ABSTRACT

Title of dissertation: Language-based Enforcement of User-defined Security Policies
As Applied to Multi-tier Web Programs

Nikhil Swamy, Doctor of Philosophy, 2008

Directed by: Professor Michael Hicks
Department of Computer Science

Over the last 35 years, researchers have proposed many different forms of security policies to control how information is managed by software, e.g., multi-level information flow policies, role-based or history-based access control, data provenance management etc. A large body of work in programming language design and analysis has aimed to ensure that particular kinds of security policies are properly enforced by an application. However, these approaches typically fix the style of security policy and overall security goal, e.g., information flow policies with a goal of noninterference. This limits the programmer's ability to combine policy styles and to apply customized enforcement techniques while still being assured the system is secure.

This dissertation presents a series of programming-language calculi each intended to verify the enforcement of a range of user-defined security policies. Rather than "bake in" the semantics of a particular model of security policy, our languages are parameterized by a programmer-provided specification of the policy and enforcement mechanism (in the form of code). Our approach relies on a novel combination of dependent types to correctly associate security policies with the objects they govern, and affine types to

account for policy or program operations that include side effects. We have shown that our type systems are expressive enough to verify the enforcement of various forms of access control, provenance, information flow, and automata-based policies. Additionally, our approach facilitates straightforward proofs that programs implementing a particular policy achieve their high-level security goals. We have proved our languages sound and we have proved relevant security properties for each of the policies we have explored. To our knowledge, no prior framework enables the enforcement of such a wide variety of security policies with an equally high level of assurance.

To evaluate the practicality of our solution, we have implemented one of our type systems as part of the LINKS web-programming language; we call the resulting language SELINKS. We report on our experience using SELINKS to build two substantial applications, a wiki and an on-line store, equipped with a combination of access control and provenance policies. In general, we have found the mechanisms SELINKS provides to be both sufficient and relatively easy to use for many common policies, and that the modular separation of user-defined policy code permitted some reuse between the two applications.

Language-based Enforcement of User-defined Security Policies

As Applied to Multi-tier Web Programs

NIKHIL SWAMY

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Advisory Committee:
Professor Michael W. Hicks, Chair/Advisor
Professor Samrat Bhattacharjee
Professor Jeffrey S. Foster
Professor Jeffrey W. Herrmann, Dean's representative
Professor Jonathan Katz

Acknowledgments

This dissertation would have been impossible, and my experience of graduate school would have been much diminished, were it not for the help, guidance, support, and friendship of several people.

I am most indebted to my advisor, Mike Hicks. Despite several false starts, and through the bleak middle years of graduate school that saw several research projects fizzle out for one reason or another, Mike's constant encouragement was a source of much-needed confidence. Watching Mike devote himself to all he does, with what appears to always be the most natural ease, has been nothing short of inspirational. Mike has been a teacher, a role model, a confidant, an indulgent sparring partner in numerous arguments, and, above all else, a very good friend.

I am also grateful to all the members of PLUM, the programming languages research group at the University of Maryland. To Jeff Foster (and Mike again), for making PLUM a creative, productive, supportive, and, basically, just a really fun research group. To Brian Corcoran, whose help with SELINKS has been invaluable. And, for way too many things to mention, to Iulian Neamtiu, Polyvios Pratikakis, Saurabh Srivastava, Mike Furr, Pavlos Papageorge, Nick Petroni, David Greenfieldboyce, Martin Ma, Chris Hayden, Khoo Yit Phang, Elnatan Reisner, David An, and Eric Hardisty.

Finally, I am deeply obliged to my family. They have patiently tolerated the garbled explanations of my work that I have grudgingly offered from time to time, and have always responded with quiet encouragement. Their unquestioning support has been, for me, an ever-present exhortation to try to do what's right, never fearing for the outcome, or being mindful of the fruits, should there be any, of my work.

Table of Contents

List of Figures

viii

1. Introduction

The 9/11 Commission Report, in an attempt to explain the failure of the United States government to prevent the September 11, 2001, terrorist attacks, included the following statement among its general findings:

> Action officers should have been able to draw on all available knowledge about al Qaeda in the government. Management should have ensured that information was shared and duties were clearly assigned across agencies, and across the foreign-domestic divide. [...] The U.S. government did not find a way of pooling intelligence and using it to guide the planning and assignment of responsibilities for joint operations ... [92]

In response to these findings, and driven in part by the success of web sites like Wikipedia, YouTube, Flickr, Facebook, and MySpace,[1] the U.S. government has begun using web applications to disseminate critical information in a timely manner across its various divisions. Examples include *SKIWEB* [20], used for "strategic knowledge integration" in the U.S. military, and *Intellipedia* [108], a set of web-based document management systems used throughout the sixteen agencies that comprise the U.S. intelligence community. While many of the details about these applications are classified, a recent press release by the Central Intelligence Agency (CIA) about Intellipedia includes the

---

[1] wikipedia.org, youtube.com, flickr.com, facebook.com, myspace.com

1

following statement:

> ... the CIA now has users on its top secret, secret and sensitive unclassi-
> fied networks reading and editing a central wiki that has been enhanced with
> a YouTube-like video channel, a Flickr-like photo-sharing feature, content
> tagging, blogs and RSS feeds. [34]

Of course, sharing sensitive information via the web is not limited to the U.S. government. For example, in the United Kingdom, the National Health Service's *Spine* is a web-based application intended to provide health-care providers with convenient access to a patient's records [93]. Even within smaller organizations, web-based information sharing is common, e.g., web applications like *Continue* [74] and *EasyChair* [134] are frequently used to manage academic conferences.

While there are substantial efficiencies to be had from information sharing, clearly, there can also be significant consequences—loss of life, health-based discrimination, identity theft, etc.—should sensitive information not be properly protected. Networked information-sharing applications must therefore balance two competing ends: to maximize the sharing of information while mitigating, to the greatest extent possible, the risk due to unauthorized release of, or tampering with, sensitive information.

Take the case of Intellipedia: to protect against improper usage of sensitive intelligence documents, it should address a number of security concerns. At the most basic level, it should control which users can *access* content by enforcing forms of multi-level security policies. Intellipedia may also need to track *provenance* information [22], such as revision history and data sourcing, on documents to reason about information integrity

and to support auditing. To improve information availability, a policy may release content

to a certain user with some particularly sensitive information withheld, either by redaction

or by some other form of downgrading. Computations over the document databases, like

PageRank-style algorithms [98], should be careful to respect these security concerns so

as not to inadvertently leak sensitive data in response to search queries.

Without recourse to formal verification to ensure that all these security needs have

been met, Intellipedia currently heads off the threat of information misuse by placing

sharp limits on the sharing of information. For example, confidentiality is achieved by

falling back on the security of the underlying computer networks of the U.S. Department

of Defense. The DoD manages several computer networks each cleared to handle data at

specific classification levels—*NIPRNet* may only handle sensitive but unclassified data,

SIPRNet is cleared for secret data, and *JWICS* for top-secret data [129]. Versions of

Intellipedia that are accessible on each of these networks are kept physically separate,

and all access controls are applied at the network level. But, by resorting to an "air

gap" to secure sensitive data, Intellipedia surrenders much of the benefit that may be had

from sharing information at a fine granularity.[2] For example, a document that contains

a fragment of top-secret information must be placed in JWICS, even though much of

its content may be of relevance to users with a lower clearance. To work around these

limitations, such a document may have to be downgraded and copied into one of the

other networks. But, if the downgrading is not performed properly, top-secret data may

have been leaked inadvertently. Additionally, as documents are edited, it is easy for the

---

[2]Admittedly, the enforcement of data confidentiality on U.S. DoD systems is strongly influenced by well-entrenched institutional practices. As for other concerns, like reliable tracking of data provenance, it appears highly unlikely that Intellipedia provides any formal guarantees about these.

original document and the downgraded version to become inconsistent, compromising information integrity.

Instead, we would like to allow Intellipedia to share information at a fine granularity while verifying that the software meets all the security requirements. In pursuit of this goal, one must, of course, begin by formalizing the security requirements. Policy languages that can articulate such requirements have been the focus of a number of research efforts—e.g., UNIX-style access control lists, RBAC [107], XACML [143], Ponder [38], and trust management frameworks like RT [76]. Evidently, the *raison d'être* of each of these languages is to express enforceable specifications of the permissible behaviors of a system. This latter point is the focus of our work: given one of these security policies, how do we assure that a software system enforces it properly?

A simple way to think about this question is as *complete mediation*—are all security-sensitive operations properly mediated by queries to the security policy? Researchers have proposed using static analysis of software source code to check this condition. For example, Zhang et al. [148] used CQual to check that SELinux operations on sensitive objects are always preceded by policy checks; Fraser [53] did the same for Minix. However, these systems only ensure that *some* policy function is called before data is accessed. Calls to the wrong policy function or incorrect calls to the right one (e.g., with incorrect arguments) are not prevented.

Security-typed programming languages, like Jif [31] or FlowCaml [104], aim to allay this weakness. Through the use of novel type systems, these languages are able to show that well-typed programs enjoy useful extensional security properties as a consequence of complete mediation. These security properties are typically based on forms

of *noninterference* for lattice-based information flow policies [42], and roughly means that high-security data cannot be inferred via a low-security channel. But, security-typed languages have problems of their own. First, noninterference is often too strong—some protected information inevitably must be released, either via downgrading or even according to simpler access control policies. Moreover, security-typed languages usually fix the mechanisms by which a policy is specified. For example, policies in Jif are specified using the decentralized label model [88], which may not always be suitable for certain applications. We have observed, for instance, that a role-based label model may be better when policies are expected to change at runtime [125]. Finally, information flow policies provide no clear way to express and provably enforce data provenance or other styles of policy, such as security automata [113] and history- or stack-based access control [1, 51]. All told, despite their promise, security-typed programming languages are not yet flexible enough to guarantee the enforcement of the variety of security policies that often must be applied to real-world applications.

This dissertation sets out to demonstrate that security typing can be made flexible enough to be applied to a broad range of policies. In particular, our thesis is the following:

> *A language-based framework, while being practical enough to construct real applications, can be used to verify that a range of user-defined security policies are correctly enforced, and that, as a consequence, programs enjoy useful extensional security properties.*

1.1 Overview of our Approach

The main contribution of this dissertation is a generalization of security typing that does not "bake in" a particular model of security as primitive. Instead, using a novel combination of several standard (but advanced) type-theoretic constructs, we are able to enforce various forms of access control, provenance, information flow, and automata-based policies. While our encodings make specific design choices, our solution is flexible enough that programmers can control all the low-level details of policy specification and enforcement. For example, programmers are free to develop custom label models when enforcing an information flow policy; or, they may implement an access control policy using capabilities, access control lists, or some combination of the two. Nevertheless, we retain the benefit of traditional security-typed languages by being able to show that type-correct programs enjoy useful security properties. We have implemented our ideas in a programming language that we call SELINKS and have validated its practicality by building SEWIKI, a web-based document management system inspired by Intellipedia. In this section, we present a summary of the main elements of our approach.

1.1.1 A Brief Primer on Security Typing

Volpano et al. [132] were the first to propose using a type system to certify the enforcement of a security policy. In particular, they address the enforcement of information flow security policies specified in Denning's lattice model [42]. In this model, a policy is specified as a lattice $(\mathscr{L}, \sqsubseteq)$, where $\mathscr{L}$ is a finite set of security labels partially ordered by the relation $\sqsubseteq$. For example, $\mathscr{L}$ may identify secrecy classes like `High` and `Low`, with

Low $\sqsubseteq$ High, where these classes are used to categorize the secrecy of objects in a system. Informally, the intention of such a policy is to ensure that no information about an object of the High security class flows to an object of the Low security class.

The key insight of Volpano et al. was to refine the types of a programming language to include a security label. For example, the type int^{High} represents the set of all High security integers. Volpano et al. define a type system that tracks the flow of information through the various constructs of a programming language. For instance, for a program statement $h := l$, where h has the type int^{High} and l has the type int^{Low}, the typing judgment records a flow from the Low security class to the High class. Such a flow is accepted as secure for the lattice Low $\sqsubseteq$ High. However, an assignment in the opposite direction ($l := h$) is judged by the type rules to cause a flow from High to Low and is deemed insecure for the example lattice. A program containing such an insecure assignment is rejected as type-incorrect.

A large body of work [111] has extended these basic ideas of security typing to accommodate variations on the lattice model and to incorporate information flow analyses of the programming constructs of real languages (e.g., exceptions, higher-order functions, objects etc.). Jif [31], an extension of Java, and FlowCaml [104], an extension of Caml, are two noteworthy language implementations that utilize security typing to guarantee the correct enforcement of information flow policies.

1.1.2 Enforcing User-defined Security Policies

Our work begins with the observation that the many security policies are enforce-able by associating labels with data in the types (as in Jif and FlowCaml), where the label expresses the security policy for that data. What varies among policies is the *specification* and *interpretation* of labels, in terms of the actions that are permitted or denied. By allowing the syntax and semantics of labels to be user-defined, we stand to benefit from the high degree of assurance provided by security typing while still retaining the flexibility we desire to enforce a range of policies.

In subsequent chapters, we develop FABLE, λAIR, and FLAIR, a succession of programming-language calculi, each building upon the previous, which embody this observation in two respects. First, a policy designer can define custom security labels and associate them with the data they protect using *dependent types* [4]. Next, rather than "hard-code" their semantics, policy enforcement is parameterized by a programmer-provided interpretation of labels, specified in a privileged part of the program. The type system forbids application programs from manipulating data with a labeled type directly. Instead, in order to use labeled data, the application must call the appropriate privileged functions that interpret the labels. By verifying the interpretation of labels, and relying on the soundness of the type system, policy implementers can prove that type-correct programs enjoy relevant security properties.

In our first language, FABLE, a programmer could define a label *High*, and give a high-security integer value a type that mentions this label, int{*High*}. As another example, the programmer could define a label *ACL*(*Alice*, *Bob*) to stand for an access control list

and give an integer a type such as int$\{ACL(Alice, Bob)\}$. Programmers define the interpretation of labels in an *enforcement policy*, a set of privileged functions distinguished from the rest of the program. Thus, in order to capture the intuition that an integer with the type int$\{ACL(Alice, Bob)\}$ is only to be accessed by *Alice* or *Bob*, one writes an enforcement policy function like the following:

policy *access_simple* (*acl*:lab, *x*:int$\{acl\}$) = if (*member user acl*) then $\{\circ\}x$ else -1

Here, *access_simple* takes a label *acl* as its first argument (like *ACL(Alice, Bob)*), and an integer protected by that label as its second argument. If the current user (represented by the variable *user*) is a member of *x*'s access control list *acl* (according to some function *member*, not shown), then *x* is returned with its label removed, expressed by the syntax $\{\circ\}x$, which coerces *x*'s type to int so that it can be accessed by the main program. If the membership test fails, it returns -1, and *x*'s value is not released. By preventing the main program (i.e., the non-policy part) from directly examining data with a labeled type, we can ensure that all its operations on data with a type like int$\{ACL(Alice, Bob)\}$ are preceded by a call to the *access_simple* policy function, which performs the necessary access control check.

In Chapter 2, we show that FABLE is powerful enough to encode the enforcement of various styles of access control, data provenance, and information flow policies. In each case, we state and prove useful security properties for well-typed programs. However, FABLE is limited in that it applies only to purely functional programs. While the purely functional setting is both useful and illustrative, complex real-world policies often rely on some mutable state to make authorization decisions. In Chapter 3, we define λAIR, a

calculus that can enforce stateful policies in addition to the policies enforceable in FABLE.

We present λAIR by first picking a specific model for stateful policies. In particular, we propose AIR, a novel model for specifying high-level information release protocols (a kind of declassification policy [112]) in terms of security automata [113]. AIR is of independent interest in that, to our knowledge, it is the first time security automata have been used to specify information release protocols. To enforce AIR policies, sensitive data in λAIR are labeled with states from an AIR security automaton. Prior to using labeled data, a λAIR program must call functions (analogous to enforcement policy functions in FABLE) that consult the state of the automaton mentioned in the label and the usage is permitted only if it authorized by the automaton. Since the state of the automaton changes as the program executes, the type system of λAIR has to be careful to ensure that stale automaton states are never used in authorization decisions. The technical machinery that accomplishes this is the use of *affine types* [140]. We prove that an AIR policy is correctly enforced in λAIR by showing that type correct λAIR programs (that use type signatures corresponding to specific AIR policy) produce execution traces that are strings in the language accepted by the AIR automaton.

Of course, useful real-world programs include side effects, e.g., some output is printed to the terminal, or a message is sent over the network. In λAIR, we model state updates in a purely functional way. While this is sufficient if programs are always written in a monadic style, we show in Chapter 4 that the combination of affine and dependent types in λAIR is powerful enough to enforce policies for programs that may cause side effects directly. The main contribution of Chapter 4 is our final calculus FLAIR, which extends λAIR with mutable references to memory. We develop an encoding of a canonical

information flow policy in FLAIR and show that type correct FLAIR programs using this encoding enjoy a standard noninterference property.

The FLAIR calculus is our main evidence in support of the claim that a language-based framework can be expressive enough to support the enforcement of broad range of policies. Chapter 3 shows that FLAIR can be used to enforce security automata policies. As a consequence of prior work on the expressiveness of security automata [113, 59, 13, 79], we get a useful lower bound on the class of properties enforceable in FLAIR—informally, a broad class of safety properties. Additionally, FLAIR can enforce noninterference, which has been categorized variously as a *2-safety* property [126] and, more recently, as a kind of *hyperproperty* [33]. We make no attempt to enforce liveness properties in FLAIR.

Unlike traditional security type systems which guarantee that type-correct programs enjoy strong security properties like noninterference, our method does not, in and of itself, guarantee any such property. Clearly, by allowing the semantics of policy enforcement to be user-defined we open the possibility of a programmer constructing policies that are patently insecure—we make no attempt to prevent this. However, the design of each of our type systems facilitates (and, indeed, greatly simplifies) proofs that the enforcement of a specific policy entails a corresponding security property of type-correct programs—we have conducted these proofs for each of the policies explored in this dissertation. This stands as evidence for the claim that our approach admits proofs of extensional security properties for programs.

