent security property in a multi-threaded while language (MWL). Russo and Sabelfeld [18] 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 [14] prove a security property for programs run under any scheduler in a class of robust schedulers. If threads are assigned security levels, under a robust scheduler, 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. Round-robin schedulers, for example, are robust. Barthe, Rezk, Russo and Sabelfeld[2] have developed a framework for security of low-level multi-threaded programs that allows programs to be written at a high level without knowledge of the scheduler. Mechanisms to interact with the scheduler and secure timing channels can be introduced during compilation.

Noninterference can also be achieved through the use of careful synchronization. Volpano and Smith [29], introduce a language in which multi-threaded programs may contain a protect statement which ensures that a statement is executed atomically. They show that, in their model, noninterference can be guaranteed if conditionals with high guards are wrapped in protect statements. Sabelfeld [19], instead adds semaphores to a concurrent language model and shows that this synchronization mechanism also allows synchronized concurrent programs to be written correctly and securely.

Chapter 3

The FSX10 Language

This chapter presents the FSX10 language, a small subset of the X10 language together with a type system which enforces noninterference between high and low computations. The syntax and semantics are based on the FX10 language [12]. We will use this simplified model to prove noninterference and then scale it up to SX10 in Section 4.1.

3.1 Syntax and Operational Semantics

3.1.1 Syntax and Language Features

Figure 3.1 shows the syntax for the FSX10 language. Figure 3.2 shows definitions that will be used later in the operational semantics and type system.

This model consists of a limited set of X10's language features, but this set includes most of the core features of the language and many other useful language features can be defined in terms of the features provided in FSX10. Some aspects of the syntax merit additional explanation. We have, as suggested by Lee and Palsberg, extended FX10 with places, and the type system with the ability to determine whether two program points may happen in parallel *at the same place*. FX10 also lacks other synchronization mechanisms present in X10, such as clocks. We, like Lee and Palsberg, leave such additions to future

Places			
P	$\in$	$Places$	
Program Points			
p	$\in$	$ProgramPoints$	
Variables			
x	$\in$	$Vars$	
References			
r	$\in$	$GlobalRefs$	
Expressions			
e	$::=$	c	Constant
	$\vert$	x	Variable
	$\vert$	$!r$	Dereference
	$\vert$	$e \oplus e$	Binary operation
Statements			
s	$::=$	skip^p	No-op
	$\vert$	$i; s$	Sequence
Instructions			
i	$::=$	skip^p	No-op
	$\vert$	$r :=^p e$	Store
	$\vert$	$\mathsf{finish}^p\ s$	Finish
	$\vert$	$\mathsf{async}^p\ s$	Asynchronous
	$\vert$	$\mathsf{if}^p\ e\ \mathsf{then}\ s\ \mathsf{else}\ s$	Conditional
	$\vert$	$\mathsf{while}^p\ e\ \mathsf{do}\ s$	Iteration
	$\vert$	$\mathsf{backat}^p\ P$	Back at
	$\vert$	$\mathsf{output}^p\ e$	Output
	$\vert$	$\mathsf{input}^p\ r$	Input
	$\vert$	$\mathsf{let}^p\ x = e\ \mathsf{in}\ s$	Let
Trees			
T	$::=$	$\langle P, s\rangle$	Located statement
	$\vert$	$T \triangleright \langle P, s\rangle$	Sequence
	$\vert$	$T \Vert T$	Parallel
	$\vert$	$\surd$	Done
Traces			
t	$::=$	ϵ	Empty
	$\vert$	$t \cdot i(c, P)$	Input event
	$\vert$	$t \cdot o(c, P)$	Output event
	$\vert$	$t \cdot w(c, r)$	Store event

Figure 3.1: Syntax for FSX10

work. Method calls, present in FX10, are omitted from FSX10 since these are fairly straightforward to add to a security analysis. Both models omit objects.

We also adopt several of the interactive language features of O'Neill, Clarkson and

Define tree replacement as follows:
$T[T \mapsto T'] = T'$
$(T_1 \triangleright T_2)[T \mapsto T'] = T_1[T \mapsto T'] \triangleright T_2[T \mapsto T']$
$(T_1 \| T_2)[T \mapsto T'] = T_1[T \mapsto T'] \| T_2[T \mapsto T']$
$T_1[T \mapsto T'] = T_1$ (if $T \notin T_1$)

$\mathit{PointsRunning}(P, T) = \{p \mid \langle P, i^p \rangle \in T\}$
$\mathit{PointsWaiting}(P, T) = \{p \mid \langle P', s_1.\mathsf{backat}\ P; i^p; s_2 \rangle \in T\} \cup \mathit{PointsRunning}(P, T)$
These may be written without the tree if the tree is implied.

Define statement composition recursively as follows:
$\mathsf{skip}^p.s = \mathsf{skip}^p; s$
$(i; s_1).s_2 = i; (s_1.s_2)$

Program point annotations for statements:
We may write s^p if $s = i^p; s'$.

Figure 3.2: Helper Functions

Chong [16], including input and output instructions, events and input strategies. In the context of FSX10, we conflate input and output channels with places, assuming that each place will have access to a distinct console with input and output channels. Thus, it is not necessary to specify the channel in an input or output instruction. The channel is the place at which the instruction executes. An event α records an event observable by an attacker, considered to be inputs, outputs and assignments to references. An output of value c at place P is recorded with $o(c, P)$, an input of value c at place P is recorded with $i(c, P)$ and an assignment of value c to reference r is recorded with $w(c, r)$. A trace t consists of an ordered list of events. We may restrict a trace to events at a certain place P with the notation $t\restriction_P$. This includes inputs and outputs on P as well as writes to a reference defined at P. As a configuration executes, it produces a trace. We say that configuration m produces the trace t, denoted $m \rightsquigarrow t$ if there exists an $m' = (H'; \omega; R'; t; T')$ such that $m \rightarrow^* m'$ (note that m' contains trace t). Input strategies ω [16] model the behavior of a user providing input to the program at a certain place. We assume that this user may have seen any observable events at that place up to the current point, and thus ω is a function of both the place and the current execution trace restricted to that place.

3.1.2 Scheduling

Since the execution of the program depends on the scheduling of threads, it is important for the model to contain a model of the scheduler. We use the concept of *refiners* [16], which can represent any sequence of decisions made by the scheduler. In keeping with the typical implementation of X10 [11], different places are assumed to be distinct computation nodes with distinct schedulers, running concurrently. However, to allow the model to use conventional semantics, we assume that places take turns executing instructions, in an order specified by the refiner. When it is a place's turn to run, its scheduling decision depends only on the set of activities currently executing at that place.

We model this type of scheduler with a refiner $R = (Ps, Sch)$ consisting of two parts. The first is a stream of places, Ps. The scheduler Sch is a function from places to streams chs of scheduling functions. These scheduling lists represent the schedulers at each place. When a place is selected, the first function, ch, is removed from that place's scheduling list. Scheduling functions ch are functions which accept a set of program points p and return one element of the set. Refiners of this kind are quite versatile and can be used to represent arbitrary behavior of a scheduler.

3.1.3 Operational Semantics

$$\textsc{Const}\ \frac{}{P; H; c \Downarrow c} \qquad \textsc{Ref}\ \frac{\mathit{PlaceOfLoc}(r) = P}{P; H; !r \Downarrow H(r)} \qquad \textsc{Op}\ \frac{P; H; e_1 \Downarrow c_1 \quad P; H; e_2 \Downarrow c_2}{P; H; e_1 \oplus e_2 \Downarrow c_1 \oplus c_2}$$

Figure 3.3: Large-step expression semantics for FSX10

Figures 3.3 through 3.5 provide the operational semantics for the language. Figure 3.3 gives the large-step operational semantic rules of the form $P; H; e \Downarrow e$, where P is a place, H is a heap, defined as a function from references to integer constants, and e is an expression. Heaps map references to values c. The place of a reference is assumed to be available using

$$\textsc{Step}\quad \frac{(H;\omega;R;t;T_1) \rightarrow (H';\omega;R';t';T_1')}{(H;\omega;R;t;T_1 \triangleright T_2) \rightarrow (H';\omega;R';t';T_1' \triangleright T_2)}$$

$$\textsc{Left}\quad \frac{(H;\omega;R;t;T_1) \rightarrow (H';\omega;R';t';T_1')}{(H;\omega;R;t;T_1 \parallel T_2) \rightarrow (H';\omega;R';t';T_1' \parallel T_2)}$$

$$\textsc{Right}\quad \frac{(H;\omega;R;t;T_2) \rightarrow (H';\omega;R';t';T_2')}{(H;\omega;R;t;T_1 \parallel T_2) \rightarrow (H';\omega;R';t';T_1 \parallel T_2')}$$

$$\textsc{Seq}\quad \frac{}{(H;\omega;R;t;\surd \triangleright T) \rightarrow (H;\omega;R;t;T)}$$

$$\textsc{DoneL}\quad \frac{}{(H;\omega;R;t;\surd \| T) \rightarrow (H;\omega;R;t;T)}$$

$$\textsc{DoneR}\quad \frac{}{(H;\omega;R;t;T \| \surd) \rightarrow (H;\omega;R;t;T)}$$

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

$$\textsc{Stmt}\quad \frac{\begin{array}{c}(H;\omega;t;\langle P, s^p\rangle) \rightarrow (H';\omega;t';T') \\ \mathit{PointsRunning}(P) \neq \emptyset \quad Sch(P) = chs \cdot ch \quad ch(\mathit{PointsRunning}(P)) = p\end{array}}{(H;\omega;(Ps \cdot P, Sch);t;\langle P, s^p\rangle) \rightarrow (H';\omega;(Ps, Sch[P \rightarrow chs]);t';T')}$$

Figure 3.4: Operational semantics on trees

the helper function *PlaceOfLoc*() before the program is run and does not change. Program execution starts with the empty heap H_{init}, which maps each reference to a default value (such as 0). These rules are straightforward. Rule Const evaluates constants. Rule Ref evaluates references to their value in the heap. Note from the condition of Ref that, as with `GlobalRef` objects in X10, references in SX10 may only be assigned at the place at which they are defined, although in X10 this condition is enforced by the type system and not at runtime. Rule Op recursively evaluates binary operations represented by $\oplus$.