1.1.3 Building Secure Web Applications

Given the increasing demand for web-based information availability, the construction of secure web applications is a useful point of reference when evaluating the practicality of a new approach to security. As such, we have used FABLE in the design of a new programming language called SELINKS. The final claim of our thesis is that SELINKS is practical enough to be used in the construction of realistic secure web applications.

Web applications are often distributed over several tiers. In a typical configuration, such an application is comprised of a client tier, where much of user-interface logic runs in a web browser (as JavaScript); a server tier, where the bulk of the application logic is executed (in a language like Java or PHP); and finally, a database tier (executing SQL code) that serves as a high-efficiency persistent store. Our effort to construct secure web applications begins with the LINKS programming language [35]. LINKS is designed to make web programming easier. Rather than programming each tier in a separate language, in LINKS, a programmer writes an entire multi-tier web application as a single program. The compiler splits that program into components to run on the client (as JavaScript), server (as a local fragment of LINKS code), and database (as SQL). From our perspective LINKS is also useful in that it makes it easier to reason about the security behavior of all three tiers of an application by analyzing a single source program.

In Chapter 5, we describe our extension to LINKS called *Security-Enhanced* LINKS, or SELINKS. Our extensions consist of two main components. The first is a new type system for LINKS based on FABLE-style security typing. Next, in order to efficiently enforce security policies in data-access code, we have designed and implemented a novel

compilation procedure that translates SELINKS enforcement policy code to user-defined functions that can be run in the database. Our experiments show that this compilation strategy can improve the throughput of a database query by as much as an order of magnitude.

To evaluate the practicality of SELINKS (and, by extension, FABLE), we have constructed two medium-sized web applications that enforce custom security policies using SELINKS. Describing these applications is the main focus of Chapter 6. The larger of these two applications is SEWIKI, a web-based document management system inspired by Intellipedia, that enforces a custom combination of a fine-grained access control policy and a provenance-tracking policy on HTML documents. The second application, SEWINESTORE, is an e-commerce application distributed with LINKS that we have extended with an access control policy. In general, we have found that SELINKS' label-based security policies are sufficient to enforce many interesting policies and are relatively easy to use. Additionally, the modular specification of the enforcement policy permits some reuse of policy code between the two applications.

A limitation of our evaluation is that we have only implemented the FABLE system for SELINKS. We have yet to evaluate the practicality of the full-generality of FLAIR for enforcing user-defined policies. As it stands, we conjecture that FLAIR may be more suitable as the basis of an intermediate language, rather than as a type system for a source level language like SELINKS. A detailed discussion of this and other limitations, along with steps we might take to overcome them, can be found in Chapter 8.

1.2 Summary of Contributions

In summary, this dissertation makes the following contributions.

1. We define FABLE, a core calculus for the enforcement of purely functional user-defined policies. We have proved FABLE sound. We provide encodings in FABLE of the following security policies and prove each of them correct:

   - Two styles of access control, one based on inlined policy checks and another based on capabilities. We formulate an extensional correctness property for access control called *non-observability* and prove that our encoding satisfies this property.

   - A data provenance tracking policy augmented with an access control policy to protect the provenance metadata itself. We prove that our encodings satisfy *dependency correctness*, a standard property for data provenance [26].

   - Two versions of a lattice-based information flow policy, one with static labels and fully static enforcement, and another with dynamic labels. We prove a standard noninterference property for the static information flow policy.

2. We propose AIR, a novel policy language for expressing high-level information release protocols based on security automata. AIR is intended to promote reasoning about the declassification behavior of a system independently from the system's implementation.

3. We define λAIR, a calculus that extends dependent typing in FABLE with support for affine types. We have proved λAIR sound. We show that λAIR can be used

to enforce stateful security policies by developing an enforcement mechanism for automata-based policies expressed in the AIR language. We prove that type-correct λAIR programs enjoy a trace-based correctness property (standard for automata-based policies [139]).

4. We extend λAIR with mutable references to produce the calculus FLAIR. We have proved FLAIR sound. We show how FLAIR can be used to enforce a static information flow policy while accounting for information leaks due to side effects on memory. We prove that type-correct FLAIR programs that use our encoding enjoy a standard noninterference property. Additionally, we show how the basic FABLE type system can be embedded in FLAIR and argue, as a consequence, that all the security policies explored in this dissertation can be enforced using FLAIR.

5. We implement SELINKS, an extension of the LINKS web-programming language with support for enforcing user-defined security policies, in the style of FABLE. SELINKS also includes a novel compilation strategy for enforcement policy functions that enables security policies to be seamlessly and efficiently enforced for code spanning the server and database tiers.

6. We demonstrate the practicality of SELINKS by building two substantial multi-tier web applications. The first, SEWIKI, is a web-based document management system that enforces a combination of fine-grained access controls and provenance tracking on HTML documents. The second, SEWINESTORE, is an e-commerce application retrofitted with an access control policy.

2. Enforcing Purely Functional Policies

This chapter presents FABLE, a core formalism for a programming language in which programmers may specify security policies and reason that these policies are properly enforced. We focus here on purely functional policies applied to purely functional programs. To illustrate FABLE's flexibility we show how to use it to encode a range of policies, including access control, static [111] and dynamic information flow [149], and provenance tracking [26].

In our experience, the soundness of FABLE makes proofs of security properties no more difficult—and arguably simpler—than proofs of similar properties in specialized languages [104, 127, 132]. To demonstrate this fact we present proofs of correctness for our access control, provenance, and static information flow policies (Appendix A), using three substantially different proof techniques. While precisely stating correctness properties for each of these policies required some careful construction, we found it relatively easy to discharge proofs of these properties by relying on various lemmas from the metatheory of FABLE. This experience indicates that with the accumulation of a set of broadly applicable lemmas about FABLE, many of our security proofs could be partially automated; however, we leave exploration of this issue to future work.

Expressions	e	$::=$	$n \mid x \mid \lambda x{:}t.e \mid e_1\, e_2 \mid \text{fix } x{:}t.v \mid \Lambda \alpha.e \mid e\,[t]$
(Fable-specific)		$\mid$	$C(\vec{e}) \mid \text{match } e \text{ with } p_i \Rightarrow e_i \mid (\![e]\!) \mid \{\circ\}e \mid \{e'\}e$
Types	t	$::=$	$\text{int} \mid \alpha \mid \forall \alpha.t_2 \mid (x{:}t_1) \rightarrow t_2$
(Fable-specific)		$\mid$	$\text{lab} \mid \text{lab} \sim e \mid t\{e\}$
Patterns	p	$::=$	$x \mid C(\vec{p})$
Pre-values	u	$::=$	$n \mid C(\vec{u}) \mid \lambda x{:}t.e \mid \Lambda \alpha.e$
App. values	v_{app}	$::=$	$u \mid (\![\{e\}v_{pol}]\!)$
Policy values	v_{pol}	$::=$	$u \mid \{e\}v_{pol}$

Figure 2.1: Syntax of FABLE

2.1 FABLE: System F with Labels

This section presents the syntax, static semantics, and operational semantics of FABLE. The next section illustrates FABLE's flexibility by presenting example policies along with proofs of their attendant security properties.

2.1.1 Syntax

Figure 2.1 defines FABLE's syntax. Throughout, we use the notation $\vec{a}$ to stand for a list of elements of the form $a_1, \ldots, a_n$. Where the context is clear, we will also treat $\vec{a}$ as the set of elements $\{a_1, \ldots, a_n\}$.

Expressions e extend a standard polymorphic λ-calculus, System F [55]. Standard forms include integer values n, variables x, abstractions $\lambda x{:}t.e$, term application $e_1\, e_2$, the fixpoint combinator $\text{fix } x{:}t.v$, type abstraction $\Lambda \alpha.e$, and type application $e\,[t]$. We exclude mutable references from the language to simplify the presentation. Subsequent chapters extend the language with references and considers their effect on various policies, e.g., information flows through side effects.

The syntactic constructs specific to FABLE are distinguished in Figure 2.1. The

expression $C(\vec{e})$ is a label, where C represents an arbitrary constructor and each $e_i \in \vec{e}$ must itself be a label. For example, in *ACL(Alice, Bob)*, *ACL* is 2-ary label constructor and *Alice* and *Bob* are 0-ary label constructors. Labels can be examined by pattern matching. For example, the expression match z with $ACL(x,y) \Rightarrow x$ would evaluate to *Alice* if z's run-time value were *ACL(Alice, Bob)*.

As explained earlier, FABLE introduces the notion of an *enforcement policy* that is a separate part of the program authorized to manipulate the labels on a type. Following Grossman et al. [58] we use *bracketed expressions* $(\![e]\!)$ to delimit policy code e from the main program. In practice, one could use code signing as in Java [56] to ensure that untrusted policy code cannot be injected into a program. As mentioned earlier, the expression $\{\circ\}e$ removes a label from e's type, while $\{e'\}e$ adds one. Labeling and unlabeling operations may only occur within policy code; we discuss these operations in detail below.

Standard types t include int, type variables α, and universally quantified types $\forall \alpha.t$. Functions have dependent type $(x{:}t_1) \rightarrow t_2$ where x names the argument and may be used in t_2. We illustrate the usage of these types shortly. Labels can be given either type lab or the *singleton type* lab$\sim e$, which describes label expressions equivalent to e. For example, the label constructor *High* can be given the type lab and the type lab$\sim$*High*. Singleton types are useful for constraining the form of label arguments to enforcement policy functions. For example, we could write a specialized form of our previous *access_simple* function:

policy *access_pub* (*acl*:lab$\sim$*ACL(World)*, x:int$\{acl\}$) = $\{\circ\}x$

The FABLE type checker ensures this function is called only with expressions that evaluate to the label *ACL(World)*, i.e., the call *access_pub(ACL(Alice,Bob),e)* will be rejected. In effect, the type checker is performing access control at compile time according to the constraint embodied in the type. We will show in Section 2.2.3 that these constraints are powerful enough to encode an information flow policy that can be checked entirely at compile time.

The dependent type $t\{e\}$ describes a term of type t that is associated with a label e. Such an association is made using the syntax $\{e\}e'$. For example, $\{High\}1$ is an expression of type $\text{int}\{High\}$. Conversely, this association can be broken using the syntax $\{\circ\}e$. For example, $\{\circ\}(\{High\}1)$ has type int. Now we illustrate how dependent function types $(x{:}t_1) \to t_2$ can be used. The function *access_simple* can be given the type $(acl{:}\text{lab}) \to (x{:}\text{int}\{acl\}) \to \text{int}$ which indicates that the first argument *acl* serves as the label for the second argument x. Instead of writing $(x{:}t_1) \to t_2$ when x does not appear in t_2, we simply omit it. Thus *access_simple*'s type could be written $(acl{:}\text{lab}) \to \text{int}\{acl\} \to \text{int}$.

The operational semantics of Section 2.1.4 must distinguish between application and policy values to ensure that policy code does not inadvertently grant undue privilege to application functions. Application values v_{app} consist of either "pre-values" u—integers n, labels containing values, type and term abstractions—or labeled policy values wrapped with $(\![\cdot]\!)$ brackets. Values within policy code are pre-values preceded by zero or more relabeling operations.

Encodings. To make our examples more readable, we use the syntactic shorthands shown in Figure 2.2. The first three shorthands are mostly standard. We use the policy keyword to

type abbreviation

$$\text{typename } N\,\alpha = t \text{ in } e_2 \qquad \equiv \qquad (N\,t' \mapsto ((\alpha \mapsto t')t))e_2$$

let binding, for some t

$$\text{let } x = e_1 \text{ in } e_2 \qquad \equiv \qquad (\lambda x{:}t.e_2)\,e_1$$

polymorphic function definition, for some t'

$$\text{let } f\langle\alpha\rangle(x{:}t) = e_1 \text{ in } e_2 \qquad \equiv \qquad \text{let } f = \text{fix } f{:}t'.\Lambda\alpha.\lambda x{:}t.e_1 \text{ in } e_2$$

policy function def, for some t'

$$\text{policy } f\langle\alpha\rangle(x{:}t) = e_1 \text{ in } e_2 \qquad \equiv \qquad \text{let } f = \text{fix } f{:}t'.\Lambda\alpha.\lambda x{:}t.(\!|e_1|\!) \text{ in } e_2$$

dependent tuple type

$$x{:}t \times t' \qquad \equiv \qquad \forall\alpha.((x{:}t) \to t' \to \alpha) \to \alpha$$

dependent tuple introduction, for some t, t'

$$(e, e') \qquad \equiv \qquad \Lambda\alpha.\lambda f{:}((x{:}t) \to t' \to \alpha).f\,e\,e'$$

dependent tuple projection, for some t, t', and t_e

$$\text{let } x,y = f \text{ in } e \qquad \equiv \qquad f\,[t_e](\lambda x{:}t.\lambda y{:}t'.e)$$

Figure 2.2: Syntactic shorthands

designate policy code instead of using brackets $(\!|\cdot|\!)$. A dependent pair (e, e') of type $x{:}t \times t'$

allows x, the name for the first element, to be bound in t', the type of the second element.

For example, the first two arguments to the *access_pub* function above could be pack-

aged into a dependent pair of type $(acl{:}\mathsf{lab}\sim ACL(World) \times \mathsf{int}\{acl\})$, which is inhabited

by terms such as $(ACL(World), \{ACL(World)\}1)$. Dependent pairs can be encoded using

dependently typed functions. We extend the shorthand for function application, policy

function definitions, type abbreviations, and tuples to multiple type and term arguments

in the obvious way. We also write _ as a wildcard ("don't care") pattern variable.

Phantom label variables. We extend the notation for polymorphic functions in a way that

permits quantification over the *expressions* that appear in a type. Consider the example

below:

$$\mathsf{policy}\ add\langle l\rangle(x{:}\mathsf{int}\{l\}, y{:}\mathsf{int}\{l\}) = \{l\}(\{\circ\}x + \{\circ\}y)$$

20

This policy function takes two like-labeled integers x and y as arguments, unlabels them and adds them together, and finally relabels the result, having type $\mathsf{int}\{l\}$. This function is unusual because the label l is not a normal term argument, but is being quantified—any label l would do.

The reason this makes sense is that in FABLE, (un)labeling operations are merely hints to the type checker to (dis)associate a label term and a type. These operations, along with all types, can be erased at runtime without affecting the result of a computation. After erasing types, our example would become policy $add\ (x, y) = x + y$, which is clearly only a function of x and y, with no mention of l. For this reason, we can treat add as *polymorphic in the labels* of x and y—it can be called with any pair of integers that have the same label, irrespective of what label that might be. We express this kind of polymorphism by writing the *phantom label variable l*, together with any other normal type variables like $\alpha, \beta, \ldots$, in a list that follows the function name. In the example above, the phantom variable of add are listed as $\langle l \rangle$. Of course, not all label arguments are phantom. For instance, in the *access_simple* function of Section 1.1.2, the *acl* is a label argument that *is* passed at runtime. For simplicity, we do not formalize phantom variable polymorphism here. Chapter 5 shows the key judgments related to phantom variable polymorphism; a related technical report [123] contains a proof of soundness.

2.1.2   Example: A Simple Access Control Policy

Figure 2.3 illustrates a simple but complete enforcement policy for access control. Protected data is given a label listing those users authorized to access the data. In partic-

policy *login*(*user*:string, *pw*:string) =
    let *token* = match *checkpw user pw* with
        USER(*k*) $\Rightarrow$ *USER*(*k*)
        _ $\Rightarrow$ *FAILED* in
    (*token*, {*token*}0)

let *member*(*u*:lab, *a*:lab) =
    match *a* with
      ACL(*u*, *i*) $\Rightarrow$ *TRUE*
      ACL(*j*, *tl*) $\Rightarrow$ *member u tl*
      _ $\Rightarrow$ *FALSE*

policy *access*$\langle k, \alpha \rangle$(*u*:lab $\sim$ *USER*(*k*), *cap*:int{*u*}, *acl*:lab, *data*:α\{*acl*\}) =
    match *member u acl* with
      TRUE $\Rightarrow$ {$\circ$}*data*
      _ $\Rightarrow$ halt

Figure 2.3: Enforcing a simple access control policy

ular, such data has type *t*{*acl*}, where *acl* encodes the ACL as a label.

The policy's *login* function calls an external function *checkpw* to authenticate a user by checking a password. If authentication succeeds (the first pattern), *checkpw* returns a label *USER*(*k*) where *k* is some unique identifier for the user. The *login* function returns a pair consisting of this label and a integer labeled with it; this pair serves as our runtime representation of a principal. The *access* function takes the two elements of this pair as its first two arguments. Since FABLE enforces that only policies can produce labeled values, we are assured that the term with type int{*USER*(*k*)} can only have been produced by *login*. The *access* function's last two arguments consist of the protected data's label, *acl*, and the data itself, *data*. The *access* function calls the *member* function to see whether the user token *u* is present in the ACL. If successful, the label *TRUE* is returned, in which case *access* returns the data with its *acl* label removed.

2.1.3 Typing

Figure 2.4 defines the typing rules for FABLE. The main judgment $\Gamma \vdash_c e : t$ types expressions. The index c indicates whether e is part of the policy or the application. Only policy terms are permitted to use the unlabeling and relabeling operators. Γ records three kinds of information: $x{:}t$ maps variables to types, α records a bound type variable, and $e \succ p$ records the assumption that e matches pattern p, used when checking the branches of a pattern match.