Figure 3.4 gives the small-step operational semantic rules of the form $(H;\omega;t;R;T) \rightarrow (H';\omega;t';R';T')$, where H and H' are heaps, ω is an input strategy, t and t' are traces, R and R' are refiners and T and T' are program trees. Rule Step recursively evaluates the current tree T_1 of a sequence structure $T_1 \triangleright T_2$ (these are generated by finish). Rules Left and Right recursively evaluate one branch of a parallel structure $T_1 \| T_2$, generated by async. Rules Seq, DoneL and DoneR remove subtrees that have finished execution, notated $\surd$, from sequence and parallel trees. The two rules that interact with the scheduler are

Skip1

$$\frac{}{(H;\omega;t;\langle P,\mathsf{skip}\rangle)\rightarrow(H;\omega;t;\surd)}$$

Skip2

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

Assign

$$\frac{\begin{array}{c}P;H;e\Downarrow c\\ \mathit{PlaceOfLoc}(r)=P\end{array}}{(H;\omega;t;\langle P,r:=e;s\rangle)\rightarrow(H[r\mapsto c];\omega;t\cdot w(c,r);\langle P,s\rangle)}$$

Let

$$\frac{P;H;e\Downarrow c}{(H;\omega;t;\langle P,\mathsf{let}\ x=e\ \mathsf{in}\ s_1;s_2\rangle)\rightarrow(H;\omega;t;\langle P,s_1[c/x].s_2\rangle)}$$

Async

$$\frac{}{(H;\omega;t;\langle P,\mathsf{async}\ s_1;s_2\rangle)\rightarrow(H;\omega;t;\langle P,s_1\rangle\parallel\langle P,s_2\rangle)}$$

Finish

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

AtStmt

$$\frac{}{(H;\omega;t;\langle P_1,\mathsf{at}\ P_2\ s_1;s_2\rangle)\rightarrow(H;\omega;t;\langle P_2,s_1.\mathsf{backat}\ P_1;s_2\rangle)}$$

If1

$$\frac{P;H;e\Downarrow c}{(H;\omega;t;\langle P,\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2;s_3\rangle)\rightarrow(H;\omega;t;\langle P,s_1.s_3\rangle)}\ c>0$$

If2

$$\frac{P;H;e\Downarrow c}{(H;\omega;t;\langle P,\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2;s_3\rangle)\rightarrow(H;\omega;t;\langle P,s_2.s_3\rangle)}\ c\leq 0$$

Out

$$\frac{P;H;e\Downarrow c}{(H;\omega;t;\langle P,\mathsf{output}\ e;s\rangle)\rightarrow(H;\omega;t\cdot o(c,P);\langle P,s\rangle)}$$

In

$$\frac{\begin{array}{c}\omega(P,t\restriction_P)=c\\ \mathit{PlaceOfLoc}(r)=P\end{array}}{(H;\omega;t;\langle P,\mathsf{input}\ r;s\rangle)\rightarrow(H[r\mapsto c];\omega;t\cdot i(c,P);\langle P,s\rangle)}$$

BackAt

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

While

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

Figure 3.5: Operational semantics for statements

IDLE and STMT. At each step in a computation, the first place in Ps, call it P, is removed. Rule IDLE applies if no threads are currently executing at P (i.e. $\mathit{PointsRunning}(P) = \emptyset$), and it has the effect of simply removing that place from Ps. If there are threads currently executing at P, the first function, ch, is removed from P's scheduling list, $Sch(P)$. $ch(\mathit{PointsRunning}(P))$ selects a program point $p \in \mathit{PointsRunning}(P)$ and Rule STMT then allows the statement starting with that program point, s^p, to reduce.

The rules in 3.5 are for judgments of the form $(H; \omega; t; \langle P, s\rangle) \to (H'; \omega; t'; T')$, where H, ω and t are as before and $\langle P, s\rangle$ is a statement executing at a place P. These rules are invoked through the rule STMT. Note that refiners need not be included in these statements, since scheduling has already been handled by STMT. Rule SKIP1 reduces a terminal skip command to the tree $\surd$, indicating that the tree has finished executing. In contrast, rule SKIP2 removes a nonterminal skip command. These could be generated, for example, when statements in branches of an if statement, which end with skip, are composed with the remaining statement. Rule ASSIGN evaluates e to a constant, and assigns it to the reference r, producing a write event. Again, mutable references can be assigned only at the place at which they are defined. Rule LET evaluates e and substitutes it for x in s_1. Statements s_1 and s_2 are composed, but no substitution is performed in s_2.

The next three rules are based on features of X10. ASYNC creates a parallel tree structure, so that $\langle P, s_1\rangle$ and $\langle P, s_2\rangle$ will be executed concurrently. FINISH creates a sequence tree structure, so that $\langle P, s_1\rangle$ (including any parallel subtrees later created by async commands in s_1) will be executed before $\langle P, s_2\rangle$. ATSTMT changes the place at which a statement executes, hence P_2 is the place of the resulting statement. We add the keyword backat to mark the point at which control returns to the original place, between s_1 and s_2. The keyword backat should not appear in source programs.

Rules IF1 and IF2 evaluate e and select a branch, either s_1 or s_2 depending on whether the value of e is positive or nonpositive. The chosen branch is composed with the remainder of the statement. Rule OUTPUT evaluates e and outputs it over the console at the current place, producing an output event. Rule INPUT inputs a value c over the console at the current place, which is specified by the input strategy ω over the current place and the trace of events at that place, and assigns it to the reference r. Rule BACKAT returns to an original place after an `at` statement, as described above. Finally, rule WHILE executes a `while` loop. The loop is unrolled to an `if` statement, which does nothing if e evaluates to a nonpositive constant and executes the body, followed by the next iteration of the while loop otherwise.

3.2 Type System

We begin with an overview of the security type system for FSX10. The main feature of the type system is that places are used to separate computations occurring at different security levels. We introduce a function $\mathit{PlaceLevel}()$ which maps each place to its security level ℓ in the lattice of security levels $\mathcal{L}$. All computation at P is assumed to be tainted with information at level at least $\mathit{PlaceLevel}(P)$. This means that, within a block of code executing at one place, we need not consider either explicit or implicit flows of information.

However, when `at` statements are used to transition between places, both explicit and implicit flows are possible. Fortunately, the syntax of X10 makes it clear where these flows between places can occur, and so the language itself does some of the work of the type system. Furthermore, existing X10 programming conventions encourage careful use of `at` to make sure that no more data than necessary is copied, since the operations of serializing, transmitting and deserializing are quite expensive. These same conventions encourage secure programming in SX10. The first restriction of the type system is that for a statement `at` $P_2\ s_1; s_2$ executing at place P_1, we must have $\mathit{PlaceLevel}(P_1) \sqsubseteq \mathit{PlaceLevel}(P_2)$. This is important since the values of local variables are transmitted to P_2 when this instruction is executed, so allowing an `at` statement to move to a lower or unrelated place would allow an explicit information flow, as well as the potential for an implicit flow of scheduling information from P_1 to P_2.

For this reason, `at` expressions, present in X10, are not included in FSX10 or SX10. Since the expression `at` $P_2\ x$, executed at P_1, transfers data from P_2 to P_1, it must be the case that $\mathit{PlaceLevel}(P_2) \sqsubseteq \mathit{PlaceLevel}(P_1)$. However, the execution of this computation at P_2 covertly transfers information from P_1 to P_2 as well. For example, if the `at` expression is contained in a branch of an `if` statement, the occurrence of the computation at P_2 transfers information about the value of the conditional. Therefore, `at` expressions are secure only in very limited contexts. SX10 could be extended to allow `at` expressions in these contexts, such as between places of equal security levels or outside of control structures. However, in these cases, equivalent behavior can also be achieved using `at` statements if desired.

These restrictions prevent flows of information explicitly and implicitly through control flow, but do not prevent leaks through timing channels. For example, consider the following program, which begins executing at low-security place `L`.

```
async {
...
    l = 0;
}
at (H) {
    ...
}
l = 1;
```

If the computation at high-security place `H` is short, it is likely that `l` will be assigned the value 1 before it is assigned the value 0, and thus the value of `l` at the end of the program will be 0. If the computation at place `H` is long, the opposite scenario is likely. This is an example of timing information flowing from a high place to a low place. The assignments to `l`, even though they occur at a low-security place, are tainted with high-security information based on how long the computation at place `H` takes to run. Thus, we introduce Δ, a function from program points p to security levels, which captures the fact that certain instructions may be tainted with higher-security information than the level of the place where they execute. Note that if p executes at place P, $\mathit{PlaceLevel}(P) \sqsubseteq \Delta(p)$. This means that the statements $\Delta(p) \neq \mathit{PlaceLevel}(P)$ and $\Delta(p) \not\sqsubseteq \mathit{PlaceLevel}(P)$ are equivalent.

To be more specific, whenever an instruction $\texttt{at}^p\ P_2\ s_1$, executed at place P_1, returns, timing information flows from P_2 to any activities currently running on P_1, that is, any program point that may happen in parallel with p, as well as the statement after the `at`. We therefore constrain Δ of each of these program points to be no lower than Δ of the last instruction in s_1 (which will be at least $\mathit{PlaceLevel}(P_2)$ but may be higher). This part of the type system requires a may-happen-in-parallel analysis, such as that of Lee and Palsberg [12]. We assume that one has been run on the program beforehand. Consider the following program, which again begins executing at low place `L`:

```
finish {
    async {
        l = 0;
    }
    l = 1;
    at (H) {
     ...
    }
    l = 2;
}
async {
    l = 3;
}
l = 4;
```

As in the previous example, the assignments `l = 0` and `l = 2` are tainted with high-security information, since the latter occurs after a return from a high-security place, and the former occurs in parallel with this return. The assignment `l = 1`, however, must occur before the `at` and is therefore low-security.

If our model considered absolute time, it must view the assignments `l = 3` and `l = 4` as high-security. However, the only ways in which this could leak information would be if an attacker were timing each instruction or if the scheduler could measure absolute time and resolve nondeterminism accordingly. We assume that attackers do not have clocks and that the scheduler at each place knows only how many instructions have been executed at that place, not how long they took in real time. The same number of instructions will be executed at `L` no matter what computation occurs at `H`. Thus, we consider the timing information to have been removed when all tainted threads finished, and so these assignments are untainted. The astute reader may notice that if a program beginning at P_1 consists only of the statement `at` P_2 $s_1; s_2$, the statement s_2 should not be tainted with information from P_2 since, in this case, the scheduler at place P_1 remains dormant during the `at` and so timing information could not be leaked to it. This is true, but is not considered by our model for the simplicity of the type system and the accompanying proof.

These rules mark the program points that are tainted with information at a security level higher than that of their place. However, the question remains of what to do with these markings. A sound type system could disallow any program containing program points with a Δ value higher than the level of their place, but such a type system would be overly restrictive. It would also be sound to disallow any instructions that produce events (i.e. any inputs, outputs or assignments) to be marked with a security level higher than their place. However, we show that it is sufficient to enforce observational determinism at program points that are marked with high security levels. That is, if program points p and p' produce events involving the same channel (where a channel may be a place or a reference) and have Δ levels higher than the level of their place, their ordering must be deterministic. We enforce this by using the may-happen-in-parallel analysis to detect

$$\textsc{Const}\quad \frac{}{p; \Gamma; \Psi; \Delta \vdash c}$$

$$\textsc{Val}\quad \frac{\Gamma(x) = \mathit{PlaceOfPt}(p)}{p; \Gamma; \Psi; \Delta \vdash x}$$

$$\textsc{RefDET}\quad \frac{\begin{array}{c}\Delta(p) \neq \mathit{PlaceLevel}(\mathit{PlaceOfPt}(p)) \\ \Psi(p, r) > 0\end{array}}{p; \Gamma; \Psi; \Delta \vdash !r}$$

$$\textsc{RefND}\quad \frac{\Delta(p) = \mathit{PlaceLevel}(\mathit{PlaceOfPt}(p))}{p; \Gamma; \Psi; \Delta \vdash !r}$$

$$\textsc{Op}\quad \frac{\begin{array}{c}p; \Gamma; \Psi; \Delta \vdash e_1 \\ p; \Gamma; \Psi; \Delta \vdash e_2\end{array}}{p; \Gamma; \Psi; \Delta \vdash e_1 \oplus e_2}$$

Figure 3.6: Expression Typing Judgments for FSX10

$$\textsc{TreeStmt}\quad \frac{P; \Gamma; \Psi; \Delta \vdash s^p : \ell}{\Psi; \Delta \vdash \langle P, s \rangle}$$

$$\textsc{TreePara}\quad \frac{\begin{array}{c}\Psi; \Delta \vdash T_1 \\ \Psi; \Delta \vdash T_2\end{array}}{\Psi; \Delta \vdash T_1 \| T_2}$$

$$\textsc{TreeJoin}\quad \frac{\begin{array}{c}\Psi; \Delta \vdash T_1 \\ \Psi; \Delta \vdash \langle P, s^p \rangle \\ \forall p' \in MHP(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{\Psi; \Delta \vdash T_1 \triangleright \langle P, s^p \rangle}$$

$$\textsc{TreeDone}\quad \frac{}{\Psi; \Delta \vdash \surd}$$

Figure 3.7: Tree Typing Judgments for FSX10

concurrent writes or concurrent reads and writes to the same channel. This is propagated through the type system using the notation of Terauchi's fractional capabilities. Note that we need not consider a case in which only one of these points is high-security, since the type system propagates information across concurrently executing activities. In the example above, assignments `l = 0` and `l = 2` do not meet this condition, since they are tainted with information from `H`. Assignments `l = 3` and `l = 4`, however, meet this condition and are secure.

The typing rules assume a number of definitions and pre-existing constraints, which will be described here. First, there are two helper functions having to do with the may-happen-in-parallel analysis that is assumed to have been run on the program.

- $MHP(p, p')$ is true if and only if program points p and p' may happen in parallel at the same place
- $MHP(p) = \{p' \mid MHP(p, p')\}$

Async
$$\frac{\begin{array}{c}P;\Gamma;\Psi;\Delta \vdash s_1 : \ell_1\\ P;\Gamma;\Psi;\Delta \vdash s_2 : \ell_2\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \Delta(p) \sqsubseteq \Delta(p_2)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash \mathsf{async}^p\ s_1^{p_1}; s_2^{p_2} : \ell_2}$$

Finish
$$\frac{\begin{array}{c}P;\Gamma;\Psi;\Delta \vdash s_1 : \ell_1\\ P;\Gamma;\Psi;\Delta \vdash s_2 : \ell_2\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \forall p' \in MHP(p), \Delta(p') \sqsubseteq \Delta(p_2)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash \mathsf{finish}^p\ s_1^{p_1}; s_2^{p_2} : \ell_2}$$

If
$$\frac{\begin{array}{c}P;\Gamma;\Psi;\Delta \vdash s_1 : \ell_1\\ P;\Gamma;\Psi;\Delta \vdash s_2 : \ell_2\\ P;\Gamma;\Psi;\Delta \vdash s_3 : \ell_3\\ p;\Gamma;\Psi;\Delta \vdash e\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \Delta(p) \sqsubseteq \Delta(p_2)\\ \ell_1 \sqcup \ell_2 \sqsubseteq \Delta(p_3)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash \mathsf{if}^p\ e\ \mathsf{then}\ s_1^{p_1}\ \mathsf{else}\ s_2^{p_2}; s_3^{p_3} : \ell_3}$$

AtStmt
$$\frac{\begin{array}{c}P_2;\Gamma;\Psi;\Delta \vdash s_1 : \ell_1\\ P_1;\Gamma;\Psi;\Delta \vdash s_2 : \ell_2\\ \Delta(p) \sqsubseteq \mathit{PlaceLevel}(P_2)\\ \mathit{PlaceLevel}(P_2) \sqsubseteq \Delta(p_1)\\ \ell_1 \sqsubseteq \Delta(p_2)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P_1;\Gamma;\Psi;\Delta \vdash \mathsf{at}^p\ P_2\ s_1^{p_1}; s_2^{p_2} : \ell_2}$$

While
$$\frac{\begin{array}{c}P;\Gamma;\Psi;\Delta \vdash s_1 : \ell_1\\ P;\Gamma;\Psi;\Delta \vdash s_2 : \ell_2\\ p;\Gamma;\Psi;\Delta \vdash e\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \ell_1 \sqsubseteq \Delta(p)\\ \ell_1 \sqsubseteq \Delta(p_2)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash \mathsf{while}^p\ e\ \mathsf{do}\ s_1^{p_1}; s_2^{p_2} : \ell_2}$$

Let
$$\frac{\begin{array}{c}P;\Gamma[x \mapsto P];\Psi;\Delta \vdash s_1 : \ell_1\\ P;\Gamma;\Psi;\Delta \vdash s_2 : \ell_2\\ p;\Gamma;\Psi;\Delta \vdash e\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \ell_1 \sqsubseteq \Delta(p_2)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash \mathsf{let}^p\ x = e\ \mathsf{in}\ s_1^{p_1}; s_2^{p_2} : \ell_2}$$

BackAt
$$\frac{\begin{array}{c}P_2;\Gamma;\Psi;\Delta \vdash s : \ell\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P_1;\Gamma;\Psi;\Delta \vdash \mathsf{backat}^p\ P_2; s^{p_1} : \ell}$$

Skip1
$$\frac{\forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)}{P;\Gamma;\Psi;\Delta \vdash \mathsf{skip}^p : \Delta(p)}$$

Skip2
$$\frac{\begin{array}{c}P;\Gamma;\Psi;\Delta \vdash s : \ell\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash \mathsf{skip}^p; s^{p_1} : \ell}$$

AssignDET
$$\frac{\begin{array}{c}P;\Gamma;\Psi;\Delta \vdash s : \ell\\ p;\Gamma;\Psi;\Delta \vdash e\\ \Delta(p) \neq \mathit{PlaceLevel}(P)\\ \Psi(p, r) \geq 1\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash r :=^p e; s^{p_1} : \ell}$$

AssignND
$$\frac{\begin{array}{c}P;\Gamma;\Psi;\Delta \vdash s : \ell\\ p;\Gamma;\Psi;\Delta \vdash e\\ \Delta(p) = \mathit{PlaceLevel}(P)\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash r :=^p e; s^{p_1} : \ell}$$

OutDET
$$\frac{\begin{array}{c}P;\Gamma;\Psi;\Delta \vdash s : \ell\\ p;\Gamma;\Psi;\Delta \vdash e\\ \Delta(p) \neq \mathit{PlaceLevel}(P)\\ \Psi(p, P) \geq 1\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash \mathsf{output}^p\ e; s^{p_1} : \ell}$$

InDET
$$\frac{\begin{array}{c}P;\Gamma;\Psi;\Delta \vdash s : \ell\\ \Delta(p) \neq \mathit{PlaceLevel}(P)\\ \Psi(p, r) \geq 1\\ \Psi(p, P) \geq 1\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash \mathsf{input}^p\ r; s^{p_1} : \ell}$$

OutND
$$\frac{\begin{array}{c}P;\Gamma;\Psi;\Delta \vdash s : \ell\\ p;\Gamma;\Psi;\Delta \vdash e\\ \Delta(p) = \mathit{PlaceLevel}(P)\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash \mathsf{output}^p\ e; s^{p_1} : \ell}$$

InND
$$\frac{\begin{array}{c}P;\Gamma;\Psi;\Delta \vdash s : \ell\\ \Delta(p) = \mathit{PlaceLevel}(P)\\ \Delta(p) \sqsubseteq \Delta(p_1)\\ \forall p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)\end{array}}{P;\Gamma;\Psi;\Delta \vdash \mathsf{input}^p\ r; s^{p_1} : \ell}$$

Figure 3.8: Statement Typing Judgments for FSX10

- $M(p) = \{p' \mid MHP(p,p') \text{ and } \mathsf{at}\ P'\ s_1; s_2^{p'} \in T \text{ or } \mathsf{backat}\ P; s^{p'} \in T\}$

Intuitively, $M(p)$ is the set of program points occurring in parallel with p at which control may return from another place. This is important for the type system since these points will be tainted with information from that place.