The rules (T-INT), (T-VAR), (T-FIX), (T-TAB) and (T-TAP) are standard for polymorphic lambda calculi. (T-ABS) and (T-APP) are standard for a dependently typed language. (T-ABS) introduces a dependent function type of the form $(x{:}t_1) \rightarrow t_2$. (T-APP) types an application of a (dependently typed) function. As usual, we require the type t_1 of the argument to match the type of the formal parameter to the function. However, since x may occur in the return type t_2, the type of the application must substitute the actual argument e_2 for x in t_2. As an example, consider an application of the *access_simple* function, having type $(acl{:}\mathsf{lab}) \rightarrow \mathsf{int}\{acl\} \rightarrow \mathsf{int}$, to the term $ACL(Alice,Bob)$. According to (T-APP) the resulting expression is a function with type $\mathsf{int}\{ACL(Alice,Bob)\} \rightarrow \mathsf{int}$, which indicates that the function can be applied only to an integer labeled with precisely $ACL(Alice,Bob)$. This is the key feature of dependent typing—the type system ensures that associations between labels and the terms they protect cannot be forged or broken.

Rule (T-LAB) gives a label term $C(\vec{e})$ a singleton label type $\mathsf{lab} \sim C(\vec{e})$ as long as each component $e_i \in \vec{e}$ has type lab. According to this rule $ACL(Alice,Bob)$ can be given the type $\mathsf{lab} \sim ACL(Alice,Bob)$. For that matter, the expression $((\lambda x{:}\mathsf{lab}.x)\ High)$ can be

23

$\Gamma \vdash_c e : t$ Expression e has type t in environment Γ under color c

$$
\begin{array}{lll}
\text{Environments} & \Gamma & ::= \quad \cdot \mid x{:}t \mid \alpha \mid e \succ p \mid \Gamma_1, \Gamma_2 \\
\text{Substitutions} & \sigma & ::= \quad \cdot \mid (x \mapsto e) \mid (\alpha \mapsto t) \mid \sigma_1, \sigma_2 \\
\text{Colors} & c & ::= \quad pol \mid app
\end{array}
$$

$$\frac{\Gamma \vdash_c e : t \qquad \Gamma \vdash t \cong t'}{\Gamma \vdash_c e : t'} \text{(T-CONV)}$$

$$\Gamma \vdash_c n : \text{int} \;\text{(T-INT)} \qquad \frac{x{:}t \in \Gamma}{\Gamma \vdash_c x : t}\;\text{(T-VAR)} \qquad \frac{\Gamma \vdash t \qquad \Gamma, f{:}t \vdash_c v : t}{\Gamma \vdash_c \text{fix } f{:}t.v : t}\;\text{(T-FIX)}$$

$$\frac{\Gamma, \alpha \vdash_c e : t}{\Gamma \vdash_c \Lambda \alpha.e : \forall \alpha.t}\;\text{(T-TAB)} \qquad \frac{\Gamma \vdash t \qquad \Gamma \vdash_c e : \forall \alpha.t'}{\Gamma \vdash_c e\,[t] : (\alpha \mapsto t)t'}\;\text{(T-TAP)}$$

$$\frac{\Gamma \vdash t \qquad \Gamma, x{:}t \vdash_c e : t'}{\Gamma \vdash_c \lambda x{:}t.e : (x{:}t) \rightarrow t'}\;\text{(T-ABS)} \qquad \frac{\Gamma \vdash_c e_1 : (x{:}t_1) \rightarrow t_2 \qquad \Gamma \vdash_c e_2 : t_1}{\Gamma \vdash_c e_1\,e_2 : (x \mapsto e_2)t_2}\;\text{(T-APP)}$$

$$\frac{\Gamma \vdash_c e_i : \text{lab}}{\Gamma \vdash_c C(\vec{e}) : \text{lab} \sim C(\vec{e})}\;\text{(T-LAB)} \qquad \frac{\Gamma \vdash_c e : \text{lab} \sim e'}{\Gamma \vdash_c e : \text{lab}}\;\text{(T-HIDE)} \qquad \frac{\Gamma \vdash_c e : \text{lab}}{\Gamma \vdash_c e : \text{lab} \sim e}\;\text{(T-SHOW)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_c e : \text{lab} \qquad \Gamma \vdash t \qquad\quad p_n = x \quad \text{where } x \notin \text{dom}(\Gamma) \\ \vec{x}_i = FV(p_i) \setminus \text{dom}(\Gamma) \qquad \Gamma, \vec{x}_i{:}\text{lab} \vdash_c p_i : \text{lab} \qquad \Gamma, \vec{x}_i{:}\text{lab}, e \succ p_i \vdash_c e_i : t\end{array}}{\Gamma \vdash_c \text{match } e \text{ with } p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n : t}\;\text{(T-MATCH)}$$

$$\frac{\Gamma \vdash_{pol} e : t\{e'\}}{\Gamma \vdash_{pol} \{\circ\}e : t}\;\text{(T-UNLAB)} \qquad \frac{\Gamma \vdash_{pol} e : t \qquad \Gamma \vdash_{pol} e' : \text{lab}}{\Gamma \vdash_{pol} \{e'\}e : t\{e'\}}\;\text{(T-RELAB)} \qquad \frac{\Gamma \vdash_{pol} e : t}{\Gamma \vdash_c (\![e]\!) : t}\;\text{(T-POL)}$$

$\Gamma \vdash t \cong t'$ Types t and t' are convertible

$$
\begin{array}{lll}
\text{Type contexts} & T & ::= \quad \bullet \mid \bullet\{e\} \mid x{:}\bullet \rightarrow t \mid x{:}t \rightarrow \bullet \mid \forall \alpha.\bullet \\
\text{Term label contexts} & L & ::= \quad \text{lab} \sim \bullet \mid t\{\bullet\}
\end{array}
$$

$$\Gamma \vdash t \cong t \;\text{(TE-ID)}$$

$$\frac{\Gamma \vdash t \cong t'}{\Gamma \vdash t' \cong t}\;\text{(TE-SYM)} \qquad \frac{\Gamma \vdash t \cong t'}{\Gamma \vdash T \cdot t \cong T \cdot t'}\;\text{(TE-CTX)} \qquad \frac{e \succ p \in \Gamma}{\Gamma \vdash L \cdot e \cong L \cdot p}\;\text{(TE-REFINE)}$$

$$\frac{\forall \sigma.(\text{dom}(\sigma) = FV(e_1) \wedge \Gamma \vdash \sigma(e_1) : \text{lab}) \;\Rightarrow\; \sigma(e_1) \overset{c}{\leadsto} \sigma(e_2) \wedge \Gamma \vdash \sigma(e_2) : \text{lab}}{\Gamma \vdash L \cdot e_1 \cong L \cdot e_2}\;\text{(TE-REDUCE)}$$

$\Gamma \vdash t$ Type t is well-formed in environment Γ

$$\Gamma \vdash \text{int} \;\text{(K-INT)} \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}\;\text{(K-TVAR)} \qquad \Gamma \vdash \text{lab} \;\text{(K-LAB)} \qquad \frac{\Gamma \vdash_{pol} e : \text{lab}}{\Gamma \vdash \text{lab} \sim e}\;\text{(K-SLAB)}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash_{pol} e : \text{lab}}{\Gamma \vdash t\{e\}}\;\text{(K-LABT)} \qquad \frac{\Gamma \vdash t_1 \qquad \Gamma, x{:}t_1 \vdash t_2}{\Gamma \vdash (x{:}t_1) \rightarrow t_2}\;\text{(K-FUN)} \qquad \frac{\Gamma, \alpha \vdash t}{\Gamma \vdash \forall \alpha.t}\;\text{(K-ALL)}$$

Figure 2.4: Static semantics of FABLE

given the type $\mathsf{lab} \sim ((\lambda x{:}\mathsf{lab}.x)\ High)$; there is no requirement that e be a value. The rule (T-HIDE) allows a singleton label type like this one to be subsumed to the type of all labels, lab. Rule (T-SHOW) does the converse, allowing the type of a label to be made more precise.

Rule (T-MATCH) checks pattern matching. The first premise confirms that expression e being matched is a label. The second line of premises describes how to check each branch of the match. Our patterns differ from patterns in ML in two respects. First, the second premise on the second line requires $\Gamma, \vec{x}_i : \mathsf{lab} \vdash_c p_i : lab$, indicating that patterns in FABLE are allowed to contain variables that are defined in the context Γ. Second, pattern variables may occur more than once in a pattern. Both of these features make it convenient to use pattern matching to check for term equality. For example, in the expression $\mathsf{let}\ y = Alice\ \mathsf{in}\ \mathsf{match}\ x\ \mathsf{with}\ ACL(y,y) \Rightarrow e$, the branch e is evaluated only if the runtime value for the label variable x is $ACL(Alice, Alice)$.

A key feature of (T-MATCH) is the final premise on the second line, which states that the body of each branch expression e_i should be checked in a context including the assumption $e \succ p_i$, which states that e matches pattern p_i. This assumption can be used to refine type information during checking (similar to typecase [60]) using the rule (T-CONV), which we illustrate shortly. (T-MATCH) also requires that variables bound by patterns do not escape their scope by appearing in the final type of the match; this is ensured by the second premise, $\Gamma \vdash t$, which confirms t is well formed in the top-level environment (i.e., one not including pattern-bound variables). For simplicity we require a default case in pattern-matching expressions: the third premise requires the last pattern to be a single variable x that does not occur in Γ.

Rule (T-UNLAB) types an unlabeling operation. Given an expression e with type $t\{e'\}$, the unlabeling of e strips off the label on the type to produce an expression with type t. Conversely, (T-RELAB) adds a label e' to the type of e. The *pol*-index on these rules indicates that both operations are only admissible in policy terms. This index is introduced by (T-POL) when checking the body of a bracketed term $(\![e]\!)$. For example, given expression $e \equiv \lambda x{:}\mathsf{int}\{Public\}.(\![\{\circ\}x]\!)$, we have $\cdot \vdash_{app} e : \mathsf{int}\{Public\} \to \mathsf{int}$ since $\{\circ\}x$ will be typed with index *pol* by (T-POL).

Rule (T-CONV) allows e to be given type t' assuming it can given type t where t and t' are *convertible*, written $\Gamma \vdash t \cong t'$. Rules (TE-ID) and (TE-SYM) define convertibility to be reflexive and symmetric. Rule (TE-CTX) structurally extends convertibility using *type contexts T*. The syntax $T \cdot t$ denotes the application of context T to a type t which defines the type that results from replacing the occurrence of the hole $\bullet$ in T with t. For example, if T is the context $\bullet\{C\}$, then $T \cdot \mathsf{int}$ is the type $\mathsf{int}\{C\}$. (Of course, rule (TE-CTX) can be applied several times to relate larger types.)

The most interesting rules are (TE-REFINE) and (TE-REDUCE), which consider types that contain labels (constructed by applying context L to an expression e). Rule (TE-REFINE) allows two structurally similar types to be considered equal if their embedded expressions e and p have been equated by pattern matching, recorded as the constraint $e \succ p$ by (T-MATCH). To see how this would be used, consider the following example:

$$
\begin{aligned}
&\mathsf{let}\ tok,cap = login\ \texttt{"Joe"}\ \texttt{"xyz"}\ \mathsf{in} \\
&\quad \mathsf{match}\ tok\ \mathsf{with}\ USER(k) \Rightarrow access\ tok\ cap \\
&\qquad\qquad\qquad\quad \_ \Rightarrow \mathsf{halt}
\end{aligned}
$$

We give the *login* function the type $\mathsf{string} \to \mathsf{string} \to (l{:}\mathsf{lab} \times \mathsf{int}\{l\})$. The type of

access (defined in Figure 2.3) is $(u{:}\mathsf{lab}{\sim}USER(k)) \to \mathsf{int}\{u\} \to t$. We type check *access tok* using rule (T-APP), which requires that the function's parameter and its formal argument have the same type t. However, here *tok* has type lab while *access* expects type $\mathsf{lab}{\sim}USER(k)$. Since the call to *access* occurs in the first branch of the match, the context includes the refinement $tok \succ USER(k)$ due to (T-MATCH). From (T-SHOW) we can give *tok* type $\mathsf{lab}{\sim}tok$, and by applying (TE-REFINE) we have $\mathsf{lab}{\sim}tok \cong \mathsf{lab}{\sim}USER(k)$ and so *tok* can be given type $\mathsf{lab}{\sim}USER(k)$ as required. Similarly, for *access tok cap*, we can check that the type $\mathsf{int}\{tok\}$ of *cap* is convertible with $\mathsf{int}\{USER(k)\}$ in the presence of the same assumption.

Rule (TE-REDUCE) allows FABLE types to be considered convertible if the expression component of one is reducible to the expression component of the other [4]; reduction $e \stackrel{c}{\rightsquigarrow} e'$ is defined shortly in Figure 2.4. For example, we have $\cdot \vdash int\{(\lambda x{:}\mathsf{lab}.x)\ Low\} \cong int\{Low\}$ since $(\lambda x{:}\mathsf{lab}.x)\ Low \stackrel{c}{\rightsquigarrow} Low$. One complication is that type-level expressions may contain free variables. For example, suppose we wanted to show

$$y : \mathsf{lab} \vdash int\{(\lambda x{:}\mathsf{lab}.x)\ y\} \cong int\{y\}$$

It seems intuitive that these types should be considered convertible, but we do not have that $(\lambda x{:}\mathsf{lab}.x)\ y \stackrel{c}{\rightsquigarrow} y$ because y is not a value. To handle this case, the rule permits two types to be convertible if, for every well-typed substitution σ of the free variables of e_1, $\sigma(e_1) \stackrel{c}{\rightsquigarrow} \sigma(e_2)$. This captures the idea that the precise value of y is immaterial—all reductions on well-typed substitutions of y would reduce to the value that was substituted for y.

Satisfying this obligation by exhaustively considering all possible substitutions is obviously intractable. Additionally, we have no guarantee that an expression appearing in a type will converge to a value. Thus, type checking in FABLE, as presented here, is undecidable. This is not uncommon in a dependent type system; e.g., type checking in Cayenne is undecidable [6]. However, other dependently typed systems impose restrictions on the usage of recursion in type-level expressions to ensure that type-level terms always terminate [17]. Additionally, there are several possible decision procedures that can be used to partially decide type convertibility. One simplification would be to attempt to show convertibility for closed types only, i.e., no free variables. In SELINKS, our implementation of FABLE, we use a combination of three techniques. First, we use type information. If l is free in a type, and the declared type of l is $\mathsf{lab} \sim e$, then we can use this information to substitute e for l. Similarly, if the type context includes an assumption of the form $l \succ e$ (when checking the branch of a pattern), we can substitute l with e. Finally, since type-level expressions typically manipulate labels by pattern matching, we use a simple heuristic to determine which branch to take when pattern matching expressions with free variables. These techniques suffice for all the examples in this chapter and both our SEWIKI and SEWINESTORE applications. A related technical report [123] discusses these decision procedures in greater detail and proves them sound.

Finally, the judgment $\Gamma \vdash t$ states that t is well-formed in Γ. Rules (K-INT), (K-TVAR), and (K-LAB) are standard, (K-FUN) defines the standard scoping rules for names in dependent function types, and (K-ALL) defines the standard scoping rule for universally quantified type variables. (K-SLAB) and (K-LABT) ensure that all expressions e that appear in types can be given lab-type. Notice that type-level expressions are typed in

Evaluation contexts E_c $::=$ $\bullet\, e \mid v_c \bullet \mid \bullet\,[t] \mid C(\vec{v}_c, \bullet, \vec{e})$
                        $\mid$ $\mathsf{match}\, \bullet \,\mathsf{with}\, p_i \Rightarrow e_i \mid \{e\}\bullet \mid \{\circ\}\bullet$

$$\frac{e \stackrel{c}{\leadsto} e'}{E_c \cdot e \stackrel{c}{\leadsto} E_c \cdot e'} \text{ (E-CTX)} \qquad \frac{e \stackrel{pol}{\leadsto} e'}{(\!|e|\!) \stackrel{app}{\leadsto} (\!|e'|\!)} \text{ (E-POL)} \qquad (\lambda x{:}t.e)\, v_c \stackrel{c}{\leadsto} (x \mapsto v_c)e \text{ (E-APP)}$$

$$(\Lambda\alpha.e)\,[t] \stackrel{c}{\leadsto} (\alpha \mapsto t)e \text{ (E-TAP)} \qquad \mathsf{fix}\, f{:}t.v \stackrel{c}{\leadsto} (f \mapsto \mathsf{fix}\, f{:}t.v)v \text{ (E-FIX)}$$

$$\frac{\forall i < j.\, v_c \not\succ p_i : \sigma_i \qquad v_c \succ p_j : \sigma_j}{\mathsf{match}\, v_c \,\mathsf{with}\, p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n \stackrel{c}{\leadsto} \sigma_j(e_j)} \text{ (E-MATCH)}$$

$$(\!|C(\vec{u})|\!) \stackrel{app}{\leadsto} C(\vec{u}) \text{ (E-BLAB)} \quad (\!|n|\!) \stackrel{app}{\leadsto} n \text{ (E-BINT)} \quad (\!|\lambda x{:}t.e|\!) \stackrel{app}{\leadsto} \lambda x{:}t.(\!|e|\!) \text{ (E-BABS)}$$

$$(\!|\Lambda\alpha.e|\!) \stackrel{app}{\leadsto} \Lambda\alpha.(\!|e|\!) \text{ (E-BTAB)} \quad (\!|e|\!) \stackrel{pol}{\leadsto} e \text{ (E-NEST)} \quad \{\circ\}\{e\}v_{pol} \stackrel{pol}{\leadsto} v_{pol} \text{ (E-UNLAB)}$$

$$p \succ p : \cdot \text{ (U-PATID)} \quad v \succ x : x \mapsto v \text{ (U-VAR)} \quad \frac{\forall i.\sigma_i^* = (\sigma_0, \ldots, \sigma_{i-1}) \qquad e_i \succ \sigma_i^* p_i : \sigma_i}{C(\vec{e}) \succ C(\vec{p}) : \vec{\sigma}} \text{ (U-CON)}$$

Figure 2.5: Dynamic semantics of FABLE

pol-context. Because FABLE enjoys a type-erasure property, any (un)labeling operations appearing in types pose no security risk. We use this feature to good effect in Section 2.2.2 to protect sensitive information that may appear in labels.

2.1.4   Operational Semantics

Figure 2.5 defines FABLE's operational semantics. We define a pair of small-step reduction relations $e \stackrel{app}{\leadsto} e'$ and $e \stackrel{pol}{\leadsto} e'$ for application and policy expressions, respectively. Rules of the form $e \stackrel{c}{\leadsto} e'$ are *polychromatic*—they apply both to policy and application expressions. Since the values for each kind of expression are different, we also parameterize the evaluation contexts E_c by the color of the expression, i.e., the context, either *app*

or *pol*, in which the expression is to be reduced. Rule (E-CTX) uses these evaluation contexts E_c, similar to the type contexts used above, to enforce a left-to-right evaluation order for a call-by-value semantics. (In the context of FABLE, which is purely functional, the call-by-value restriction is unnecessary. However, in subsequent chapters, a call-by-value semantics is important.) Policy expression reduction $e \overset{pol}{\leadsto} e'$ takes place within brackets according to (E-POL). The rules (E-APP), (E-TAP), and (E-FIX) define function application, type application, and fixed-point expansion, respectively, in terms of substitutions; all of these are standard. Rule (E-MATCH) relies on a standard pattern-matching judgment $v \succ p : \sigma$, also defined in Figure 2.5, which is true when the label value matches the pattern such that $v = \sigma(p)$. (E-MATCH) determines the first pattern p_j that matches the expression v and reduces the match expression to the matched branch's body after applying the substitution. The (U-CON) rule in the pattern-matching judgment $v \succ p : \sigma$ is the only non-trivial rule. As explained in Section 2.1.3, since pattern variables may occur more than once in a pattern, (U-CON) must propagate the result of matching earlier sub-expressions when matching subsequent sub-expressions. For example, pattern matching should fail when attempting to match $ACL(Alice, Bob)$ with $ACL(x, x)$. This is achieved in (U-CON) because, after matching $(Alice \succ x : x \mapsto Alice)$ using (U-VAR), we must try to match Bob with $(x \mapsto Alice)x$, which is impossible.

An applied policy function will eventually reduce to a bracketed policy value v_{pol}. When v_{pol} has the form $(\!|u|\!)$, the brackets may be removed so that the value u can be used by application code. (E-BLAB) and (E-BINT) handle label expressions $(\!|C(\vec{u})|\!)$ and integers n, respectively. To maintain the invariant that (un)labeling operators only appear in policy code, rules (E-BABS) and (E-TABS) extrude only the λ and Λ binders, respectively, from

bracketed abstractions, allowing them to be reduced according to (E-APP) or (E-TAP). Brackets cannot be removed from labeled values $(\llbracket \{e\}u \rrbracket)$ by application code, to preserve the labeling invariant. On the other hand, brackets can be removed from any expression by policy code, according to (E-NEST). This is useful when reducing expressions such as $(\llbracket \lambda x{:}t.x \rrbracket)(\llbracket v \rrbracket)$, which produces $(\llbracket (\llbracket v \rrbracket) \rrbracket)$ after two steps; (E-NEST) (in combination with (E-POL)) can then remove the inner brackets. Finally, (E-UNLAB) allows an unlabeling operation to annihilate the top-most relabeling operation. Notice that the expressions within a relabeling operation are never evaluated at runtime—relabelings only affect the types and are purely compile time entities. The types that appear elsewhere, such as (E-TAP), are also erasable, as is usual for System F.

2.1.5 Soundness

We state the standard type soundness theorems for FABLE here. In addition to ensuring that well-typed programs never go wrong or get stuck, we have put this soundness result to good use in proving that security policies encoded in FABLE satisfy desirable security properties. We discuss this further in the next section. Appendix A contains a full statement and proof of this theorem.

Theorem 1 (Type soundness). *If $\cdot \vdash_c e : t$; then either $\exists e'.e \overset{c}{\rightsquigarrow} e'$ or $\exists v_c.e = v_c$. Furthermore, if $e \overset{c}{\rightsquigarrow} e'$; then, $\cdot \vdash_c e' : t$.*

2.2 Example Policies in FABLE

This section uses FABLE to encode several security policies. We prove that any well-typed program using one of these policies enjoys relevant security properties—i.e., the program is sure to enforce the policy correctly. We focus on four kinds of policies: access control, provenance, static information flow, and dynamic information flow.

As mentioned in the introduction, FABLE does not, in and of itself, guarantee that well-typed programs implement a particular security policy's semantics correctly. That said, FABLE has been designed to facilitate proof of such theorems. To illustrate how, we chose to use three very different techniques for each of the correctness results reported here. We conclude from our experience that the metatheory of FABLE provides a useful repository of lemmas that can naturally be applied in showing the correctness of various policy encodings. As such, we believe the task of constructing a correctness proof for a FABLE policy to be no more onerous, and possibly considerably simpler, than the corresponding task for a special-purpose calculus that "bakes in" the enforcement of a single security policy.

2.2.1 Access Control Policies

Access control policies govern how programs release information but, once the information is released, do not control how it is used. To prove that an access control policy is implemented correctly, we must show that programs not authorized to access some information cannot learn the information in any way, e.g., by bypassing a policy check (something not uncommon in production systems [114]) or by exploiting leaks due

to control-flow or timing channels. We call this security condition *non-observability*.

Intuitively, we can state non-observability as follows. If some program P is not allowed to access a resource v_1 having a label l, then a program P' that is identical to P except that v_1 has been replaced with some other resource v_2 (having the same type and label as v_1) should evaluate in the same way as P—it should produce the same result and take the same steps along the way toward producing that result. If this were not true then, assuming P's reduction is deterministic, P must be inferring information about the protected resource.

To make this intuition formal, we will show that the evaluations of programs P and P' are *bisimilar*, where the only difference between them is the value of the protected resource. To express this, first we define an equivalence relation called *similarity up to l* (analogous to definitions of low equivalence [111, 26]) which holds for two terms e and e' if they only differ in sub-terms that are labeled with l, with the intention that l is the label of restricted resources.

Definition 2 (Similarity up to l)**.** *Expressions e and e′, identified up to α-renaming, are similar up to label l according to the relation $e_1 \sim_l e_2$ shown in Figure 2.6.*

The most important rule in Figure 2.6 is (SIM-L), which states that arbitrary expressions e and e' are considered similar at label l when both are labeled with l. Other parts of the program must be structurally identical, as stated by the remaining congruence rules. We extend similarity to a bisimulation as follows: two similar terms are bisimilar if they always reduce to similar subterms, and do so indefinitely or until no further reduction is possible. This notion of bisimulation is the basis of our access control security theorem;

$$e \sim_l e \text{ (SIM-ID)} \qquad \{l\}e \sim_l \{l\}e' \text{ (SIM-L)} \qquad \frac{e \sim_l e' \quad l' \neq l}{\{l'\}e \sim_l \{l'\}e'} \text{ (SIM-L2)}$$

$$\frac{e \sim_l e'}{\lambda x{:}t.e \sim_l \lambda x{:}t.e'} \text{ (SIM-ABS)} \qquad \frac{e_1 \sim_l e_1' \quad e_2 \sim_l e_2'}{e_1 \, e_2 \sim_l e_1' \, e_2'} \text{ (SIM-APP)}$$

$$\frac{v \sim_l v'}{\text{fix } f{:}t.v \sim_l \text{fix } f{:}t.v'} \text{ (SIM-FIX)} \qquad \frac{e \sim_l e'}{\Lambda \alpha.e \sim_l \Lambda \alpha.e'} \text{ (SIM-TAB)}$$

$$\frac{e \sim_l e' \quad t \sim_l t'}{e[t] \sim_l e[t']} \text{ (SIM-TAP)} \qquad \frac{\forall i.e_i \sim_l e_i'}{C(\vec{e}) \sim_l C(\vec{e'})} \text{ (SIM-LAB)}$$

$$\frac{e \sim_l e' \quad e_i \sim_l f_i \quad p_i \sim_l q_i}{\text{match } e \text{ with } p_i \rightarrow e_i \sim_l \text{match } e' \text{ with } q_i \rightarrow f_i} \text{ (SIM-MATCH)} \qquad \frac{e \sim_l e'}{(\![e]\!) \sim_l (\![e']\!)} \text{ (SIM-POL)}$$

Figure 2.6: Similarity of expressions under the access control policy

it is both *timing and termination sensitive*.

Definition 3 (Bisimulation). *Expressions e_1 and e_2 are bisimilar at label l, written $e_1 \approx_l e_2$, if and only if $e_1 \sim_l e_2$ and for $\{i, j\} = \{1, 2\}$, $e_i \overset{c}{\rightsquigarrow} e_i' \Rightarrow e_j \overset{c}{\rightsquigarrow} e_j'$ and $e_1' \approx_l e_2'$.*

Theorem (Non-observability). *Given all of the following:*

1. *A $(\![\cdot]\!)$-free expression e.*

2. *$(a{:}t_a, m{:}t_m, cap{:}\text{int}\{user\}, x{:}t\{acl\} \vdash_{app} e : t_e)$ where acl and user are label constants.*

3. *A type-respecting substitution $\sigma = (a \mapsto access, m \mapsto member, cap \mapsto (\![\{user\}0]\!))$.*

4. *Type-respecting substitutions $\sigma_i = \sigma, x \mapsto v_i$ where $\cdot \vdash_{app} v_i : t\{acl\}$ for $i = 1, 2$.*

   Then, we have (member user acl $\overset{c*}{\rightsquigarrow}$ False) $\Rightarrow \sigma_1(e) \approx_{acl} \sigma_2(e)$.

This theorem is concerned with a program e that contains no policy-bracketed terms

(it is just application code) but, via the substitution σ, may refer to our access control func-

34

tions *access* and *member* (defined in Figure 2.3) through the free variables *a* and *m*. Additionally, the program is granted a single user capability ($\{user\}0$) through the free variable *cap*, which gives the program the authority of user *user*. The program may also refer to some protected resource *x* whose label is *acl*, but the authority of *user* is insufficient to access *x* according to the access control policy because (*member user acl $\overset{c_*}{\leadsto}$ False*). Under these conditions, we can show that for any two (well-typed) v_i we substitute for *x* according to substitution σ_i, the resulting programs are bisimilar—their reduction is independent of the choice of v_i.

In all our proofs, two key features of FABLE play a central role. First, dependent typing in FABLE allows a policy analyst to assume that all policy checks are performed correctly. For instance, when calling the *access* function to access a value *v* of type $t\{acl\}$, the label expressing *v*'s security policy must be *acl*, and no other. The type system ensures that the application program cannot construct a label, say *ACL(Public)*, and trick the policy into believing that this label, and not *acl*, protects *v*, i.e., dependent typing rules out *confused deputies* [18]. Second, the restriction that application code cannot directly inspect labeled resources ensures that a policy function must mediate every access of a protected resource. Assuring complete mediation is not unique to FABLE— Zhang et al. [148] used CQual to check that SELinux operations on sensitive objects are always preceded by policy checks and Fraser [53] did the same for Minix. However, the analysis in both these instances only ensures that *some* policy check has taken place, not necessarily the correct one. As such, these other techniques are vulnerable to flaws due to confused deputies.

When combined with these two insights, our proof of non-observability for the ac-

cess control policy is particularly simple. In essence, the FABLE system ensures that a value with labeled type must be treated abstractly by the application program. With this observation, the proof proceeds in a manner very similar to a proof of value abstraction [58]. This is a general semantic property for languages like FABLE that support parametric polymorphism or abstract types. Indeed, the policy as presented in Figure 2.3 could have been implemented in a language like ML, which also has these features. For instance, an integer labeled with an access control list could be represented in ML as a pair consisting of an access control list and an integer with type (string list × int). A policy module could export this pair as an abstract type, preventing application code from ever inspecting the value directly, and provide a function to expose the concrete type only after a successful policy check.

While such an encoding using ML's module system would suffice for the simple policy of Figure 2.3, it would not work for more sophisticated models of access control. For example, a form of access control using capabilities can be easily encoded in FABLE. Such a model could provide access to more than one resource with a single membership test, as in the following code

$$\text{policy } access\_cap\langle k\rangle(u{:}\mathsf{lab}{\sim}\mathit{USER}(k), \mathit{cred}{:}\mathsf{int}\{u\}, \mathit{acl}{:}\mathsf{lab}) =$$
$$\text{match } member\ u\ acl \text{ with } \mathit{True} \Rightarrow \Lambda\alpha.\lambda x{:}\alpha\{acl\}.\{\circ\}x$$
$$\_ \Rightarrow \mathit{\#fail}$$

Here the caller presents a user credential and an access control label *acl* but no resource labeled with that label. If the membership check succeeds, a function with type $\forall\alpha.\alpha\{acl\} \rightarrow \alpha$ is returned. This function can be used to immediately unlabel any resource with the authorized label, i.e., the function is a kind of key that can be used to gain access to a protected resource. This is useful when policy queries are expensive. It is

also useful for encoding a form of delegation; rather than releasing his user credential, a user could release a function that uses that credential to a limited effect. Of course, this may be undesirable if the policy is known to change frequently, but even this could be accommodated. Variations that combine static and dynamic checks are also possible.

Finally, notice that this theorem is indifferent to the actual implementation of the *acl* label and the *member* function. Thus, while our example policy is fairly simplistic, a far more sophisticated model could be used. For instance, we could have chosen labels to stand for RBAC- or RT-style roles [76], and *member* could invoke a decision procedure for determining role membership. Likewise, the theorem is not concerned with the origin of the *user* authentication token—a function more sophisticated than *login* (e.g., that relied on cryptography) could have been used. The important point is that FABLE ensures the second component of the user credential (l:lab $\sim USER(k) \times$ int$\{l\}$) is unforgeable by application code.

2.2.2 Dynamic Provenance Tracking

Provenance is "information recording the source, derivation, or history of some information" [26]. Provenance is relevant to computer security for at least two reasons. First, provenance is useful for auditing, e.g., to discover whether some data was inappropriately released or modified. Second, provenance can be used to establish data integrity, e.g., by carefully accounting for a document's sources. This section describes a label-based provenance tracking policy we constructed in FABLE. To prove that this policy is implemented correctly we show that all programs that use it will accurately capture the

dependences (in the sense of information flow) on a value produced by a computation.

Figure 2.7 presents the provenance policy. We define the type *Prov α* to describe a pair in which the first component is a label *l* that records the provenance of the second component. The policy is agnostic to the actual form of *l*. Provenance labels could represent things like authorship, ownership, the channel on which information was received, etc. An interesting aspect of *Prov α* is that the provenance label is itself labeled with the 0-ary label constant *Auditors*. This represents the fact that provenance information is subject to security concerns like confidentiality and integrity. Intuitively, one can think of data labeled with the *Auditors* label as only accessible to members of a group called *Auditors*, e.g., as mediated by the access control policy of Figure 2.3; of course, a more complex policy could be used. Finally, note that because the provenance label *l* is itself labeled (having type lab{*Auditors*}), it would be incorrect to write $\alpha\{l\}$ as the second component of the type since this requires that *l* have type lab. Therefore we unlabel *l* when it appears in the type of the second component. As explained in Section 2.1.3, unlabeling operations in types pose no security risk since the types are erased at runtime.

The policy function *apply* is a wrapper for tracking dependences through function applications. In an idealized language like FABLE it is sufficient to limit our attention to function application, but a policy for a full language would define wrappers for other constructs as well. The first argument of *apply* is a provenance-labeled function *lf* to be called on the second argument *mx*. The body of *apply* first decomposes the pair *lf* into its label *l* and the function *f* itself and does likewise for the argument *mx*. Then it applies the function, stripping the label from both it and its argument first. The provenance of the result is a combination of the provenance of the function and its argument. We write this

```
typename Prov α= (l:lab{Auditors} × α{{∘}l})

policy apply⟨α,β⟩ (lf:Prov (α → β), mx:Prov α) =
    let l,f = lf in
    let m,x = mx in
    let y = ({∘}f) ({∘}x) in
    let lm = Union({∘}l, {∘}m) in
      ({Auditors}lm, {lm}y)

policy flatten⟨α⟩ (x:Prov (Prov α)) =
    let l,inner = x in
    let m,a = inner in
    let lm = Union({∘}l, {∘}m) in
      ({Auditors}lm, {lm}a)
```

Figure 2.7: Enforcing a dynamic provenance-tracking policy

as the label pair $Union(l,m)$ which is then associated with the final result. Notice that we

strip the *Auditors* labels from labels l and m before combining them, and then relabel the

combined result.

The policy also defines a function *flatten* to convert a value of type *Prov* (*Prov* α) to

one of type *Prov* α by extracting the nested labels (the first two lines) and then collapsing

them into a *Union* (third line) that is associated with the inner pair's labeled component

(fourth line).