We also introduce a capability map Ψ, similar to the fractional capability of Terauchi [26]. This function maps abstract channels (we will use the term *channels* to refer to both places and references) and program points to rational numbers in the range $[0, 1]$. As noted in Section 2.4.2, however, we do not implement Terauchi's analysis directly. Instead, we assume that Ψ is initialized at the start of the analysis so that the following conditions are met for each channel γ and each program point p.

$$\text{If } MHP(p,p') \text{ and } p' \text{ writes on channel } \gamma,\ \Psi(p,\gamma) = 0.$$
$$\text{If } MHP(p,p') \text{ and } p' \text{ reads from channel } \gamma,\ 0 < \Psi(p,\gamma) < 1.$$

The type system for FSX10 is shown in Figures 3.6 through 3.8. Figure 3.6 shows rules for the typing judgment $p; \Gamma; \Delta; \Psi \vdash e$, where p is the program point at which the expression is computed, Γ is a function mapping immutable variables to the places at which they are defined, Δ is a function from program points to security levels, Ψ is the fractional capability function described above, and e is an expression. Rule Const, which types constants, is straightforward. Rule Val enforces that variables are accessed only at the place at which they are defined. There are two rules for typing dereferences. Rule RefND applies to a program point p executing at place P only if $\Delta(p) = \mathit{PlaceLevel}(P)$, i.e. the scheduling of threads at this point has not been influenced by computation at a security level higher than that of P. At these points, observational determinism is not required and so no constraints on the capabilities Ψ are enforced. Rule RefDET applies if $\Delta(p) \neq \mathit{PlaceLevel}(P)$, which indicates that this computation is tainted by high-security information and therefore observational determinism is required. Thus, these typing rules enforce that $\Psi(p, \gamma) > 0$. Rule Op recursively types both subexpressions of a binary operation.

Figure 3.7 shows rules for the typing judgment $\Psi; \Delta \vdash T$, where Ψ and Δ are as above and T is a program tree. Rule TreeStmt states that a tree $\langle P, s \rangle$ can be typed if the statement s can be typed with rules in Figure 3.8. Rule TreePara ensures that parallel trees can be typed if the two subtrees can be typed. Rule TreeJoin requires that subtrees can be typed, and also requires that, after control of a thread elevates to a high place, computation at the lower place will be tainted with high information as long as there is another computation that may happen in parallel at the same low place, i.e. until all threads at that place join. Rule TreeDone allows trees $\surd$ to trivially be typed.

Finally, 3.8 shows rules for the typing judgment $P; \Gamma; \Psi; \Delta \vdash s : \ell$, where Γ, Ψ and Δ are as above, s is a statement running at place P and ℓ is the security level of the last instruction in s. Except where noted, every typing rule enforces several common conditions on a statement $i^p; s^{p'}$. First, s must be typable, and the post-level of $i; s$ is the post-level of s. Second, the Δ level of p' must be at least as high as $\Delta(p)$, and in general Δ of instructions in a statement must monotonically increase throughout execution. Third, $\Delta(p)$ must be at least as high as the level of any `backat` instruction occurring in parallel with p. This ensures that the type system tracks any timing information that might flow from a higher place while p is running.

Rule Async ensures that both branches are typable. Since s_1 and s_2 are executed concurrently, no relation is needed between p_1 and p_2. The post-level of the statement can be either ℓ_1 or ℓ_2 since the third condition above will propagate levels between s_1 and s_2. Rule Finish, in addition to typing both substatements, requires a condition similar to that on TreeJoin, explained above. Rule AtStmt requires that $\Delta(p)$ be protected by the level of P_2, to prevent explicit flows. For similar reasons, the level of P_2 must be protected by $\Delta(p_1)$, which ensures that s_2 will be marked as tainted with information from P_2. Rule If conservatively requires that $\Delta(p_3)$ protect both l_1 and l_2 since it is not known at compile time which branch will be taken. Rule WHILE requires that the level of s_1 is equal to its post-level, since s_1 may follow itself. Rule LET types the expression e, assumed to be evaluated at program point p. Variable x is added to $\Gamma()$ in typing s_1. Rule BACKAT types s_1 at place P_2. Rules SKIP1 and SKIP2 are straightforward. Like the rules for dereference, there are two rules for typing assignments and input and output statements. As above, the rules ending in DET enforce appropriate constraints on the capabilities: $\Psi(p, \gamma) > 0$ to read on γ, and $\Psi(p, \gamma) \geq 1$ to write on γ. Note that executing the command input r at place P requires write capability on both r and P. Write capability is required on r because storing an input value to r involves an implicit assignment, and write capability on P is required because the use of input strategies means that the ordering of two console input prompts affects the result of computation.

3.3 Security Condition

We will show a security condition based on attacker knowledge, similar to that of Askarov and Sabelfeld [1]. Intuitively, attacker knowledge, k, is the set of input strategies that could have resulted in the low-observable events present in a trace. The smaller the set, the more the attacker knows about the input strategy. Progress knowledge, k^+, is the set of input strategies that could have resulted in a low-equivalent trace *and* made further progress. Intuitively, this is the knowledge gained by seeing a trace t followed by any other event. Progress knowledge allows us to define a progress-insensitive security condition.

Before formally defining knowledge, we define low-equivalence of heaps and traces, which is used in the definition of security.

Definition 1 (Low-equivalence of Heaps and Traces). Let $\ell \in \mathcal{L}$. Let H_0 and H_1 be heaps, and let $H \upharpoonright_P$ denote heap H restricted to references at place P (each reference is assigned uniquely and permanently to a place.) All references at other places are undefined. $H_0 \sim_\ell H_1$ if $H_0 \upharpoonright_P = H_1 \upharpoonright_P$ for all places P such that $\mathit{PlaceLevel}(P) \sqsubseteq \ell$. Let t_0 and t_1 be traces, and let $t \upharpoonright_P$ denote trace t restricted to events at place P. $t_0 \sim_\ell t_1$ if $t_0 \upharpoonright_P = t_1 \upharpoonright_P$ for all places P such that $\mathit{PlaceLevel}(P) \sqsubseteq \ell$. Note that low-equivalence is an equivalence relation.

Definition 2 (Attacker Knowledge)**.** For any $\ell \in \ell$, program $\langle P, s\rangle$, trace t, and refiner R, define attacker knowledge as:

$$k(\langle P, s\rangle, t, P, R) = \{\omega \mid \exists t'.(H_{init}; \omega; R; \epsilon; \langle P, s\rangle) \rightsquigarrow t', t' \upharpoonright_P = t \upharpoonright_P\}$$

Define progress knowledge as:

$$k^+(\langle P, s\rangle, t, P, R) = \{\omega \mid \exists t', \alpha.\mathit{PlaceOfEvent}(\alpha) = P, (H_{init}; \omega; R; \epsilon; \langle P, s\rangle) \rightsquigarrow t' \cdot \alpha, t' \upharpoonright_P = t \upharpoonright_P\}$$

where $\mathit{PlaceOfEvent}(\alpha)$ is the place at which event α occurs.

Our security condition requires that the knowledge gained from seeing a trace $t \cdot \alpha$ is no more specific than the knowledge gained from seeing trace t followed by any other event. That is, seeing the event α has not given an attacker any more information than seeing any other event would have. Theorem 1 proves this statement of the security condition.

Definition 3 (Knowledge-Based Security)**.** Program $\langle P, s\rangle$ is secure if for all $\ell \in \mathcal{L}$ and all configurations $m = (H_{init}; \omega; R; \epsilon; \langle P, s\rangle)$ such that $m \rightsquigarrow t \cdot \alpha$, we have $k(\langle P, s\rangle, t \cdot \alpha, P', \ell) \supseteq k^+(\langle P, s\rangle, t, P', \ell) \cap [\omega]_\ell$ for any P' such that $\mathit{PlaceLevel}(P') \sqsubseteq \ell$, where $[\omega]_\ell$ is the equivalence class of ω under relation $\sim_\ell$.

3.4 Proof of Security Condition

This section provides a proof of the security condition given in Definition 3. Below is the theorem we aim to prove. It states that a well-typed program is secure under the security condition.

Theorem 1. *Let $\ell \in \mathcal{L}$. For any program $\langle P_0, s\rangle$, if $\Psi; \Delta \vdash \langle P_0, s\rangle$, then $\langle P_0, s\rangle$ is secure according to Definition 3 on all places P such that $\mathit{PlaceLevel}(P) \sqsubseteq \ell$.*

A high-level overview of this proof is as follows. The proof technique is similar to that of Terauchi [26]. We begin with two configurations which share the same program code but may differ in high input strategies. We then define "erased" states which agree with the original code and refiners on all low computation but perform no computation at high places. We simulate simultaneous runs of original configurations with their corresponding erased configurations to show that the execution trace produced by a configuration agrees with that produced by its erasure on all events observable at low places. Finally, another simulation shows that the two erased configurations, which perform no high computation and therefore do not depend on high inputs, agree on low-observable events.

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 $\Psi;\Delta \vdash T$, if $m \to m'$, then $\Psi;\Delta \vdash T'$.*

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 $P;\Gamma;\Psi;\Delta \vdash s_1 : \ell_1$ and $P;\Gamma;\Psi;\Delta \vdash s_2^{p_2} : \ell_2$ and $\ell_1 \sqsubseteq \Delta(p_2)$, then $P;\Gamma;\Psi;\Delta \vdash s_1.s_2 : \ell_2$.*

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 the typing rule AtStmt, $P_2;\Gamma;\Psi;\Delta \vdash s_1 : \ell_1$, $P_1;\Gamma;\Psi;\Delta \vdash s_2 : \ell_2$ and $\ell_1 \sqsubseteq \Delta(p_2)$. Applying BackAt, $P_2;\Gamma;\Psi;\Delta \vdash \mathsf{backat}\ P_1; s_2 : \ell_2$. Composing this with s_1 and applying TreeStmt gives $\Psi;\Delta \vdash T'$.