An example client program that uses this provenance policy is the following:

```
let client⟨α,β,γ⟩ (f : Prov(α → β→ γ), x : Prov α, y : Prov β) =
    apply [β][γ] (apply [α][β → γ] f x) y
```

This function takes a labeled two-argument function f as its argument and the two argu-

ments x and y. It calls *apply* twice to get a result of type *Prov* γ. This will be a tuple in

which the first component is a labeled provenance label of the form $Union(Union(lf,lx), ly)$

and the second component is a value labeled with that provenance label. In the label, we

will have that *lf* is the provenance label of the function argument f and *lx* and *ly* are the

$$[\![C]\!] \stackrel{\text{def}}{=} \{C\} \qquad [\![Union(l_1, l_2)]\!] \stackrel{\text{def}}{=} [\![l_1]\!] \cup [\![l_2]\!]$$

$t \leq t'$ Prefixing relation on types

$$t \leq t \qquad\qquad \frac{t \leq t'}{t \leq t'\{e\}}$$

$e \approx_l e' : t,t'$ e and e' are related by provenance p at types t and t'

$$\frac{\begin{array}{c} i \in \{1,2\} \quad \cdot \vdash_c v_i : t_i \quad t_i'\{e_i\} \leq t_i \quad t \leq t_i \quad e_i \stackrel{pol_*}{\rightsquigarrow} v_i^{lab} \\ l \in [\![v_1^{lab}]\!] \cap [\![v_2^{lab}]\!] \qquad \vee \qquad Auditors \in [\![v_1^{lab}]\!] \cap [\![v_2^{lab}]\!] \end{array}}{v_1 \approx_l v_2 : t_1, t_2} \text{ (R-EQUIVP)}$$

$$n \approx_l n : \text{int}, \text{int} \text{ (R-INT)} \qquad \frac{\begin{array}{c} i \in \{1,2\} \qquad \cdot \vdash_c e_i : t_i \\ e_i \stackrel{c_*}{\rightsquigarrow} v_i \quad \Rightarrow \quad v_1 \approx_{[l} v_2 : t_1, t_2 \end{array}}{e_1 \approx_l e_2 : t_1, t_2} \text{ (R-EXPR)}$$

$$\frac{\begin{array}{c} \cdot \vdash_c v : (x{:}t_1) \rightarrow t_2 \qquad\qquad \cdot \vdash_c v' : (x{:}t_1') \rightarrow t_2' \\ \forall v_1, v_1'. \ v_1 \approx_l v_1' : t_1, t_1' \quad \Rightarrow \quad v v_1 \approx_l v' v_1' : (x \mapsto v_1)t_2, (x \mapsto v_1')t_2' \end{array}}{v \approx_l v' : (x{:}t_1) \rightarrow t_2, (x{:}t_1') \rightarrow t_2'} \text{ (R-ABS)} \qquad \dots$$

Figure 2.8: A logical relation that relates terms of similar provenance (selected rules)

provenance of the arguments x and y, respectively. Note that a caller of *client* can instantiate the type variable γ to be a type like *Prov* int. In this case, the type of the returned value will be *Prov* (*Prov* int), which can be flattened if necessary.

We can prove that provenance information is tracked correctly following Cheney et al. [26]. The intention is that if a value x of type *Prov* α influences the computation of some other value y, then y must have type *Prov* β (for some β) and its provenance label must mention the provenance label of x. If provenance is tracked correctly, a change to x will only affect values like y; other values in the program will be unchanged.

The essence of this correctness condition is much like the similarity relation $v_1 \sim_l v_2$

defined for the non-observability property in Section 2.2.1. However, there is a difference between provenance tracking and access control that complicates the statement of the correctness condition for provenance. When a computation e that depends on the value of some variable x is reduced in two different contexts for x, the intermediate terms that are produced in one context can be entirely different from the terms that are produced in the other context. That is, if we have $e(x \mapsto v_1) \overset{c}{\leadsto} e_1 \overset{c}{\leadsto} \ldots$ and $e(x \mapsto v_2) \overset{c}{\leadsto} e'_1 \overset{c}{\leadsto} \ldots$; then the terms e'_1 and e'_2 may not even have the same shape, which makes it difficult to state a purely syntactic similarity condition between the terms. However, the provenance tracking policy ensures that if both reduction sequences terminate with a value, then the corresponding values are labeled with the appropriate provenance label. In contrast, although non-observability for access control also applies to programs e that are reduced in different contexts, we can assure that at each step the terms that are produced are identical, except for holes in the terms that contain the access-protected values of x.

Our formulation of dependency correctness follows technique that is due to Tse and Zdancewic [127] (although Tse and Zdancewic use this technique to show noninterference in the presence of a form of dynamic labeling). This approach involves defining a logical relation [85] that relates terms whose set of provenance labels include the same label l. Figure 2.8 shows a selection of rules in this relation. (The full relation can be found in Section A.3.)

The top of the figure begins by giving a semantics for label values in terms of sets, $[\![e]\!]$. The relation $t \leq t'$ is a prefixing relation on types, which is convenient for constraining the shape of types in the main relation $e_1 \approx_l e_2 : t_1, t_2$. This latter relation, states that expressions e_1 and e_2 are related by provenance label l, and can be given types

41

t_1 and t_2, respectively.

The key rule in the relation is (R-EQUIVP). It states that two arbitrary values v_1 and v_2 are related at the label l, if they both have labeled types t_1 and t_2 (the third premise) and if these types share a common prefix t (the fourth premise). The constraints on the labels on these types require that related values be labeled with the provenance label l. In the fifth premise, we require that some label e_i on each type reduce to a label value v_i^{lab}—since we are only concerned with terminating computations, we can safely ignore the case where the label expression diverges. The last premise is a disjunct in which the first clause requires the provenance label l to be mentioned in the labels of both expressions—notice that the labels do not have to be identical; the sets represented by each label just have to contain l. The second clause in the disjunct handles an important corner case. Since our encoding uses dynamic provenance labels that are themselves always protected with an access control policy, and because the way in which these labels are constructed can depend on the other values in the program, we treat all provenance labels (those terms that are protected by the label *Auditors*) as being related.

The remaining rules in Figure 2.8 are standard and give a flavor of the elided rules. (R-INT) states that identical integers are related. (R-EXPR) states that expressions e_1 and e_2 are related if their normal forms v_1 and v_2 (if these exist) are related. (R-ABS) relates function-typed values if these functions reduce to related values when they are applied to related arguments.

Theorem (Dependency correctness). *Given all of the following:*

(A1) A $(\![\cdot]\!)$-free expression e such that $a{:}t_a, f{:}t_f, x{:}Prov\ t \vdash_{app} e : t'$,

(A2) A type-respecting substitution $\sigma = (a \mapsto apply, f \mapsto flatten)$.

(A3) $\vdash_{app} v_i : Prov\ t,\ for\ i = 1, 2\ and\ v_1 \approx_l v_2 : Prov\ t, Prov\ t$

(A4) For $i \in \{1, 2\}$, $\sigma_i = \sigma, x \mapsto v_i$

Then, $(\sigma_1(e) \overset{app_*}{\leadsto} v_1' \ \wedge \ \sigma_2(e) \overset{app_*}{\leadsto} v_2') \Rightarrow v_1' \approx_l v_2' : \sigma_1 t', \sigma_2 t'$.

Intuitively, this theorem states that an application program e that is compiled with the policy of Figure 2.7 and is executed in contexts that differ only in the choice of a tracked value of label l will compute results that differ only in sub-terms that are also colored using l. The crux of this proof involves showing that the logical relation is preserved under substitution, i.e., a form of substitution lemma for the logical relation. While constructing the infrastructure to define the logical relation requires some work, strategic applications of standard substitution lemma for FABLE can be used to discharge the proof without much difficulty.

2.2.3 Static Information Flow

Both policies discussed so far rely on runtime checks. This section illustrates how FABLE can be used to encode *static* lattice-based information flow policies that require no runtime checks. In a static information flow type system (as found in FlowCaml [111]) labels l have no run-time witness; they only appear in types $t\{l\}$. Labels are ordered by a relation $\sqsubseteq$ that typically forms a lattice. This ordering is lifted to a subtyping relation on labeled types such that $l_1 \sqsubseteq l_2 \Rightarrow t\{l_1\} <: t\{l_2\}$. Assuming the lattice ordering is fixed during execution, well-typed programs can be proven to adhere to the policy defined by the initial label assignment appearing in the types.

```
policy lub(x:lab, y:lab) = match x,y with
    _, HIGH | HIGH, _ ⇒ HIGH
    | _, _ ⇒ LOW
policy join⟨α,l,m⟩ (x:α{l}{m}) = ({lub l m}{∘}{∘}x)
policy sub⟨α,l⟩ (x:α{l}, m:lab) = ({lub l m}{∘}x)
policy apply⟨α,β,l,m⟩ (f:(α → β){l}, x:α) = {l}(({∘}f) x)

let client (f:(int{HIGH} → int{HIGH}){LOW}, x:int{LOW}) =
    let x = (sub [int] x HIGH) in
        join [int] (apply [int{HIGH}][int{HIGH}] f x)
```

Figure 2.9: Enforcing an information flow policy

Figure 2.9 illustrates the policy functions, along with a small sample program. In our encoding we define a two-point security lattice with atomic labels *HIGH* and *LOW* and protected expressions will have labeled types like $t\{HIGH\}$. The ordering $LOW \sqsubset HIGH$ is exemplified by the *lub* (least upper bound) operation for the lattice. The *join* function (similar to the *flatten* function from Figure 2.7) combines multiple labels on a type into a single label. The interesting thing here is the label attached to x is a label expression *lub l m*, rather than an label value like *HIGH*. The type rule (T-CONV) presented in Figure 2.4 can be used to show that a term with type int$\{lub\ HIGH\ LOW\}$ can be given type int$\{HIGH\}$ (since $lub\ HIGH\ LOW \overset{c}{\leadsto} HIGH$). This is critical to being able to type programs that use this policy.

The policy includes a subsumption function *sub*, which takes as arguments a term x with type $\alpha\{l\}$ and a label m and allows x to be used at the type $\alpha\{lub\ l\ m\}$. This is a restatement of the subsumption rule above, as $l \sqsubseteq m$ implies $l \sqcup m = m$. (Once types are erased, *join* and *sub* are both essentially the identity function and could be optimized away.) Finally, the policy function *apply* unlabels the function f in order to call it, and then adds f's label on the computed result.

Consider the client program at the bottom of Figure 2.9 as an example usage of the static information flow policy. The function *client* calls the function f with x, where f expects a parameter of type int$\{HIGH\}$ while x has type int$\{LOW\}$. For the call to type check, the program uses *sub* to coerce x's type to int$\{lub\ LOW\ HIGH\}$ which is convertible to int$\{HIGH\}$. The call to *apply* returns a value of type int$\{HIGH\}\{LOW\}$. The call to *join* collapses the pair of labels so that *client*'s return type is int$\{lub\ HIGH\ LOW\}$, which converts to int$\{HIGH\}$.

We have proved that FABLE programs using this policy enjoy the standard non-interference property—a statement of this theorem appears below. We have also shown that a FABLE static information flow policy is at least as permissive as the information flow policy implemented by the functional subset of Core-ML, the formal language of FlowCaml [104]. Both proofs may be found in Appendix A.

Theorem (Noninterference). *Given* $\vec{p} : \vec{t}, x : t\{HIGH\} \vdash_c e : t'\{LOW\}$, *where e is* $(\![]\!)$*-free and t' is not a labeled type; and, for $i = 1, 2$,* $\cdot \vdash_c v_i : t\{HIGH\}$. *Then, for type-respecting substitutions* $\sigma_i = (\vec{p} \mapsto \pi, x \mapsto v_i)$, *where π is the policy of Figure 2.9,* $\sigma_1(e) \stackrel{c_*}{\leadsto} v'_1 \wedge \sigma_2(e) \stackrel{c_*}{\leadsto} v'_2 \Rightarrow v'_1 = v'_2$.

While it would be possible to reuse our infrastructure for the dependency correctness proof to show the noninterference result for the static information flow policy (as in Tse and Zdancewic), we choose instead to use another technique, due to Pottier and Simonet [104]. This technique involves representing a pair of executions of a FABLE program within the syntax of a single program and showing a subject reduction property holds true. As with the logical relations proof, once we had constructed the infrastruc-

ture to use this technique, the proof was an easy consequence of FABLE's preservation theorem.

2.2.4 Dynamic Information Flow

Realistic information flow policies are rarely as simple as that of Section 2.2.3. For example, the security label of some data may not be known until run-time, and the label itself may be more complex than a simple atom, e.g., it might be drawn from the DLM [89] or some other higher-level policy language, such as RT [125]. Figure 2.10 shows how dynamic security labels can be associated with the data and an information flow policy enforced using a combination of static and dynamic checks [149].

The label lattice is defined by the external *oracle* function. The enforcement policy interfaces with the oracle through the function *flow*, which expects two labels *src* and *dest* as arguments and determines whether the oracle permits information to flow from *src* to *dest*. The representation of these labels is abstract in the policy and depends on the implementation of the *oracle*. The *flow* function is given the type

$$(src{:}\mathsf{lab}) \rightarrow (dest{:}\mathsf{lab}) \rightarrow (l{:}\mathsf{lab} \times \mathsf{unit}\{l\})$$

If the *oracle* permits the flow, the *flow* function returns a capability similar to that provided by the *login* function of Figure 2.3. The *sub* function takes this capability as its first argument as proof that type $\alpha\{src\}$ may be coerced to type $\alpha\{dest\}$. The *low* function must appeal to the oracle to acquire the bottom label in the lattice. The *app* function is analogous to the *apply* function in the static information flow policy. It takes a

46

policy *flow*(*src*:lab, *dest*:lab) =
          let f = if *oracle src dest* then
                   FLOW(*src*,*dest*)
               else *NOFLOW* in
            (f, $\{f\}()$)

policy *low*⟨α⟩ (*x*:α) = let l = *oracle_low*() in (l, $\{l\}x$)

policy *sub*⟨α,*src*,*dest*⟩(*cap*:unit$\{FLOW(src,dest)\}$, *x*:$\alpha\{src\}$) = $\{dest\}x$

policy *app*⟨α,β,*l*,*m*⟩(*f*:($\alpha \to \beta$)$\{l\}$, *x*:$\alpha\{m\}$) = $\{JOIN(l, m)\}$ ($\{\circ\}f$) ($\{\circ\}x$)

---

let *client*⟨α⟩(*lb*:lab, *b*:bool$\{lb\}$, *lx*:lab, *x*:$\alpha\{lx\}$, *ly*:lab, *y*:$\alpha\{ly\}$) =
         let *lxy* = *JOIN*(*lx*,*ly*) in
         let *fx*,*capx* = *flow lx lxy* in
         let *fy*,*capy*= *flow ly lxy* in
           match *fx*,*fy* with
             FLOW(*lx*,*lxy*), *FLOW*(*ly*,*lxy*) $\to$
               let *x'* = *sub* [α] *capx x* in
               let *y'* = *sub* [α] *capy y* in
               let *tmp* = *app* [α] [$\alpha \to \alpha$] (*b*[α]) *x'* in
                app [α] [α] *tmp y'*
           _ , _ $\to$ ... #*flow must be allowed if oracle is a lattice*

Figure 2.10: A dynamic information flow policy and a client that uses it

labeled function *f* and its argument *x* as parameters. In the body, it unlabels *f* and applies

it to *x* (after unlabeling *x* also). Since the returned value depends both on the function and

the argument, we label it with the labels of both *f* and *x*.

The bottom part of Figure 2.10 shows a client program that illustrates a usage of this

policy. This client program has the same high-level behavior as the example program we

showed for the static information flow policy—it branches on a boolean and returns either

x or *y*—but here the security labels of the arguments are not statically known. Instead, the

argument *lb* is a label term that specifies the security level of *b*, and similarly *lx* for *x* and

ly for *y*. As previously, our encoding of booleans requires each branch to have the same

type, including the security label. In this case, the program arranges the branches to have the type *JOIN(lx,ly)*. The first three lines of the main expression use the flow function to attempt to obtain capabilities that witness the flow from *lx* and *ly* to *JOIN(lx,ly)*. The match inspects the labels that are returned by *flow* and in case where they are actually *FLOW(...)* the final premise of (T-MATCH) permits the type of *fx* to be refined from lab∼*fx* to lab∼*FLOW(lx,lxy)* and the type of *capx* to be refined to unit{*FLOW(lx,lxy)*}, and similarly for *capy*. The remainder of the program is similar to the static case, but requires more uses of subsumption since less is known statically about the labels. The type of this program is:

$$\forall \alpha. (lb:\text{lab}) \rightarrow \beta \text{bool}\{lb\} \rightarrow (lx:\text{lab}) \rightarrow \alpha\{lx\} \rightarrow (ly:\text{lab}) \rightarrow \beta\{ly\} \rightarrow \alpha\{JOIN(JOIN(lx,ly), lb)\}$$

We have not explicitly proved a noninterference property for this policy. However, a proof would essentially combine the proof of dependency correctness for the provenance tracking policy and the proof of noninterference for the static information flow policy.

2.3 Composition of Security Policies

All our correctness theorems impose the condition that an application program be "⟦⟧-free". That is, these theorems apply only to situations where a single policy is in effect within a program. However, in practice, multiple policies may be used in conjunction and we would like to reason that interactions between the policies do not result in violations of the intended security properties. To characterize the conditions under which a policy can definitely be composed with another, we define a simple type-based criterion, which when satisfied by two (or more) policies π_P and π_Q, implies that neither policy will interfere

Composes(P,t)	A type t is wrapped within the label namespace P

$$\frac{\cdot \vdash t\{e\} \cong t\{P(\vec{e})\}}{\mathsf{Composes}(P,t\{e\})} \qquad \frac{\mathsf{Composes}(P,t)}{\mathsf{Composes}(P,\forall\alpha{::}\kappa.t)} \qquad \frac{\forall i.\mathsf{Composes}(P,t_i)}{\mathsf{Composes}(P,(x{:}t_1)\to t_2)}$$

Figure 2.11: A type-based composability criterion

with the functioning of the other policy when applied in tandem to the same program.

Figure 2.11 defines a predicate $\mathsf{Composes}(P,t)$, which states that all the labels that appear in the type t of a policy term are enclosed within a top-level constructor P, i.e., the constructor P serves as a *namespace* within which all the labels are enclosed. Intuitively, a policy can be made composable by enclosing all its labels within a unique top-level label constructor that fulfills the role of a namespace. A policy that only manipulates labels and labeled terms that belong to its own namespace can be safely composed with another policy. The main benefit of compositionality is modularity; when multiple composable policies are applied to a program, one can reason about the security of the entire system by considering each policy in isolation. Policy designers that are able to encapsulate their policies within a namespace can package their policies as libraries to be reused along with other policy libraries.

Our notion of composition is a noninterference-like property—a policy is deemed composable if it can be shown not to depend on, or influence the functioning of another policy. The statement of this property appears below.