- WHILE Then $T = \langle P, \mathsf{while}^p\ e\ \mathsf{do}\ s_1^{p_1}; s_2^{p_2}\rangle$,
  $T' = \langle P, \mathsf{if}\ e\ \mathsf{then}\ s_1.(\mathsf{while}\ e\ \mathsf{do}\ s_1); \mathsf{skip}\ \mathsf{else}\ \mathsf{skip}; s_2\rangle$. By inversion of typing rule WHILE, $P; \Gamma; \Psi; \Delta \vdash s_1 : \ell_1$, $\ell_1 \sqsubseteq \Delta(p_1)$ and $\ell_1 \sqsubseteq \Delta(p_2)$. We have $P; \Gamma; \Psi; \Delta \vdash \mathsf{while}^p\ e\ \mathsf{do}\ s_1^{p_1}; \mathsf{skip} : \ell_1$ where $\Delta(p) = \Delta(p_1)$. Composing gives $P; \Gamma; \Psi; \Delta \vdash s_1.(\mathsf{while}^p\ e\ \mathsf{do}\ s_1); \mathsf{skip} : \ell_1$. We also have $P; \Gamma; \Psi; \Delta \vdash \mathsf{skip} : \ell$, where $\ell \sqsubseteq \ell_1 \sqsubseteq \Delta(p_2)$. Thus, the resulting if statement can be typed with IF and applying TREESTMT gives $\Psi; \Delta \vdash T'$.

- LET Then $T = \langle P, \mathsf{let}\ x = e\ \mathsf{in}\ s_1^{p_1}; s_2^{p_2}\rangle$ and $T' = \langle P, s_1[c/x].s_2\rangle$. By inversion of typing rule LET, $P; \Gamma; \Psi; \Delta \vdash s_1 : \ell_1$. Substitution gives $P; \Gamma; \Psi; \Delta \vdash s_1[c/x] : \ell$ since typing rules VAL and CONST have the same conclusions and CONST has no conditions. Also by inversion, $P; \Gamma; \Psi; \Delta \vdash s_2 : \ell_2$ and $\ell_1 \sqsubseteq \Delta(p_2)$. Composing statements by the claim and applying TREESTMT gives $\Psi; \Delta \vdash T'$.

□

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 3.9. 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 $\surd$ trees with a place, to create the notation $\surd_P$. The semantic rule SKIP1 is modified as follows:

SKIP1

$$\frac{}{(H; \omega; t; \langle P, \mathsf{skip}\rangle) \rightarrow (H; \omega; t; \surd_P)}$$

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

E-LowDone
$$\frac{\mathit{PlaceLevel}(P) \sqsubseteq \ell}{\surd_P \preceq_\ell \surd_P}$$

E-Join
$$\frac{\begin{array}{c} T_1 \preceq_\ell T_1' \\ T_2 \preceq_\ell T_2' \end{array}}{T_1 \triangleright T_2 \preceq_\ell T_1' \triangleright T_2'}$$

E-Para
$$\frac{\begin{array}{c} T_1 \preceq_\ell T_1' \\ T_2 \preceq_\ell T_2' \end{array}}{T_1 \| T_2 \preceq_\ell T_1' \| T_2'}$$

E-Skip1
$$\frac{\mathit{PlaceLevel}(P) \not\sqsubseteq \ell}{\langle P, \mathsf{skip} \rangle \preceq_\ell \surd_P}$$

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

E-HighSeq
$$\frac{\mathit{PlaceLevel}(P) \not\sqsubseteq \ell}{\surd_P \triangleright T \preceq_\ell T}$$

E-HighParaL
$$\frac{\mathit{PlaceLevel}(P) \not\sqsubseteq \ell}{\surd_P \| T \preceq_\ell T}$$

E-HighParaR
$$\frac{\mathit{PlaceLevel}(P) \not\sqsubseteq \ell}{T \| \surd_P \preceq_\ell T}$$

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

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

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

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

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

$$\frac{}{\mathsf{skip} \preceq_\ell \mathsf{skip}} \qquad \frac{}{r := e \preceq_\ell r := e} \qquad \frac{}{\mathsf{output}\ e \preceq_\ell \mathsf{output}\ e} \qquad \frac{}{\mathsf{input}\ r \preceq_\ell \mathsf{input}\ r}$$

$$\frac{}{\mathsf{backat}\ P \preceq_\ell \mathsf{backat}\ P} \qquad \frac{s \preceq_\ell s'}{\mathsf{finish}\ s \preceq_\ell \mathsf{finish}\ s'} \qquad \frac{s \preceq_\ell s'}{\mathsf{async}\ s \preceq_\ell \mathsf{async}\ s'} \qquad \frac{s \preceq_\ell s'}{\mathsf{at}\ P\ s \preceq_\ell \mathsf{at}\ P\ s'}$$

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

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

Figure 3.9: Judgments for erasure

At points in a program when, for some place P and all $p \in \mathit{PointsWaiting}(P)$, we have $\Delta(p) \not\sqsubseteq \mathit{PlaceLevel}(P)$, the type system enforces observational determinism. 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:

$$\textsc{StmtND}\quad \frac{\begin{array}{c}(H; \omega; t; \langle P, s\rangle) \rightarrow (H'; \omega; t'; T') \\ \mathit{PointsRunning}(P) \neq \emptyset \quad \mathit{Sch}(P) = \mathit{chs} \cdot *\end{array}}{(H; \omega; (\mathit{Ps} \cdot P, \mathit{Sch}); t; \langle P, s\rangle) \rightarrow (H'; \omega; (\mathit{Ps}, \mathit{Sch}[P \rightarrow \mathit{chs}]); t'; T')}$$

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 4. Let $\mathit{chs}_0 \cdot E_0$ and $\mathit{chs}_1 \cdot E_1$ be scheduling lists (recall that a scheduler Sch is a function from places to scheduling lists). These two lists are equivalent, written $\mathit{chs}_0 \cdot E_0 \sim \mathit{chs}_1 \cdot E_1$, if $\mathit{chs}_0 \sim \mathit{chs}_1$ 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 5 (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 $LP = \{P \mid \mathit{PlaceLevel}(P) \sqsubseteq \ell\}$ and for all $P \in LP$, $Sch_0(P) \sim Sch_1(P)$ (this relation was defined in Definition 4.)

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 6 (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

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

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 7. Let $m_0 \rightarrow m_1 \rightarrow ... \rightarrow m_n$ 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 $m_i \rightarrow m_{i+1}$ 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 $m_0 \rightarrow^* m_n$ if, for $i \in [0, n-1]$, $E_i = *$ if and only if $\Delta(p) \neq \mathit{PlaceLevel}(P)$ for all $p \in \mathit{PointsWaiting}(P, T_i)$. 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 references. If $H_1 \upharpoonright_C = H_2 \upharpoonright_C$, e is part of a well-typed configuration and dereferences only references in the set C, and $P; H_1; e \Downarrow c$, then $P; H_2; e \Downarrow c$.

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

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

(a)
- CONST is trivial.
- REF By inversion, $H_1(r) = c$. By assumption, $r \in C$, so $H_2(r) = c$ and $P; H_2; !r \Downarrow c$.
- OP $e = e_1 \oplus e_2$. By inversion, $P; H_1; e_1 \Downarrow c_1$, $P; H_1; e_2 \Downarrow c_2$, and $c_1 \oplus c_2 = c$. e_1 and e_2 dereference only references in C, so by induction, $P; H_2; e_1 \Downarrow c_1$ and $P; H_2; e_2 \Downarrow c_2$. Applying OP gives $P; H_2; e \Downarrow c$.

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

$\square$

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 $\Psi; \Delta \vdash T$, $T \preceq_\ell T'$ and $\Delta(p) \sqsubseteq \ell$ for some $p \in \mathit{PointsWaiting}(P, T)$, then $\mathit{PointsRunning}(P, T) = \mathit{PointsRunning}(P, T')$.*

Proof. Let $p \in \mathit{PointsRunning}(P, T)$. $\mathit{PlaceLevel}(P) \sqsubseteq \ell$ since $\mathit{PlaceLevel}(P) \sqsubseteq \Delta(p)$, so by the definition of erasure, $p \in \mathit{PointsRunning}(P, T')$. Now let $p \in \mathit{PointsRunning}(P, T')$ and suppose $p \notin \mathit{PointsRunning}(P, T)$. 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 $p \in \mathit{PointsRunning}(P, T)$. Again by inspection of the rules, we have $\mathit{PlaceLevel}(P) \not\sqsubseteq \ell$ 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 $p \in \mathit{PointsWaiting}(P, T)$. However, by the typing rule for backat, $\mathit{PlaceLevel}(P') \sqsubseteq \Delta(p)$, and so $\Delta(p) \not\sqsubseteq \ell$. We also have that for all $p' \in \mathit{PointsRunning}(P, T)$, $MHP(p', p_1)$, and so $\mathit{PlaceLevel}(P') \sqsubseteq \Delta(p')$ for all p', which is a contradiction. $\square$

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 $m_1 \rightarrow m_2$ 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 $(H_1; \omega; R; t_1; T_1) \rightarrow (H_2'; \omega; R'; t_2'; T_2')$, $R \sim_\ell R_1$, $H_2' \sim_\ell H_1'$, $t_2' \sim_\ell t_1'$, 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 $(H_1'; \omega; R_1'; t_1'; T_1') \rightarrow m_2'$ 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 $m_1 \rightarrow m_2$.

- If this transition instantiates Seq, DoneL or DoneR, 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 $R_1 = (Ps \cdot P, Sch)$ and $\mathit{PointsRunning}(P, T_1') = \emptyset$. Then

$(H'_1; \omega; R'_1; t'_1; T'_1) \rightarrow (H'_1; \omega; R'_2; t'_1; T'_1)$ for some R'_1 such that $R_1 \preceq_\ell R'_1$, and this reduction instantiates Idle. Since no instructions were executed, the third case of the lemma applies.