Theorem (Noninterference for policy composition). *Given*

(A1) $\vec{x}:\vec{t},\vec{y}:\vec{s}\vdash_{app} e : t\{P(\vec{l})\}$, *such that e is* $(\!(\cdot)\!)$*-free*

(A2) $\{e_1,...,e_n\}$ *such that* $\forall i.\cdot \vdash_{pol} e_i : t_i \quad \wedge \quad \mathsf{Composes}(P,t_i)$

(A3) $\{f_1, ..., f_m\} \{g_1, ..., g_m\}$ *such that* $\forall i. \cdot \vdash_{pol} f_i : s_i \quad \wedge \quad \cdot \vdash_{pol} g_i : s_i \quad \wedge \quad \mathsf{Composes}(Q, s_i)$

with $P \neq Q$

(A4)
$$\sigma_f = (\vec{x} \mapsto (\llbracket e_1 \rrbracket, ..., \llbracket e_n \rrbracket), \vec{y} \mapsto (\llbracket f_1 \rrbracket, ..., \llbracket f_m \rrbracket)), \text{ and}$$

$$\sigma_g = (\vec{x} \mapsto (\llbracket e_1 \rrbracket, ..., \llbracket e_n \rrbracket), \vec{y} \mapsto (\llbracket g_1 \rrbracket, ..., \llbracket g_m \rrbracket))$$

(A5) $\sigma_f(e) \overset{c}{\rightsquigarrow}^* v_f \quad \wedge \quad \sigma_g(e) \overset{c}{\rightsquigarrow}^* v_g$

Then, $v_f = v_g$

Assumption (A1) in the statement of the theorem above posits a well-typed application program e that refers to two sets of policy terms $\vec{x}$ and $\vec{y}$. Additionally, (A1) requires e to have a labeled type, where the label $P(\vec{l})$ is drawn from the namespace P. Assumption (A2) posits the existence of well-typed terms $\vec{e}$ that inhabit the types $\vec{t}$ of $\vec{x}$, where each type t_i is drawn from the P-namespace. Similarly, assumption (A3) posits two sets of terms $\vec{f}$ and $\vec{g}$, both of which inhabit the types $\vec{s}$ of $\vec{y}$, where each type s_i is drawn from a different namespace Q. The remaining hypotheses and conclusion of this theorem state that if e is linked with $\vec{e}$ and, in one case, with $\vec{f}$ and in another case with $\vec{g}$, then the values v_f and v_g produced by an evaluation of e in each case are identical. That is, when the type of e indicates that it should produce a value protected by the policy in the P-namespace, then the specific implementation of the policy in the Q-namespace is insignificant. Or, somewhat more intuitively, this theorem states that the choice of the Q-policy cannot influence the behavior of the P-policy.

The proof of this theorem is a corollary of the noninterference result for the static information flow policy using a degenerate lattice where P and Q are incomparable.

This notion of security policy composition generalizes the results of all the security

theorems shown here. The Composes(P, t) predicate provides a recipe by which each of our policy encodings can be adapted so that they compose well with all other policies that have also be so adapted. However, the model of composition proposed here is fairly simple—it essentially allows no interaction between policies. As with noninterference properties in other contexts, this is often too restrictive for many realistic examples in which policies, by design, must interact with each other. We find that policies that do not compose according to this definition perform a kind of declassification (or endorsement) by allowing labeled terms to exit (or unlabeled terms to enter) the policy's namespace. We conjecture that the vast body of research into declassification [112] can be brought to bear here in order to recover a degree of modularity for interacting policies.

Aside from generalization via composition, we could also imagine generalizing our security theorems in more ad hoc ways. For example, one could try to prove that non-observability holds in the presence of multiple user credentials, or with multiple protected objects. In the case of access control, it appears straightforward to prove that such a generalization holds. However, it seems unlikely that such extensions could be proved secure without a policy-specific analysis. For example, in the case of access control with multiple user credentials, one would need to show that a policy implementation does not mistakenly confuse credentials and grant improper access to an unauthorized user. The idea of an enforcement policy in FABLE speaks directly to this concern—it allows all the details of policy enforcement to be defined precisely so that a security analysis can be conducted.

2.4 Concluding Remarks

This chapter has presented FABLE, a core formalism for a programming language in which programmers may specify security policies and reason that these policies are properly enforced. We have shown that FABLE is flexible enough to implement a wide variety of security policies, including access control, provenance, and static information flow, among other policies. We have defined extensional correctness properties for each of our policies and proved that type-correct programs using our policy encodings exhibit these properties. In discussing the structure of the proofs of each of our security theorems, we have argued that FABLE's design greatly simplifies these proofs. In particular, FABLE's metatheory provides a useful repository of lemmas that can be used to discharge many important proof obligations. Finally, we have proposed a method by which composite policies can be applied to a program while still preserving the security properties of each component.

Our focus here has been on enforcing security policies in a purely functional setting. While this has helped keep the presentation simple, in practice, security policies are often stateful and must be applied to programs that may themselves manipulate mutable state. In the following chapters, we show the basic approach of FABLE can be extended to enforce policies that account for state modification.

3. Enforcing Stateful Policies for Functional Programs

Security policies frequently make authorization decisions based on events that may have occurred during a program's execution. For example, various models of stack- and history-based access control have been proposed to modulate the privileges of a piece of code depending on what code has already been executed in a program [51, 1]. The access rights of principals can also change during a program's execution. For instance, with long-running operating systems, network servers, and database systems, new principals may enter the system, while existing principals may leave or change duties. Changes to a principal's privileges may also be transient. In a role-based policy [107], in adherence to the principle of least privilege, users are required to activate a role before requesting access to a resource. Once the access is complete, the user deactivates the role. Such a policy can be implemented in terms of a security automaton [113], where each state records a set of valid facts (e.g, the rights of principals) and security-sensitive events (e.g., role-activations) trigger state transitions.

Even when not concerned with the dynamic changes to access rights, many common policies are naturally phrased in terms of mutable state (in contrast to the purely functional policies of the previous chapter). For example, in an effort to ensure separation of duties, a company's policy may permit a payment to be released only after it has been authorized by two different managers [15]. One could implement this policy using an automaton

which is in the initial state when a payment is requested. Each time an authorization is submitted by a manager, the automaton transitions to a new state. An accepting state is reached when two different authorizations have been received, and only then is the permission to make the payment granted.

Security automata policies have been studied extensively, and are particularly important because they are known to precisely characterize the set of safety properties that can be enforced by an execution monitor. Prior work on enforcing automata-based policies has, for the most part, relied on transforming programs to insert inlined reference monitors [45]. However, this approach has a large trusted computing base in that the compiler that does the transformation has to be trusted to correctly insert code to intercept all security relevant program actions. We would prefer to have a way of verifying that the code produced by one of these transformations correctly implemented the automaton policy—this would remove the complicated compiler from the trusted computing base.

In this context, a type-based approach to verifying the correct enforcement of an automaton-based policy can be particularly useful. Type checking is generally a fairly lightweight syntactic procedure, likely to scale to large programs. In this chapter, we describe an extension to FABLE that, in addition to all the policies explored in Chapter 2, can be used to verify the enforcement of automata-based policies.

3.1 Overview

Our approach has two parts. We begin by introducing a concrete instance of a stateful policy intended to control the terms under which information in the possession of

one principal can be released to another, i.e., an *information release* policy. Our model is based on AIR (Automata for Information Release), a formal language we developed for defining information release policies. AIR's design follows from the observation that information release policies can be naturally expressed as automata. As obligations mentioned in the policy are fulfilled by the program, the state of the automaton advances towards an accepting state—a release is authorized only in the accepting state. AIR policies are able to address a number of concerns, including, to varying degrees, each of the four dimensions of declassification [112]. As such, AIR is of independent interest insofar as, to our knowledge, no other language allows a high-level information release policy (of comparable expressiveness) to be specified separately from the program that is to be secured.

Second, we define λAIR (pronounced "lair"), a language related to FABLE in which type-correct programs can be shown to correctly enforce an AIR policy. Although λAIR extends FABLE with *singleton* and *affine* types [121, 140] and uses a more general language of type constructors, the means by which a policy is enforced in λAIR follows the same pattern as in FABLE. The first step is to protect sensitive data in the program using a security labeling. For example, an object x representing the state of a security automaton is given the affine type ¡*Instance*^{N}, where N is a type-level name unique to x. (Affine types in λAIR are written ¡t, to contrast with the "of course" modality in linear logic, which is typically denoted using "!".) Then, an integer i protected by x would be given type *Protected Int N*, which is analogous to a labeled type in FABLE. When the automaton transitions to a new state y, because x has an affine type, we are able to consume the old state x and ensure that the new state y is used in subsequent authorization decisions.

Operations in the AIR policy that correspond to policy state transitions are represented by privileged policy functions in λAIR. In order to manipulate data with a *Protected* type, a λAIR program is required to call these policy functions—i.e., just as with enforcement policy functions in FABLE. Policy functions in λAIR take arguments that express release obligations. These obligations are given dependent types, where an object having that type serves as a proof that the obligation has been fulfilled. For example, data could be released to a principal *p* only if *p* acts for some principal *q* (where *p* and *q* are program variables that store public keys). A proof of this fact could be represented by an object with type *ActsFor p q*, where *ActsFor* is a programmer-defined dependent type constructor. Generally speaking, proof objects represent certificates which are used to produce a *certified evaluation* of stateful policy logic—every authorization decision is accompanied by a proof that all obligations mandated by the high-level policy have been met.

To focus on the new elements of the type system, our presentation of λAIR takes a more abstract view (relative to FABLE) of the policy functions. Rather than include concrete enforcement policy functions, in λAIR, we allow the policy designer to provide just a type signature for these functions. For example, to interpret access control lists in λAIR, we might include a type signature that gives *access_simple* the type (*acl*:*Lab*) $\rightarrow$ (*Protected Int acl*) $\rightarrow$ *Int*. This type states that *access_simple* is a function that takes a label *acl* as its first argument; an integer protected by this label as the second argument; and returns an unlabeled integer. The runtime behavior of *access_simple*, in FABLE is implemented in the language itself (in bracketed code) to include specific membership test and an unlabeling operation. Here, the semantics of this function is specified outside of λAIR, using an abstract model. Additionally, λAIR's more general

language of type constructors allows us to give more convenient types to capabilities and certificates. For example, instead of representing a flow capability using a value of type unit{*FLOW*(*src*, *dst*)}, as we did with FABLE in Section 2.2.4, we can simply define a dependent type constructor *Flow*, and give the capability a type *Flow src dst*. In Section 3.6 we show how the FABLE type system can be embedded in λAIR.

3.2 AIR: Automata for Information Release

Many organizations, including financial institutions, health-care providers, the military, and even the organizers of academic conferences, wish to specify the terms under which sensitive information in their possession can be released to their partners, clients, or the public. Such a specification constitutes an *information release policy*. These policies are often quite complex. For example, consider the policy that regulates the disclosure of military information to foreign governments as defined by the United States Department of Defense [128]. This policy includes the following provisions: a release must be authorized by an official with disclosure authority who represents the "DoD Component that originated the information"; the system must "edit or rewrite data packages to exclude information that is beyond that which has been authorized for disclosure"; a disclosure shall not occur until the foreign government has submitted "a security assurance [. . .] on the individuals who are to receive the information"; and, that the release must take place in the Foreign Disclosure and Technical Information System in which both approvals and denials of a release request must be logged.

We would like to ensure that software systems that handle sensitive data—including

military systems, but also programs like medical-record databases, online auction software, and network appliances—correctly enforce such a high-level policy. As a concrete example, consider a specific kind of application called a *cross-domain guard*. These are programs, like network firewalls, that mediate the transfer of information between organizations at different trust levels. Commercial guards, e.g., the Data Sync guard produced by BAE [48], do not enforce high-level policies but rather implement low-level "dirty keyword" filters.

The research community has only recently begun to consider the verified enforcement of release policies. For instance, FlowWall [62] is arguably the research counterpart of a system like DataSync guard. By virtue of its being built with the Jif programming language [31], FlowWall is sure to enforce a low-level filtering policy, but it does not appeal to high-level information release criteria. Augmenting information flow policies with high-level conditions that control information release has been proposed by Chong and Myers [30] and, more recently, by Banerjee and Naumann [9]. However, in both these cases, reasoning separately about high-level release decisions is difficult since the release policy is embedded within the program.

To fill this gap, we define AIR, a formal language for defining information release policies separately from the program that is to be secured. AIR's design follows from the observation that an information release policy is a kind of stateful authorization policy naturally expressed as an automaton. Satisfaction of a release obligation advances the state of the automation, and once all obligations have been fulfilled, the automaton reaches the accepting state and the protected information can be released. AIR allows one to express such automata in a natural way.

In subsequent sections, we show how an AIR policy can be compiled to an API in λAIR, where each API function corresponds to an automaton transition such that the type of that function precisely expresses the evidence necessary for a transition to succeed—these API functions represent the enforcement policy that ties an AIR policy to a λAIR program. The type system of λAIR ensures that programs use the compiled AIR API correctly and, as a consequence, meet the specifications of the high-level policy. More precisely, we prove that the sequence of events produced by a program's execution is a word in the language accepted by the AIR automaton.

Using our techniques, one could build a cross-domain guard that adheres to high-level policy prescriptions; e.g., it would release information only after confirming that appropriate security assurances have been received, that to-be-released data packages have been rewritten appropriately, and that audit logs have been updated.

Our use of AIR policies for information release departs from prior work on declassification policies in that we do not focus on establishing a noninterference-like property for programs. However, our work complements noninterference-oriented interpretations of information release. In particular, by showing how to embed FABLE in λAIR(Section 3.6), we argue that high-level AIR policies can be enforced in conjunction with information flow in λAIR. For example, we could ensure that an adversary can never influence a program to cause information to be released, and furthermore, when it is released, it always follows the prescription of the high-level AIR policy.

3.2.1 Syntax of AIR, by Example

An AIR policy consists of one or more *class declarations*. A program will contain *instances* of a class, where each instance protects some sensitive data via a labeling. Protected data can be accessed in two ways. First, each class C has an *owning principal P* such that *P* and all who *act for P* may access data protected by an instance of C. Second, each class defines a *release policy* by which its protected data can be released to an instance of a different class.

The release policy is expressed using rules that define a security automaton, which is a potentially infinite state machine in which states represent security-relevant configurations. In the case of AIR, the security automaton defines conditions that must hold before data can be released. Each class instance consists of its current state, and each condition that is satisfied transitions the automaton to the next state. These transitions ultimately end in a release rule that allows data to be released to a different class instance, potentially in a modified form. Because sensitive data is associated with instances rather than classes, multiple resources may be governed by the same policy template (i.e., the automaton defined by the class) but release decisions are made independently for each resource. Dually, related resources can be protected by the same instance, thereby allowing release decisions made with respect to one resource to affect the others.

The formal syntax of AIR policies is presented in Figure 3.1. We explain the syntax of AIR while stepping through a running example, shown in Figure 3.2. A class declaration consists of a class identifier, an identifier for the owning principal, a list of automaton states, and a sequence of rules that define the automaton transitions. Our example de-

Metavariables

id	class and rule ids	*P*	principals
$\mathscr{C}$	state constructors	n, i, j	integers
x, y, z	variables		

Core language

Declarations	D	::=	class id = (principal:*P*; states: $\overrightarrow{S}$; $\overrightarrow{R}$)
States	S	::=	$\mathscr{C}$ | $\mathscr{C}$ of $\overrightarrow{t}$
Rules	R	::=	id : R | id : T
Release	R	::=	When *G* release *e* with next state *A*
Transition	T	::=	When *G* do *e* with next state *A*
Guards	G	::=	x requested for use at y and $\overrightarrow{\exists x{:}t.C}$
Conditions	C	::=	A_1 *IsClass* A_2 | A_1 *InState* A_2
			| A_1 *ActsFor* A_2 | $A_1 \leq A_2$
Atoms	A	::=	n | x | id | P | $\mathscr{C}(\overrightarrow{A})$ | $A_1 + A_2$
			| Self | Class(A) | Principal(A)

e is an expression and *t* is a type in λAIR. (cf. Figure 3.4)

Figure 3.1: Syntax of AIR

clares a single class US_Army_Confidential, owned by the principal *US_Army*, that defines

the policy for confidential data owned by the U.S. Army. For simplicity, our examples

use a flat namespace for class identifiers, and abstract names for principals.

Automaton states are represented by terms constructed from an algebraic datatype.

The example has two kinds of states. The nullary constructor Init represents the initial

state of the automaton; all classes must have this state. The other kind of state is an

application of the unary constructor Debt to an argument of type *Int*. Constructors of the

form $\mathscr{C}$ of $\overrightarrow{t}$ may carry data as indicated by the types $\overrightarrow{t}$. Types *t* (such as *Int*) are drawn

from the programming language λAIR in which programs using AIR policies are written;

λAIR is discussed in the next section.

Each rule in an AIR class is given a name, and is either a *release rule* or a *transition*

rule. Each rule begins with a clause "When *x* requested for use at *d*," which serves to bind

variables *x* and *d* in the remainder of the rule. Here, *x* names the information protected by an instance of this class, requested for release to some other instance *d* (usually of another class). This clause is followed by a conjunction of conditions that restrict the applicability of a rule; we discuss these in more detail below. Following these conditions, the rule specifies a λAIR expression *e* that can either release information (perhaps after downgrading it by filtering or encryption) or do some other action (like logging), depending on whether the rule is a release rule or a transition rule. A rule concludes with the next state of the automaton.

The first rule in the US_Army_Confidential class is a release rule called Conf_secret. This rule is qualified by a condition expression Class(*d*) *IsClass* US_Army_Secret stating that the rule applies when releasing *x* to an instance *d* of a class named US_Army_Secret. If applicable, this rule allows *x* to be released without modification—the release expression is simply *x*, and not, some function that downgrades *x*. After the release, the automaton remains in its current state; i.e. the state Self.