- If it instantiates Idle with $R_1 = (Ps \cdot P, Sch)$ and $\mathit{PointsRunning}(P, T'_1) \neq \emptyset$, then there is some $p \in \mathit{PointsWaiting}(P, T_1)$ waiting for return of control from a place P', $\mathit{PlaceLevel}(P') \not\sqsubseteq \ell$. $(H_1; \omega; (Ps \cdot P \cdot P', Sch); t_1; T_1) \rightarrow (H'_2; \omega; R_1; t'_2; T'_2)$ for some H'_2, t'_2, T'_2, ch. $(Ps \cdot P \cdot P', Sch) \sim_\ell R_1$. 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 $m_1 \rightarrow m_2$ executes an instruction at a place P, such that $\mathit{PlaceLevel}(P) \not\sqsubseteq \ell$. Then this step must only modify high parts of the tree, heap and trace, so the first case of the lemma applies.

- Suppose $m_1 \rightarrow m_2$ executes an activity $\langle P, \mathsf{at}^p\ P'\ s_1; s_2\rangle$ such that $\mathit{PlaceLevel}(P) \sqsubseteq \ell$ but $\mathit{PlaceLevel}(P') \not\sqsubseteq \ell$. By rule E-At, $T'_1 = T_1[\langle P, \mathsf{at}^p\ P'\ s_1; s_2\rangle \mapsto \langle P, s_2\rangle]$, so $T_2 \preceq_\ell T'_1$ still holds and the first case of the lemma applies.

- Suppose $R_1 = (Ps \cdot P, Sch)$ such that $\mathit{PlaceLevel}(P) \sqsubseteq \ell$ and $m_1 \rightarrow m_2$ executes a program point $p \in \mathit{PointsRunning}(P, T_1)$. If for some $p \in \mathit{PointsWaiting}(P, T_1)$ we have $\Delta(p) = \mathit{PlaceLevel}(P) \sqsubseteq \ell$, then let $R'_1 = (Ps' \cdot P, Sch')$, where $(Ps, Sch) \preceq_\ell (Ps', Sch')$. By Lemma 3, we have $\mathit{PointsRunning}(P, T_1){=}\mathit{PointsRunning}(P, T'_1)$, so instantiating Stmt on both configurations will execute the same program point p. Otherwise, let $R'_1 = (Ps' \cdot P, Sch'[P \mapsto \mathit{chs} \cdot *])$ 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 Stmt or StmtND 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 $m_1 \rightsquigarrow t_0$, then there exist t, t', R_1, R_1' such that*

$$R_0 \sim_\ell R_1,\ R_1 \preceq_\ell R_1'$$

$$(H_1; \omega; R_1; t_1; T_1) \rightsquigarrow t$$

$$(H_1'; \omega; R_1'; t_1'; T_1') \rightsquigarrow t'$$

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

and R_1' is properly formed with respect to the derivation of $(H_1'; \omega; R_1'; t_1'; T_1') \rightsquigarrow t'$.

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 $R_2 \sim_\ell R_i$, $R_i \preceq_\ell R_i'$, $(H_2; \omega; R_i; t_2; T_2) \rightsquigarrow t_i$, $(H_2'; \omega; R_i'; t_2'; T_2') \rightsquigarrow t_i'$ 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 $R_1 = (Ps_i \cdot P, Sch_i[P \mapsto Sch_i(P) \cdot ch])$. 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). Let $R_1' = (Ps_i' \cdot P, Sch_i[P \mapsto Sch_i'(P) \cdot E])$. 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 8 (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 $\mathit{PlaceLevel}(P) \sqsubseteq \ell$. 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 $m_0 = (H; \omega_0; R_0; t; T)$ and $m_1 = (H; \omega_1; R_1; t; T)$ be well-typed erased configurations, with $\Psi; \Delta \vdash T$. Suppose $p_0, p_1 \in PointsRunning(P, T)$, $\Delta(p_0) \not\sqsubseteq PlaceLevel(P)$ and $\Delta(p_1) \not\sqsubseteq PlaceLevel(P)$ (using Δ from the original, non-erased configuration). If $m_0 \to m_0' \to m_0''$, executing p_0 followed by p_1 and $m_1 \to m_1' \to m_1''$ executing p_1 followed by p_0, to reach final configurations*
$m_0'' = (H_0''; \omega_0; R_0''; t_0''; T_0'')$ and $m_1'' = (H_1''; \omega_1; R_1''; t_1''; T_1'')$,
then $H_0'' = H_1''$, $t_0'' = t_1''$ and $T_0'' = T_1''$.

Proof. Let $m_0' = (H_0'; \omega_0; R_0'; t_0'; T_0')$ be the intermediate step such that $m_0 \to m_0'$ executes p_0 and $m_1' = (H_1'; \omega_1; R_1'; t_1'; T_1')$ be the intermediate step such that $m_1 \to m_1'$ executes p_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

$$(H_0; \omega_0; R_0; t_0; S_0) \to (H_0; \omega_0; R_0; t_0; S_0'), (H_1; \omega_1; R_1; t_1; S_1) \to (H_1; \omega_1; R_1; t_1; S_1')$$

By the reduction rules on trees, $T_0' = T[S_0 \mapsto S_0']$ and $T_1' = T[S_1 \mapsto S_1']$.

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 $MHP(p_0, p_1,)$, our definition of Ψ ensures 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 $m_0' \to m_0''$. 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 $m_0' \to m_0''$ contains the reduction

$$(H_0'; \omega_0; R_0'; t_0'; \langle P, i_1; s_1\rangle) \to (H_0''; \omega_0; R_0''; t_0''; T)$$

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 $MHP(p_0, p_1)$, 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, $T_1'' = T_1'[S_0 \mapsto S_0']$. By symmetry, $T_0'' = T_1'' = T[S_0 \mapsto S_0'][S_1 \mapsto S_1']$. □

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 $R_0 = (Ps_0 \cdot P, Sch_0)$, $R_1 = (Ps_1 \cdot P, Sch_1)$ (the same place must be at the front of both lists since $\mathit{PlaceLevel}(P) \sqsubseteq \ell$ and$R_0 \sim_\ell R_1$).

- If $m_0 \to m_0'$ instantiates Seq, DoneL or DoneR, then T contains $\surd \triangleright T'$, $\surd \| T_2$ or $T_1 \| \surd$ (the place attached to $\surd$ 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 $Sch_0(P) = chs_0 \cdot ch$. Since R_0 is properly formed, it must be the case that $\Delta(p) \sqsubseteq \ell$ for some $p \in \mathit{PointsWaiting}(P, T)$. Since R_1 is also properly formed with respect to a step from T and $R_0 \sim_\ell R_1$, $Sch_1(P) = chs_1 \cdot ch$. Thus, $m_0 \to m_0'$ and $m_1 \to m_1'$ both instantiate Stmt with the same ch, so both execute the same instruction. The statement reduction rules only depend on H, t and $\omega(P, t \restriction_P)$, all of which are the same for both configurations, so $T_0' = T_1', H_0' = H_1'$ and $t_0' = t_1'$.

- Otherwise, $Sch_0(P) = chs_0 \cdot *$. Since R_0 is properly formed, it must be the case that $\Delta(p) \not\sqsubseteq \ell$ for all $p \in \mathit{PointsWaiting}(P, T)$. Since R_1 is also properly formed with respect to a step from T and $R_0 \sim_\ell R_1$, $Sch_1 = chs_1 \cdot *$, and $m_0 \to m_0'$ and $m_1 \to m_1'$ instantiate StmtND. 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 StmtND with different instructions, then suppose $m_0 \to m_0'$ executes program point p_0 and $m_1 \to m_1'$ executes program point p_1. Since R_0 and R_1 were properly formed, $\Delta(p_0) \not\sqsubseteq \mathit{PlaceLevel}(P)$ and $\Delta(p_1) \not\sqsubseteq \mathit{PlaceLevel}(P)$. Let $R_0'' = R_0$, $R_1'' = R_1$. This allows the transitions

$$m_0 \to (H_0'; \omega_0; (Ps_0 \cdot P, Sch_0[P \mapsto chs_0 \cdot *]); t_0'; T_0') \to (H_0''; \omega_0; R_0'; t_0''; T_0'')$$

$$m_1 \to (H_1'; \omega_1; (Ps_1 \cdot P, Sch_1[P \mapsto chs_1 \cdot *]); t_1'; T_1') \to (H_1''; \omega_1; R_1'; t_1''; T_1'')$$

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, $\Delta(p) \neq \mathit{PlaceLevel}(P)$ for all $p \in \mathit{PointsWaiting}(P, T)$. The typing rules FINISH and TREEJOIN ensure that if all activities at a place are tainted with high information, this will be the case until they all finish executing, so $\Delta(p) \neq \mathit{PlaceLevel}(P)$ for all $p \in \mathit{PointsWaiting}(P, T'_0)$ and $p \in \mathit{PointsWaiting}(P, T'_1)$. Thus, $(Ps_0 \cdot P, Sch_0[P \mapsto chs_0 \cdot *])$ and $(Ps_1 \cdot P, Sch_1[P \mapsto chs_1 \cdot *])$ 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 [26, 27]. 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 $\Psi; \Delta \vdash T$, $R_0 \sim_\ell R_1$, $\omega_0 \sim_\ell \omega_1$, *if* $m_0 \rightsquigarrow t_0$, $m_1 \rightsquigarrow t_1$ *and* R_0 *and* R_1 *are properly formed with respect to their respective derivations, then for all* P, *either* $t_0 \restriction_P$ *is a prefix of* $t_1 \restriction_P$ *or* $t_1 \restriction_P$ *is a prefix of* $t_0 \restriction_P$.

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, $t_0 \restriction_P$ and $t_1 \restriction_P$ are both prefixes of $t' \restriction_P$, and so $t_0 \restriction_P$ is a prefix of $t_1 \restriction_P$ or $t_1 \restriction_P$ is a prefix of $t_0 \restriction_P$. □

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 $\Psi; \Delta \vdash \langle P, s\rangle$, R is fully specified, and $\omega_0 \sim_\ell \omega_1$, if $m_0 \rightsquigarrow t_0$ and $m_1 \rightsquigarrow t_1$, then for all P, $PlaceLevel(P) \sqsubseteq \ell$, $t_0 \upharpoonright_P$ 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 $\langle P, s\rangle \preceq \langle P, s'\rangle$. If a program is well-typed, its erasure is well-typed with the same Δ and Ψ, so $\Psi; \Delta \vdash \langle P, s'\rangle$. By Lemma 5, there exist $t_0', t_1', t_0'', t_1'', R_0, R_1, R_0', R_1'$ such that

$$(H_{init}; \omega_0; R_0; \epsilon; \langle P, s\rangle) \rightsquigarrow t_0',$$

$$(H_{init}; \omega_1; R_1; \epsilon; \langle P, s\rangle) \rightsquigarrow t_1',$$

$$(H_{init}; \omega_0; R_0'; \epsilon; \langle P, s'\rangle) \rightsquigarrow t_0'',$$

$$(H_{init}; \omega_1; R_1'; \epsilon; \langle P, s'\rangle) \rightsquigarrow t_1'',$$

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

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 $PlaceLevel(P) \sqsubseteq \ell$. 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. $\square$

We now prove Theorem 1.

Proof of Theorem 1. Let P be a place such that $\mathit{PlaceLevel}(P) \sqsubseteq \ell$. 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 3 is empty, the theorem is trivially satisfied. Otherwise, there exists an $\omega' \sim_\ell \omega$ such that $(H_{init}; \omega'; R; \epsilon; \langle P, s\rangle) \rightsquigarrow t' \cdot \alpha'$ and $t' \restriction_P = t \restriction_P$. By Theorem 2, $(t \cdot \alpha) \restriction_P$ is a prefix of $(t' \cdot \alpha') \restriction_P$ or vice versa. Let the function $\mathit{PlaceOfEvent}()$ return the place of an event. If $\mathit{PlaceOfEvent}(\alpha) = P$, then $(t \cdot \alpha) \restriction_P$ and $(t' \cdot \alpha') \restriction_P$ are the same length since $\mathit{PlaceOfEvent}(\alpha') = P$ by definition and $t' \restriction_P = t \restriction_P$. Therefore, $(t \cdot \alpha) \restriction_P = (t' \cdot \alpha') \restriction_P$ and $\omega' \in k(\langle P, s\rangle, t \cdot \alpha, P, \ell)$. If $\mathit{PlaceOfEvent}(\alpha) \neq P$, then $(t \cdot \alpha) \restriction_P = t \restriction_P = t' \restriction_P$. Since $(H_{init}; \omega'; R; \epsilon; \langle P, s\rangle) \rightsquigarrow t'$, $\omega' \in k(\langle P, s\rangle, t \cdot \alpha, P, \ell)$. This shows the required inclusion. $\square$

Chapter 4

Implementation and Case Studies

4.1 Implementation

In this section, we describe the changes necessary to scale from the FSX10 model to SX10, a more practical language that can, in principle, support almost all of the features of the X10 language. We then describe an implementation of the SX10 type system building upon the existing compiler for X10.

4.1.1 Scaling Up: From FSX10 to SX10

The full SX10 language looks quite similar to X10, but contains some restrictions and extra annotations which facilitate the security analysis. A notable feature of X10 not present in SX10 is `at` expressions, which are not included for reasons discussed in Section 3.2. As noted, one could extend the language to allow limited support for `at` expressions, but we leave this to future work. An additional restriction enforced in SX10 but not X10 is that places must be static. X10 allows the place term for an `at` statement to be a general expression that evaluates to a Place object. This feature may allow covert channels which leak information through the choice of place at which a computation is executed. As a result, we simplify the analysis by only allowing the place term of an `at` statement to be an identifier corresponding to a place. Place identifiers must be specified at compile time and cannot be added or modified at runtime. Places are declared and annotated with security levels using configuration files, which are passed to the compiler as command line arguments. SX10 supports arbitrary finite latices. The language need not support infinite lattices, as the number of security levels is bounded by the number of places.

SX10 requires another type of annotation as well. Statements that may input or output over a console must be annotated with `sxinput` or `sxoutput`. It is up to the programmer to determine what statements may result in an observable "input" or "output" and be disciplined in annotating accordingly. An implementation of SX10 could infer these annotations, but our prototype implementation currently does not.

It is relatively straightforward to adapt the FSX10 type system presented in Chapter 3 to the full SX10 language. As noted by Lee and Palsberg [12], most interesting features of the may-happen-in-parallel analysis for X10 can be seen in the analysis for FX10. Similarly, the FSX10 type system demonstrates most of the constraints needed in the SX10 type system. The typing rule for `while` loops, for example, can be modified slightly and used for other types of loops, and the typing rules for dereference and assignment are used for local variables and array access.

4.1.2 Prototype Implementation

We have begun work on implementing the SX10 type system, and have developed a prototype implementation over a reasonable subset of the X10 language. The X10 distribution contains an open source X10-to-Java compiler which is built using the Polyglot extensible compiler framework [15]. We implemented SX10 by modifying this compiler.

Our implementation consists of adding three passes to the X10 compiler. Like the implementation of the may-happen-in-parallel analysis [12], each pass builds a set of constraints. Each AST node adds constraints based on rules derived from the type system. After all of the constraints for a pass are generated, these constraints are solved before the subsequent pass. The first two passes implement the may-happen-in-parallel analysis, following the implementation described by Lee and Palsberg [12]. Note that Lee and Palsberg's implementation uses three passes. The first generates constraints to build the set $SLabels(p)$ for each program point p, which is a conservative estimate of the program points that may be encountered during execution of the statement at p. For example, in the program $(\mathsf{if}^{p_1}\ e\ \mathsf{then\ skip}^{p_2}\ \mathsf{else\ skip}^{p_3}); \mathsf{skip}^{p_4}$, we have $SLabels(p_1) = \{p_1, p_2, p_3\}$. Following this are two passes that generate *level 1* and then *level 2* constraints. Level 1 constraints act over sets of program points (variables r and o in Lee and Palsberg's notation), and level 2 constraints use these variables to build the may-happen-in-parallel sets, which are sets of pairs of program points. Our implementation combines the second and third passes, and generates and solves level 1 and level 2 constraints at once.

The third pass in our implementation collects and solves constraints for the SX10 analysis. For each program point p, variables $\Delta(p)$ and $\Delta'(p)$ are generated. $\Delta(p)$ represents the security level of the place at which p executes before p executes. $\Delta'(p)$ represents the security level of this place after p executes. The map $\Psi(p, \gamma)$ for channels γ and program points p is generated prior to SX10 constraint generation. For simplicity, $\Psi(p, \gamma)$ is defined to be 0 if p may happen in parallel with a write to γ, 0.5 if p may happen in parallel with a read from γ and 1 otherwise.

We implement one optimization on the type system. Recall the constraint from the FSX10 type system on program points p that ensures timing information from high at statements will be propagated to concurrent activities:

$$\forall p, p' \in M(p), \Delta(p') \sqsubseteq \Delta(p)$$

In the FSX10 type system, this was included as a condition for every typing rule, and so this constraint is enforced on every program point p. In our implementation, we enforce this constraint only for async and at statements. This is equivalent, since $MHP(p)$ is determined by p's enclosing async and at statements. The information is propagated through to p by the constraint for statements $i^p; s^{p'}$ requiring $\Delta(p) \sqsubseteq \Delta(p')$.

4.2 Case Studies

In this section, we implement two sample programs that model real parallel computations in which security plays a role. These sample programs have been written in SX10 and compiled with the modified X10 compiler including our SX10 security analysis. They demonstrate that it is possible to write practical, secure, parallel programs using SX10.

4.2.1 Online Shopping

Following Tsai, Russo and Hughes[7], we use the example of a server running a shopping website. This is a good example on which to test our framework, since shopping websites are extremely prevalent, handle large amounts of data concurrently and have a clear security concern, that is, 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-observable. 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. For simplicity of the code, we use only two activities and assume only two users. This is not a fundamental restriction of the language, however. Segments of the code for this example are reproduced below:

```
public class Shopping {
  public static def main(Array[String]) {
    val items = new Array[Int](2);
    val costs = new Array[Double](5);
    //Initialize costs array

    while(true) {
      //Input all item numbers (low security)
      finish {
        async {
          //Get the customer id number
          val id:int;

          sxinput id = Int.parse(Console.IN.readLine().trim());
          sxinput items(id) = Int.parse(Console.IN.readLine().trim());
          at log {
            sxoutput Console.OUT.println(id + "\t" + items(id));
          }
        }
        //Same for other user
        }
      }
      //Input all credit card numbers (high security)
      finish at highform {
        async {
              val id:int;
              val card:int;
              sxinput id = Int.parse(Console.IN.readLine().trim());
              sxinput card = Int.parse(Console.IN.readLine().trim());
              at cc {
                    sxoutput Console.OUT.println(card + "\t" + costs(items(id)));
              }
        }
        //Same for other user
        }
      }
    }
  }
}
```

First, the server spawns two threads at place `lowform`. The two threads, in parallel, read input on the first form for two users. This is simulated by taking console input. Both threads switch to place `log` and output the ID and order. The order is also recorded in the local array `items`, which persists through the rest of this loop iteration. These threads then join, and the server spawns two threads at place `highform`, which read inputs from the second form. The credit card number and amount to be charged (which is determined from the order in the `items` array) are then output on the console at place `cc` to simulate exporting this data for processing.

Places `lowform` and `log` are at equal security levels, and are protected by `highform` and `cc`, which are also at equal security levels. Since credit card information is introduced at `highform`, this ensures that secure credit card data must be limited to places `highform` and `cc` and no information about this data can appear in the log of customer IDs and orders.

It is clear from this example that some concurrency is lost and overhead is gained in joining and re-spawning threads twice per loop iteration. One of these join points is, however, necessary to ensure the security of the information. A program in which threads didn't join after outputting credit card information would not be secure according to SX10's type system, since the computation would become tainted with high-security information. Observational determinism would then require that writes to the log be synchronized, which would again add overhead. Note, however, that the joining of threads between writing to the log and elevating to `highform` is not strictly necessary, since the threads terminate at place `cc`, and so it could be argued that no information returns to a low-security place. The SX10 type system could be relaxed to allow such programs by adding a special case for `at` statements at the end of `async` branches. However, we leave such a modification to future work.