We use a small ontology for conditions based on integers, principals, classes and their instances—*IsClass* mentioned above, is one such condition. We expect this ontology to be extended, as needed. Generally speaking, condition expressions *C* are typed binary predicates over atoms *A*. For example, A_1 *ActsFor* A_2 is defined for *Principal*-typed atoms A_1 and A_2, and asserts that A_1 acts for A_2 according to some acts-for hierarchy among principals (not explicitly modeled here). Atoms include integers *n*, variables *x*, identifiers *id*, principal constants *P*, state literals constructed from an application of a state constructor $\mathscr{C}$ to a list of atoms, addition of integers and the implicit variable Self. We also include two operators: Class(*z*) is the class of the argument *z*, a class instance; and, Principal(*z*),

```
class US_Army_Confidential =
    principal : US_Army;    states : Init, Debt of Int;

  Conf_secret :
      When x requested for use at d and
          Class(d) IsClass US_Army_Secret
      release x with next state Self

  Conf_init :
      When x requested for use at d and
          Self InState Init
      do _ with next state Debt(0)

  Conf_coalition :
      When x requested for use at d and
          Principal(Class(d)) ActsFor Coalition,
          ∃count:Int.Self InState Debt(count),
          count ≤ 10
      release
          (log(...x...d); encrypt (pubkey (principal (class d))) x)
      with next state Debt(count + 1)
```

Figure 3.2: A stateful information release policy in AIR

which is the principal that owns the class z. Finally, we permit a condition C to be prefixed

by one or more existentially quantified variables—i.e., in $\exists x_1{:}t_1.C_1, \ldots, \exists x_n{:}t_n.C_n$, each x_i

is a variable of type t_i and is in scope as far to the right as possible, until the end of the

rule. We omit the quantifier prefix when no such variables exist.

3.2.2   A Simple Stateful Policy in AIR

Taken as a whole, the class US_Army_Confidential can be thought of as implement-

ing a simple kind of *risk-adaptive* access control [27], in which information is released

according to a *risk budget*, with the intention of quantifying the risks vs. the benefits of

releasing sensitive information. This class maintains a current risk debt, as reflected in

the state Debt of *Int*. Each time the class authorizes an information release we add an

estimate of the risk associated with that release to the debt. When the accumulated risk debt exceeds a threshold then releases outside the U.S. Army are no longer permitted. The other two rules in the policy, Conf_init and Conf_coalition, implement this behavior.

The Conf_init transition rule applies when processing a release to an instance d and when the automaton is in the Init state. The "do" expression initializes the risk debt to 0 by transitioning the automaton to the Debt(0) state. The Conf_coalition rule allows information to be released to a coalition partner. In particular, if the release target class is owned by a principal that acts for the *Coalition* (expressed by Principal(Class(d)) *ActsFor Coalition*), then information can be released only if the current risk debt has not exceeded the budget, as expressed in the latter two conditions. The first of these requires the current state of the automaton to be Debt(*count*), where *count* is variable with type *Int* which holds the current risk debt. The last condition requires that *count* is not above the preallocated risk budget of 10. With these conditions satisfied, Conf_coalition logs the fact that a release has been authorized and permits release of the data after it has been downgraded using an encryption function. In this case, the downgrading expression encrypts x with the public key of the principal that owns the class of the instance d. Unlike releases to US_Army_Secret which do not alter the risk debt, Conf_coalition increments the risk debt by transitioning to the Debt(*count* + 1) state, indicating that releases to the *Coalition* are more risky than upgrading to a higher classification level of the same organization (via rule Conf_secret).

AIR as presented here is particularly simple. We anticipate extending AIR with support for more expressive condition ontologies and release rules. For instance, instead of a fixed set of ontologies, we could embed a stateful authorization logic (say, in the style of SMP [15]) to allow custom ontologies and release rules to be programmed within

an AIR class. We could also introduce a set of downgrading and logging primitives to completely separate AIR from λAIR. Additionally, AIR's object-oriented design is intended to support extensions like inheritance and overloading that are likely to help with the modular construction and management of large policies.

3.3 A Programming Model for AIR

Given a particular AIR policy, we would like to do two things. First, we must have a way of reflecting an AIR policy in a program by protecting sensitive resources with instances of an AIR class. Second, we must ensure that all uses of protected data adhere to the prescriptions of the AIR policy. Taken together, we can then claim that an AIR policy is correctly enforced by a program. To achieve these goals, we have defined a formal model for a language called λAIR in which one writes programs that use AIR policies. λAIR's type system ensures that these policies are used correctly. The rest of this section defines the programming model for this language and the next two sections flesh out its syntax and semantics. Section 3.5.4 proves that type-correct programs act only in accordance with their AIR policies.

The programming model for using AIR policies has two elements. First, programmers tie an AIR policy to data in the program by constructing instances of AIR classes and labeling one or more pieces of data with these instances. This association defines (1) the set of principals that may view the data (in particular, the principal P that owns the class, and any principals that may act for P), and (2) the rules that allow the data to be released. As in other security-typed languages, the labeling specification (expressed

using type annotations) is part of the trusted computing base.

Second, programmers manipulate data protected by an AIR class instance through a class-specific API that is generated by compiling each AIR class definition to a series of program-level definitions. For example, each AIR class's release and transition rules are compiled to functions that can be used to release protected data. The types given to these functions ensure that a caller of the function must always provide evidence that the necessary conditions to release protected data have been met.

Figure 3.3 illustrates a program using the AIR policy of Figure 3.2, written using a ML-like notation. (Significantly, our examples omit type annotations where they do not help clarify the exposition. λAIR does not support type inference at all.) At a high level, this program processes requests to release information from a secret file. The files are stored on the file system together with a policy label that represents a particular AIR class instance. Before disclosing the information, the program must make sure that the automaton that protects the data is in a state that permits the release. The first two lines set up the scenario. At line 1, we read the contents of a secret file into the variable *x_a1* and the automaton that protects this file into the variable *a1*. Initially, only the principals that act for the owner of the class of *a1* can view these secrets. At line 2, the program blocks until a request is received. The request consists of an output *channel* and another automaton instance *a2* that represents the policy under which the requested information will be protected after the release. In effect, the information, once released, will be under the protection of the principal that owns the class of *a2*.

Prior to responding to the request, on lines 4-7 we must establish that a1 is in a state that permits the release. At line 4, we extract the class of the instance a2. At line

```
1    let x_a1, a1 = get_secret_file_and_policy () in
2    let a2, channel = get_request () in
3     (∗ generating evidence of policy compliance ∗)
4    let a2, a2_class = get_class a2 in
5    let ev1 = acts_for (principal a2_class) Coalition in
6    let a1, Debt(debt), ev2 = get_current_state a1 in
7    let ev3 = leq debt 10 in
8     (∗ supplying evidence to policy API and releasing data ∗)
9    let a1', a2, x_a2 = Conf_coalition a1 x_a1 a2 ev1 debt ev2 ev3 in
10      send channel x_a2
```

Figure 3.3: Programming with an AIR policy

5, we check that the owner of a2's class acts for the Coalition principal and, if this check

succeeds, we obtain a certificate ev1 as evidence of this fact. At line 6, we extract the

current state of the automaton a1, use pattern matching to check that it is of the form

Debt(debt) (for some value of debt) and receive an evidence object ev2 that attests to the

fact that a1 is currently in this state. At line 7, we check that the total debt associated with

the current state of the automaton is not greater than 10 and obtain ev3 as evidence if the

check succeeds.

At line 9 we call Conf_coalition, a function produced by compiling the AIR policy.

We pass in the automaton a1 and the secret data x_a1; the automaton a2 to which x_a1

is to be released; and the certificates that serve as evidence for the release conditions.

Conf_coalition returns a1' which represents the next state of the automaton (presumably in

the Debt(debt+1) state); a2 the unchanged destination automaton; and finally, x_a2, which

contains the suitably downgraded secret value. On the last line, we send the released

information on the channel received with the request.

For programs like our example, we would like to verify that all releases of infor-

mation are mediated by calls to the appropriate transition and release rules as defined

by the AIR policy (functions like Conf_coalition). Additionally, we would like to verify that a program satisfies the mandates of an AIR policy rule by presenting evidence that justifies the appropriate release conditions. This evidence-passing style supports our goal of certifying the evaluation of all authorization decisions, while being flexible about the mechanism by which an obligation is fulfilled. To return to the DoD example from the introduction, this design gives us the flexibility to allow release authorizations to be obtained in one part of the system and security assurances from the recipient to be handled in another; the cross-domain guard must simply collect evidence from the other components rather than performing these operations itself. λAIR's type system is designed so that type correctness ensures these goals are satisfied, i.e., a type-correct program uses its AIR policy correctly. The type system has three key elements:

Singleton types. First, in order to ensure complete mediation, we must be able to correctly associate data with the class instance that protects it. For example, Conf_coalition expects its first argument to be an automaton and its second argument to be data protected by that automaton. In an ML-like type system, this function's type might have the form $\forall \alpha.Instance \rightarrow \alpha \rightarrow t$ But such a type is not sufficiently precise since it does not prescribe any relationship between the first and second argument, e.g., allowing the programmer to erroneously pass in a2 as the first argument, rather than a1. To remedy this problem, we can give Conf_coalition a type like the following (as a first approximation):

$$\forall N, \alpha.Instance^N \rightarrow Protected \ \alpha \ N \rightarrow \ldots$$

Here, N is a unique type-level name for the class instance provided in the first argument.

The second argument's type *Protected α N* indicates it is an α value protected by the instance *N*, making clear the association between policy and data. We can ensure that values of type *Protected α N* may only be accessed by principals *P* that act for the owner of the class instantiated by the instance named *N*. This approach is more flexible than implicitly pairing each protected object with its own (hidden) automaton. For example, with our approach one can encode policies like secret sharing, in which a set of related documents are all protected by the same automaton instance. Each document's type would refer to the same automaton, e.g., *Protected Doc N*. Information released about one document updates the state of the automaton named *N* and can limit releases of the other documents.

Dependent types. Arguments 4-7 of Conf_coalition represent evidence (proof certificates) that the owner of class instance a2 acts for Coalition, and that a1 is in a state authorized to release the given data. The types we give to these arguments reflect the propositions that the arguments are supposed to witness. For example, we give the seventh argument (ev3) to Conf_coalition the type *LEQ debt* 10 where *LEQ* is a dependent type constructor applied to two *expressions*, debt and 10, which themselves have type *Int*. Data with type *LEQ n m* represents a certificate that proves $n \leq m$. If we allow such certificate values to only be constructed by trusted functions that are known to correctly implement the semantics of integer inequality, then we can be sure that functions like Conf_coalition are only called with valid certificates, i.e., type correctness guarantees that all certificates are valid proofs of the propositions represented by their types, and there is no need to inspect these certificates at run time. If we interface with other programs, we can check the

validity of proof certificates at run time before allowing a call to proceed. Either way, the type system supports an architecture that enables certified evaluation of an AIR policy.

Affine types. The final piece of our type system is designed to cope with the stateful nature of an AIR policy. The main problem caused by a state change is illustrated by the value returned by the Conf_coalition function. In our example, a1' represents the state of the policy automaton that protects x_a1 after a release has been authorized. Thus, we need a way to break the association between x_a1 and the old, stale automaton state a1. We achieve this in two steps. First, even though our type system supports dependent types, as shown earlier, we use singleton types to give x_a1 the type *Protected α N*, where *N* is a unique type name for a1 (rather than giving x_a1 a more-direct dependent type of the form *Protected α a1*). The second step is to use *affine types* (values with an affine type can never be used more than once) to consume stale automaton values, so that at any program point, there is only one usable automaton value that has the type-name *N*. Thus, we give both a1 and a1' the type ¡*Instance^N*, where ¡*t* denotes an affinely qualified type *t*. Once a1 is passed as an argument to Conf_coalition (constituting a use) it can no longer be used in the rest of the program; a1' is the only automaton that can be used in subsequent authorization checks for x_a1. Thus, a combination of singleton and affine types transparently takes care of relabeling data with new automaton instances. (One might also wonder how we deal with proof certificates that can become stale because of the changing automaton state; we discuss this issue in detail in Section 3.5.1.)

To illustrate how singleton, dependent, and affine types interact, we show the (slightly simplified) type of Conf_coalition below. The full type is discussed in Section 3.5.2.

70

$$\forall N, M, \alpha. \; ¡Instance^N \rightarrow Protected \; \alpha \; N \rightarrow ¡Instance^M \rightarrow$$
$$\ldots \rightarrow (\mathsf{debt} : Int) \rightarrow \ldots \rightarrow (LEQ \; \mathsf{debt} \; 10) \rightarrow$$
$$(¡Instance^N \times ¡Instance^M \times Protected \; \alpha \; M)$$

The first three arguments are the *affine* source automaton (a1), the data it protects (x_a1), and the *affine* destination automaton (a2). On the next line, we show the dependent type given to the evidence that the current debt of the automaton is not greater than 10. Finally, consider the return type of Conf_coalition. The first component of this three-tuple is a class instance with the same name N as the first argument. This returned value is the new state of the automaton named N—it protects all existing data of type *Protected* α N (such as x_a1). The second component of the three-tuple is the unchanged target automaton. The third component contains the data ready to be released—its type, *Protected* α M, indicates that it is now protected by the target automaton instance M. In effect, λAIR models state modifications by requiring automata states to be manipulated in a store-passing style, reminiscent of a monadic treatment of side effects in a purely functional language [69]. However, by imposing the additional discipline of affine types, we are able to ensure that the program always has a consistent view of an automaton's state, while still retaining the benefits of a well-understood and relatively simple functional semantics.

The reader may be concerned about the difficulty of programming with affine types in λAIR. We put forth an argument in two parts in order to quell this concern. First, when enforcing purely functional FABLE-style policies in λAIR, affine types need not be used at all. More subtly, even when affine types are used to enforce stateful policies, we conjecture that λAIR's type system may actually simplify the programming task rather

than complicate it. Admittedly, prior work on adding affine types to a programming language flies in the face of this conjecture. For example, affine types have been used in Cyclone to prevent the creation of pointer aliases [124]. When the referent of an alias-free pointer is deallocated, it is easy to show that no dangling pointers remain. In our experience, using affine types to control pointer aliasing makes programming in Cyclone considerably harder. A main difficulty is that non-affine aliasing is not always a symptom of a programming error, e.g., pointers may be freely aliased so long as no pointer in an alias set is dereferenced after the referent is deallocated. In contrast, affine types in λAIR are used to restrict the use of stale policy states rather than to control pointer aliasing. We are optimistic about the usability of affine types in λAIR because these types appear to very naturally capture the only correct usage mode of a stateful policy—any use of a stale policy state in an authorization decision violates the consistency of the policy. Thus, we conjecture that any correct implementation of a stateful policy must adhere to an affine discipline on policy states. λAIR's type system may actually simplify this task, since it can detect common programming errors that cause the required affine discipline to be violated.

Nevertheless, we acknowledge that adhering to the constraints of λAIR's type system is surely more burdensome than when using a more traditional programming language. Thus λAIR may be most appropriate for the security-critical kernel of an application, or even as the (certifiable) target language of a program transformation for inline reference monitoring. We leave to future work support for improving λAIR's usability, such as type inference.

Metavariables

B	Base terms functions	*T*	Type constructors
D	Data constructors	α, β, γ	Type variables

Core language

Terms	e	$::=\;\; x \mid \lambda x{:}t.e \mid e\,e \mid \Lambda\alpha{::}k.e \mid e\,[t] \mid B \mid D$
		$\mid\;\;$ case e of $\overrightarrow{x{:}t.e} : e$ else $e \mid \bot \mid$ new e
Types	t	$::=\;\; (x{:}t) \to t \mid \alpha \mid \forall\alpha{::}k \xrightarrow{\varepsilon} t \mid T$
		$\mid\;\; t \Rightarrow t \mid q\,t \mid t\,t \mid t\,e \mid t^{\eta}$
Type names	η	$::=\;\; \alpha \mid \circ$
Affinity	q	$::=\;\; ¡ \mid \cdot$
Simple kinds	k	$::=\;\; \mathtt{U} \mid \mathtt{A} \mid \mathtt{N}$
Kinds	K	$::=\;\; k \mid k \to K \mid t \to K$
Name constraints	ε	$::=\;\; \cdot \mid \alpha \mid \varepsilon \uplus \varepsilon \mid \varepsilon \cup \varepsilon$

Signatures and typing environments

Phase index	φ	$::=\;\;$ term $\mid$ type
Signatures	S	$::=\;\; (B{:}t) \mid (D{:}t) \mid (T{::}K) \mid S,S$
Type env.	Γ	$::=\;\; \Gamma, x{:}t \mid \Gamma, \alpha{::}k \mid S$
Affine env.	A	$::=\;\; x \mid A,A$

Figure 3.4: Syntax of λAIR

3.4 Syntax and Semantics of λAIR

λAIR extends a core System F^{ω} [85] with support for singleton, dependent, and affine types. λAIR is parameterized by a *signature S* that defines base term functions *B*, data constructors *D*, and type constructors *T*—each AIR class declaration D is compiled to a signature S_D that acts as the API for programs that use D. All AIR classes share some elements in common, like integers, which appear in a *prelude* signature S_0. We explain the core of λAIR using examples from the prelude. The next section describes the remainder of the prelude and shows how our example AIR policy is compiled.

3.4.1 Syntax

Figure 3.4 shows the syntax of λAIR. The core language expressions e are mostly standard, including variables x, lambda abstractions $\lambda x{:}t.e$, application $e\ e'$, type abstraction $\Lambda\alpha{::}k.e$, and type application $e\ [t]$. Functions have dependent type $(x{:}t) \rightarrow t'$ where x names the argument and may be bound in t'. Type variables are α. A type t universally quantified over all types α of kind k is denoted $\forall\alpha{::}k \xrightarrow{\varepsilon} t$. Here, ε is a name constraint that records the type names α given to automaton instances in the body of the abstraction; we discuss these in detail later. When the constraint is empty we write a universally quantified type as $\forall\alpha{::}k.t$. The signature S defines the legal base terms, B and D, and type constructors T, mapping them to their types t and kinds K, respectively. (We distinguish between base terms functions and data constructors syntactically since, as illustrated in Section 3.4.3, they have different operational semantics.) The prelude S_0 defines several standard terms and types which we use to illustrate some of λAIR's main features.