This example represents a somewhat nonstandard use of places in X10, if the program is assumed to model a process running on a single server. In this case, places are used only to separate security levels, and not to physically separate computation. The next example demonstrates a case in which places are used to separate security levels as well as to separate parts of a distributed system.

4.2.2 Mobile GPS Navigation

Most smartphones sold today are equipped with a GPS and navigation software. However, many smartphone users, the author included, would like to be assured of the security of their location information. We model a system consisting of a number of phones communicating with a central location database that maps destinations to GPS coordinates. In this system, many phones are calculating routes simultaneously, and the database is serving many concurrent requests. Despite this concurrency, we wish to ensure that no data about users' locations leaks to the database or other phones.

In our model, a phone prompts the user for the desired destination, e.g., an address or perhaps "Hotel" or "Restaurant," and then queries the global database for the coordinates of the destination. The coordinates will be sent back to the phone, which will then use its GPS to determine the distance to the destination (one could also imagine calculating the route; the security policies are identical). Each phone runs computation at two places: a low-security place that accepts the destination and a high-security place that processes location information. The central database is stored at a separate place. Thus, in this example, places are used both to separate computations at different security levels on one machine and to separate computations on different machines. For simplicity, we model the map as a one-dimensional array `locations`. The coordinates of a "location" consist of the index at which the string for that location is stored. Again, we restrict to two users, but could have allowed more. Selected portions of the code for this example are reproduced below.

```
public class GPS {
    public static def main(Array[String]) {
        val locations = new Array[String](NUM_LOCATIONS, "");
        //Initialize locations array

        async at phone0 {
            //Get desired destination
            val dest:String;
            sxinput dest = Console.IN.readLine().trim();

            //Send to database to look up coordinates
            at db {
                val coord = lookup(dest, locations);
                //Send coordinates to GPS
                at gps0 {
                    val loc = getCurrentLocation();
                    Console.OUT.println("Distance: " + Math.abs(coord - loc));
                }
            }
        }
        //Same code for phone1...
    }
}
```

Methods `lookup(dest, locations)` and `getCurrentLocation()` are omitted for the purposes of clarity, but can be implemented in a straightforward manner. The method `lookup(dest, locations)` returns the coordinates of `dest`, and `getCurrentLocation()` returns the current GPS coordinates of the phone. Since method calls have not yet been implemented for the SX10 analysis, implementations of these methods were inlined when the code was compiled. The full code was compiled using the X10 compiler with SX10 analysis and a configuration file specifying the security policy described below.

Each phone i runs two places, `phone`i and `gps`i, which handle destination information and current location information respectively. For each i, `phone`i is at a security level protected by that of the global database at `db`, and this is in turn protected by the security level of `gps`i. However, the security levels of `gps`i and `gps`j for $i \neq j$ are not related, and this fact disallows any information about one phone's location from flowing to any other place.

Unlike the previous example, this code requires no additional synchronization on the worker threads. Both may proceed through the entire computation in parallel. The main performance difference between this code and an insecure but optimal version of the same code is the overhead of running computations at two separate places on each phone. This example also shows how security policies can be constructed easily and intuitively. An SX10 programmer simply needs to determine what components of the program handle data at different security levels, assign each component to a place, and construct the security policy for these places appropriately.

Chapter 5

Conclusion

This thesis demonstrates a language in which concurrent programs with strong security guarantees can be easily written. The security type system for this language uses the abstraction of places in X10 to make explicit the separation of data and computation at different security levels. Within a place, all data and computation are tainted with the same level of information and so, unlike in previous security analyses for concurrent programs, implicit flows, including timing channels, need not be considered within a place. The syntax also makes clear the points in a program at which data and timing information may flow between places. These two features lead to coarse-grained security policies which are easy for programmers to construct and about which it is easy to reason. We described a prototype implementation of the type system on a subset of X10. Finally, we presented two case studies which demonstrate the use of SX10 in writing practical, secure parallel programs.

5.1 Observations and Discussion

A number of observations can be drawn from the SX10 type system and the case studies shown in Section 4.2. The first observation comes directly from the type system. Data and computation may only flow from low-security places to high-security places. If this policy were not enforced, data could flow to and influence computation at lower-security places. The use of the X10 syntax and place abstraction makes it easier to see when this property is violated. The policy can be, and is, enforced, by disallowing `at` statements to move computation to a higher-level place. For similar reasons, the use of `at` expressions is very rarely secure and so they are not included in the language. It must be noted that when computation returns from an `at` statement, there is a flow of information from a high to a low place. To ensure that this does not leak information to an attacker, low-observable events must be deterministic at these points, or synchronization must be applied to ensure that low-observable events may not happen in parallel with these place changes. The type system enforces these restrictions as well.

It may seem that the separation of data and computation enforced by SX10 will result in unusually cumbersome use of places and synchronization. While SX10 may restrict X10 programmers from using the natural style to which they are accustomed, the case studies show that SX10 can be used to produce practical programs, especially if certain design patterns are kept in mind. One intuitive such pattern is a pipeline in which data moves through a linear sequence of stages in a computation, which are represented by places of nondecreasing security levels. Such a pipeline can be seen in the GPS navigation example, where data moves from the user input place to the global database and then to the GPS place. Since these places are in order of increasing security level and no data is required to flow in the other direction, this program is intuitively secure and is accepted as such by the type system.

Another pattern can be seen in the online shopping example. In this program, we wish to run concurrent computations at mixed security levels. In order to do this securely, our implementation alternates between two phases, one in which low data is accepted and processed, and one in which high data is accepted and processed. Separating these phases allows observational nondeterminism in the order in which the requests are processed. Otherwise, there would be a channel for data in a high phase to flow into a simultaneously running low phase. This extra synchronization is not necessary in the GPS example because the outputs occur at different places which are unrelated in the security lattice, and so observational determinism, by our definition, trivially applies. This is a trade-off that is likely to be seen often in SX10 programs. When writing programs with computation at multiple security levels, one must give up concurrency somewhere, either to synchronize concurrent writes and eliminate nondeterminism, or to eliminate timing channels between high and low security computations. A guiding principle of SX10 and future similar languages should be to allow programmers the greatest possible level of concurrency while maintaining strong security guarantees.

5.2 Future Work

Many features of X10 were omitted from the analysis, but could be added. Method calls, for example, are present in the may-happen-in-parallel analysis of Lee and Palsberg and could easily be added to the FSX10 model and SX10 implementation with few additional security constraints. Objects could be added to the language and would pose challenges mainly in the implementation. Few conceptual changes would have to be made to the model to support objects.

Additional synchronization mechanisms could also be added to the language. We believe that these additions would mainly require refinements to the may-happen-in-parallel analysis but few, if any, additional security constraints. For example, support for clocks, which is included in X10, could be added to the FSX10 model using semantics similar to those of Saraswat and Jagadeesan [24]. The main conceptual challenge would be developing a may-happen-in-parallel analysis which supports clocks. A very conservative approximation, which ignores the clock phases and assumes that any instructions in two clocked threads may occur in parallel, would require little additional effort. A more refined analysis, however, would allow programmers to more naturally express patterns like the two-phase loop seen in the online shopping example. Since the details of the may-happen-in-parallel analysis are outside the scope of this thesis, we leave it to future work.

Futures are also supported as a synchronization mechanism in X10, and could also be added to SX10 with a refined may-happen-in-parallel analysis. For example, the code in a future evaluated at place P may happen in parallel with any computation at place P occurring in parallel with the declaration of the future, and would therefore leak information to this computation. Information flows back, however, only to the thread that forces the future, and only at the time it is forced. Again, we leave this refinement to future work.

We have discussed three axes along which a secure parallel language can be evaluated. Such a language can allow a high degree of concurrency, allow a high degree of nondeterminism, and be simple to use. These goals are, to some extent, in conflict. However, we believe that SX10, which allows practical implementations of highly concurrent systems, allows nondeterminism where it is secure to do so, and lends itself to intuitive, coarse-grained policies and implementations, is a promising development on all three axes.

Bibliography

[1] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 207–221, May 2007.

[2] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3):21:1–21:32, July 2010.

[3] A. Birgisson, A. Russo, and A. Sabelfeld. Capabilities for information flow. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11, pages 5:1–5:15, New York, NY, USA, 2011. ACM.

[4] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 4–4, Berkeley, CA, USA, 2009. USENIX Association.

[5] 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*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.

[6] J. Boyland. Checking interference with fractional permissions. In *Proceedings of*

the 10th international conference on Static analysis, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[7] T c Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in haskell. In *Computer Security Foundations Symposium, 2007. CSF '07. 20th IEEE*, pages 187 –202, July 2007.

[8] 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. *SIGPLAN Not.*, 40:519–538, October 2005.

[9] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, May 1976.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20:504–513, July 1977.

[11] D. Grove, O. Tardieu, D. Cunningham, B. Herta, I. Peshansky, and V. Saraswat. A performance model for x10 applications. 2011.

[12] J. K. Lee and J. Palsberg. Featherweight x10: a core calculus for async-finish parallelism. *SIGPLAN Not.*, 45:25–36, January 2010.

[13] H. Mantel. Information Flow Control and Applications – Bridging a Gap. In Jose Nuno Olivera and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe*, LNCS 2021, pages 153–172, Berlin, Germany, March 12-16 2001. Springer.

[14] H. Mantel and H. Sudbrock. Flexible scheduler-independent security. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 116–133, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] V. Saraswat et al. X10 language specification, version 2.2. May 2011.

[24] V. Saraswat and R. Jagadeesan. *Concurrent clustered programming*, pages 353–367. Springer-Verlag, London, UK, 2005.

[25] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 355–364, New York, NY, USA, 1998. ACM.

[26] T. Terauchi. A type system for observational determinism. In *Computer Security Foundations Symposium, 2008. CSF '08. IEEE 21st*, pages 287–300, June 2008.

[27] T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *ACM Trans. Program. Lang. Syst.*, 30:27:1–27:30, September 2008.

[28] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996.

[29] 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.

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