The type constructor *Int* represents the type of integers, and is given U kind in the prelude (written *Int*::U). Kind U is one of three simple kinds k. A type t with simple kind A is affine in that the typing rules permit terms of type t to be used at most once. ¡t is an instance of the form $q\ t$ where $q = $ ¡. Terms whose types have kind U are unrestricted in their use (explaining the choice of U as the name of this kind). We explain kind N, the kind of type names, shortly.

The prelude also defines two base *data constructors* for constructing integers: Zero : *Int* represents the integer 0, while Succ : *Int* $\Rightarrow$ *Int* is a unary data constructor that produces an *Int* given an *Int*. Data constructor application is written $e\ (e)$; thus the integer 1 is

represented Succ (Zero) (but we write $0, 1, 2$ etc. for brevity). Programs can pattern match data constructors applications using the expression form case e of $\overrightarrow{x{:}t}.e : e$ else e. This is mostly standard; details are in Appendix B.

In addition to simple kinds k, kinds K more generally can classify functional type constructors, using the forms $k \to K$ and $t \to K$. A type constructor t_1 having the first form can be applied to another type (as $t_1 \ t_2$) to produce a (standard) type, while one of the second form can be applied to a term (as $t \ e$) to produce a dependent type. As an example of the first case, the prelude defines a type constructor $\times{::}U \to U \to U$ to model pairs; $\times \textit{Int Int}$ is the type of a pair of integers (for clarity, from here on we will use infix notation and write a pair type as $t \times t'$). The prelude also defines a base-term constructor Pair which has a polymorphic type $\forall \alpha, \beta {::} U. \alpha \Rightarrow \beta \Rightarrow \alpha \times \beta$ for constructing pair values.

Evidence for condition expressions in an AIR policy are given dependent types. For example, the prelude provides means to test inequalities $A_1 \leq A_2$ that appear in a policy and generate certificates that witness an inequality:

$$(\textit{LEQ}{::}\textit{Int} \to \textit{Int} \to U),$$

$$(\textit{leq}{:}(x{:}\textit{Int}) \to (y{:}\textit{Int}) \to \textit{LEQ } x \ y)$$

$\textit{LEQ}$ is a dependent-type constructor that takes two expressions of type $\textit{Int}$ as arguments and produces a type having kind U. This type is used to classify certificates that witness the inequality between the term arguments. These certificates are generated by $\textit{leq}$ which is a *base term function* with a dependent type: the labels x and y on the first two arguments appear in the returned type. Thus the call $\textit{leq } 3 \ 4$ would return a certificate of type $\textit{LEQ } 3 \ 4$

because 3 is indeed less than 4. An attempt to construct a certificate *LEQ* 4 3 by calling

leq 4 3 would fail at run time, returning $\bot$ (an unrecoverable failure) in our semantics—

we could use option types to handle failures more gracefully. The signature does not

include a data constructor for the *LEQ* type, so its values cannot be constructed directly

by programs—the only way is by calling the *leq* function.

We discuss the remaining constructs—including name constraints ε, named types

t^η, and the new *e* construct—in conjunction with the type rules next.

3.4.2   Static Semantics

Figure 3.5 shows the main rules from the static semantics of λAIR, which consists

of two judgments. The full semantics can be found in Appendix B. The typing judgment

is parameterized by a *phase index* φ, which indicates whether the judgment applies to a

term- or type-level expression. (Note that, though seemingly related, the phase index is

not to be confused with the color index in FABLE. Colors distinguish application from

enforcement policy code. The phase index distinguishes expressions that appear at the

type-level from those that appear within term.) The judgment giving an expression *e* a

type *t* is written $\Gamma; A \vdash_\varphi e : t; \varepsilon$ where Γ is the standard typing environment augmented

with the signature *S* (used to type base terms and type constructors), *A* is a list of affine

assumptions, and ε is a name constraint that records the set of fresh type names assigned

to automata instances in *e*. The second judgment, $\Gamma \vdash t :: K$ states that a type *t* has kind *K*

in the environment Γ.

Recall that the type system must address three main concerns. First, we must cor-

$$\Gamma; A \vdash_\varphi e : t; \varepsilon \qquad\qquad\qquad A\ \varphi\text{-level expression } e \text{ has type } t \text{ and uses names } \varepsilon$$

$$\frac{\Gamma \vdash \Gamma(x) :: U}{\Gamma; \cdot \vdash_\varphi x : \Gamma(x); \cdot} \text{ (T-X)} \qquad \frac{}{\Gamma; x \vdash_\varphi x : \Gamma(x); \cdot} \text{ (T-XA)} \qquad \frac{}{\Gamma; \cdot \vdash_{\text{type}} x : \Gamma(x); \cdot} \text{ (T-X-type)}$$

$$\frac{\Gamma; A \vdash_{\text{type}} e : t; \varepsilon_1 \uplus \varepsilon}{\Gamma; A \vdash_{\text{type}} e : t; \varepsilon} \text{ (T-NC-type)} \qquad \frac{\Gamma; A \vdash_\varphi e : t; \varepsilon \quad \varepsilon' \subseteq \text{dom}(\Gamma)}{\Gamma; A, A' \vdash_\varphi e : t; \varepsilon \uplus \varepsilon'} \text{ (T-WKN)}$$

$$\frac{\Gamma; A \vdash_\varphi e : t; \varepsilon \quad \Gamma \vdash t :: U \quad \Gamma(\alpha) = \mathtt{N}}{\Gamma; A \vdash_\varphi \text{new } e : \mathbf{i} t^\alpha; \alpha \uplus \varepsilon} \text{ (T-NEW)} \qquad \frac{\Gamma; A \vdash_\varphi e : t^\alpha; \varepsilon}{\Gamma; A \vdash_\varphi e : t^\circ; \varepsilon} \text{ (T-DROP)}$$

$$\frac{\Gamma, \alpha :: k; A \vdash_\varphi e : t; \varepsilon \uplus \varepsilon' \quad \alpha \notin \varepsilon \quad \varepsilon' \in \{\cdot, \alpha\} \quad q = p(A, \varepsilon)}{\Gamma; A \vdash_\varphi \Lambda \alpha :: k.e : q(\forall \alpha :: k \xrightarrow{\varepsilon'} t); \varepsilon} \text{ (T-TAB)}$$

$$\frac{\Gamma \vdash t_x :: k \quad q = p(A, \varepsilon) \quad \Gamma, x : t_x; A, a(x, k) \vdash_\varphi e : t_e; \varepsilon}{\Gamma; A \vdash_\varphi \lambda x {:} t_x.e : q((x {:} t_x) \to t_e); \varepsilon} \text{ (T-ABS)}$$

$$\frac{\Gamma; A \vdash_\varphi e : q(\forall \alpha :: k \xrightarrow{\varepsilon'} t'); \varepsilon \quad \Gamma \vdash t :: k}{\Gamma; A \vdash_\varphi e\,[t] : [\alpha \mapsto t]t'; \varepsilon \uplus ([\alpha \mapsto t]\varepsilon')} \text{ (T-TAP)}$$

$$\frac{\Gamma; A \vdash_\varphi e : q((x {:} t') \to t); \varepsilon_1 \quad \Gamma; A' \vdash_\varphi e' : t'; \varepsilon_2}{\Gamma; A, A' \vdash_\varphi e\ e' : [x \mapsto e']t; \varepsilon_1 \uplus \varepsilon_2} \text{ (T-APP)}$$

$$\begin{aligned} \textit{where} \quad a(x, \mathtt{A}) &= x & a(x, \mathtt{U}) &= \cdot \\ p(A, \varepsilon) &= \mathbf{i} & p(\cdot, \cdot) &= \cdot \end{aligned}$$

$$\Gamma \vdash t :: K \qquad\qquad\qquad A \text{ type } t \text{ has kind } K \text{ in environment } \Gamma$$

$$\frac{\Gamma(\alpha) = k}{\Gamma \vdash \alpha :: k} \text{ (K-A)} \qquad \frac{\Gamma \vdash t :: \mathtt{A} \quad \Gamma(\eta) = \mathtt{N} \vee \eta = \circ}{\Gamma \vdash t^\eta :: \mathtt{A}} \text{ (K-N)} \qquad \frac{\Gamma \vdash t :: \mathtt{U}}{\Gamma \vdash \mathbf{i} t :: \mathtt{A}} \text{ (K-AFN)}$$

$$\frac{\Gamma \vdash t :: k \quad \Gamma, x : t \vdash t' :: k'}{\Gamma \vdash (x {:} t) \to t' :: \mathtt{U}} \text{ (K-FUN)} \qquad \frac{\Gamma \vdash t :: t' \to K \quad \Gamma; \cdot \vdash_{S,\text{type}} e : t'; \cdot}{\Gamma \vdash t\ e :: K} \text{ (K-DEP)}$$

$$\frac{\Gamma' = \Gamma, \alpha :: k \quad \Gamma' \vdash t :: k \quad \alpha' \in \varepsilon \Rightarrow \Gamma'(\alpha') = \mathtt{N}}{\Gamma \vdash \forall \alpha :: k \xrightarrow{\varepsilon} t :: \mathtt{U}} \text{ (K-UNIV)}$$

Figure 3.5: Static semantics of λAIR (Selected rules)

rectly assign unique type names to automata instances and then associate these names with protected data. Next, for certified evaluation, we must be able to accurately type evidence using dependent types. Finally, to cope with automaton state changes, we must (via affine types) prevent stale automaton instances from being reused. We consider each of these aspects of the system in turn, first in the typing judgment and then in the kinding judgment.

Assigning unique names to automata. We construct new automata using new e. (T-NEW) assigns the name α to the type in the conclusion, ensuring (via $\alpha \uplus \varepsilon$) that α is distinct from all other names ε that have been assigned to other automata. We require α to be in the initial environment Γ, or to be introduced into the context by a type abstraction. Recall from Section 3.3 that protected values will refer to this name α in their types (e.g., *Protected Int* α). The resulting type $\mathfrak{i}t^{\alpha}$ is also affinely qualified; we discuss this shortly.

(T-DROP) allows the unique name associated with a type to be replaced with the distinguished constant name $\circ$. This is sound because although the name α of a type $\mathfrak{i}t^{\alpha}$ can be hidden, α cannot be reused as the type-level name of any other automaton (i.e., ε is unaffected). This form of subtyping is convenient for giving types to proof objects that witness properties of the state of an automaton, while keeping our language of kinds for type constructors relatively simple. Section 3.5.1 illustrates an example use of (T-DROP).

(T-TAB) is used to check type abstractions. The first premise checks the body of the abstraction e in a context that includes the abstracted type variable α. Since we treat type names and types uniformly, functions polymorphic in a type name can be written by quantifying over $\alpha::\mathbb{N}$—the interesting elements of this rule have to do with managing

these names. If the body of the abstraction e constructs a new automaton assigned the name α in (T-NEW), then α will be recorded in $\varepsilon \uplus \varepsilon'$, the name constraints of e. In this case $\varepsilon' = \alpha$ and ε contains all the other names used in the typing derivation of e; otherwise ε' is empty. In the conclusion, we decorate the universally quantified type with ε' to signify that the abstracted name α is used in e. Type abstractions are destructed according to (T-TAP). In the premises we require the kind of the argument to match the kind of the formal type parameter. In the conclusion, we must instantiate all the abstracted names ε' used in the body e' and ensure that these are disjoint from all other names ε used in the body.

Two additional points are worth noting. First, universally quantified types can be decorated with arbitrary name constraints ε (rather than just singleton names α). We expect this to be useful when enforcing composite policies. The name instantiation constraint ε can ensure that a function always constructs automata that belong to a specific set of classes in a large policy. Second, we could support recursion by following an approach taken by Pratikakis et al [105]. This requires using existential quantification to abstract names in recursive data structures and including a means to forget names assigned to automata that go out of scope (e.g., in each iteration of a loop).

Dependently typed functions and evidence. (T-ABS) gives functions a dependent type, $(x{:}t) \rightarrow t'$. Here, x names the formal parameter and is bound in t'. When a function is applied, (T-APP) substitutes the actual argument e' for x in the return type. Thus, given a function f that has type $(\mathsf{debt} : \mathit{Int}) \rightarrow (\mathit{LEQ}\ \mathsf{debt}\ 10) \rightarrow t$, the application $(f\ 11)$ is given the type $(\mathit{LEQ}\ 11\ 10) \rightarrow t$. That is, the type of the second argument of f *depends on* the

term passed as the first argument. Note that although λAIR permits arbitrary expressions to appear in types, type checking the enforcement of an AIR policy is decidable because we never have to reduce expressions that appear in types. However, in order to enforce policies like the static information flow policy of Chapter 4, reduction of type-level expressions is, as in FABLE, critical.

Affine types for consistent state updates. Finally, we consider how the type system enforces the "use at most once" property of affine types. First, (T-NEW) introduces affine types by giving new automaton instances the type $\mathop{!}t^{\alpha}$. Values of affine type can be destructed in the same way as values of unrestricted type. For example, (T-APP) and (T-TAP) allow e to be applied irrespective of the affinity qualifier on e's type. However, we must make sure that variables that can be bound to affinely typed values are not used more than once. This is prevented by the type rules through the use of affine assumptions A, which lists the subset of variables with affine type in Γ which have not already been used. The use of an affine variable is expressed in the rule (T-XA), which types a variable x in the context of the single affine assumption x. To prevent variables from being used more than once, other rules, such as (T-APP), are forced to split the affine assumptions between their subexpressions. Affine assumptions are added to A by (T-ABS) using the function $a(x, k)$, where x is the argument to the function and k is the kind of its type. If the argument x's type has kind A then it is added to the assumptions, otherwise it is not. We include a weakening rule (T-WKN) that allows affine assumptions to be forgotten (and for additional names ε' to be consumed). Finally, the function $p(A, \varepsilon)$ is used to determine the affinity qualifier of an abstraction. If no affine assumptions from the environment are

80

used in the body of the abstraction ($A = \cdot$) and if no new automata are constructed in the body ($\varepsilon = \cdot$), then it is unrestricted. Otherwise, it has captured an assumption from the environment or encloses an affinely tracked automaton and should be called at most once.

Kinding judgment. In $\Gamma \vdash t :: K$, the rule (K-A) is standard. (K-N) allows a name to be associated with any affine type t. (K-AFN) checks an affinely-qualified type: types such as ¡¡t are not well-formed. (K-FUN) is standard for a dependent type system—it illustrates that x is bound in the return type t'. (K-UNIV) is mostly standard, except that we must also check that the constraint ε only contain names that are in scope. (K-DEP) checks the application of a dependent-type constructor. Here, we have to ensure that the type of the argument e matches the type of the formal. However, since e is a type-level expression, we check it in a context with the phase index $\varphi = $ type. Since types are erased at run time, type-level expressions are permitted, via (T-X-type), to treat affine assumptions intuitionistically. Erasure of types also allows us to lift the name constraints for type-level expressions e—(T-NC-type) allows any subset ε_1 of the names used in e to be forgotten.

3.4.3   Dynamic Semantics

Figure 3.6 defines the dynamic semantics of λAIR as a call-by-value, small-step reduction relation, using a left-to-right evaluation order. The form of the relation is :

$$M \vdash e \xrightarrow{\;l\;} e'$$

Equations, models, and certificates

equation	E	$::=$	$\mathscr{D} \rightsquigarrow e$	model	M	$::=$	$B : \vec{E} \mid M, M$
eqn. domain	$\mathscr{D}$	$::=$	$v \mid t \mid \mathscr{D}, \mathscr{D} \mid \cdot$	certificates	e	$::=$	$\ldots \mid [\![B]\!]^{\mathscr{D}}$

Values and evalutation contexts

values	v	$::=$	$D \mid v\,(v') \mid [\![B]\!]^{\mathscr{D}} \mid \lambda x{:}t.e \mid \Lambda\alpha{::}k.e \mid$ new v
eval ctxt	$\mathscr{E}$	$::=$	$\bullet \mid \bullet\,e \mid v\,\bullet \mid \bullet\,[t] \mid \bullet\,(e) \mid v\,(\bullet) \mid$ case $\bullet$ of$\ldots \mid$ new $\bullet$

$M \vdash e \xrightarrow{l} e'$ 　　　　　　　　　　　　　　An expression e reduces to e' recording l in the trace.

$$\frac{M \vdash e \xrightarrow{l} e' \quad e' \neq \bot}{M \vdash \mathscr{E} \cdot e \xrightarrow{l} \mathscr{E} \cdot e'} \text{(E-CTX)} \qquad \frac{M \vdash e \xrightarrow{l} \bot}{M \vdash \mathscr{E} \cdot e \xrightarrow{l} \bot} \text{(E-BOT)} \qquad \frac{}{M \vdash \bot \longrightarrow \bot} \text{(E-INF)}$$

$$\frac{e' = (x \mapsto v)\,e}{M \vdash \lambda x{:}t.e\ v \longrightarrow e'} \text{(E-APP)} \qquad \frac{e' = (\alpha \mapsto t)e}{M \vdash \Lambda\alpha{::}k.e\ [t] \longrightarrow e'} \text{(E-TAP)}$$

$$\frac{\textbf{if } (v \succ e_{pat} : \sigma) \textbf{ then } e = \sigma(e') \textbf{ else } e = e''}{M \vdash \text{ case } v \text{ of } \overrightarrow{x{:}t}.e_{pat} : e' \text{ else } e'' \longrightarrow e} \text{(E-CASE)} \qquad \frac{B{:}\vec{E} \in M}{M \vdash B \longrightarrow [\![B]\!]^{\cdot}} \text{(E-DELTA)}$$

$$\frac{B{:}\vec{E} \in M \quad \mathscr{D}, v \rightsquigarrow e \in \vec{E} \quad l = B : \mathscr{D}, v}{M \vdash [\![B]\!]^{\mathscr{D}}\ v \xrightarrow{l} e} \text{(E-B1)} \qquad \frac{B{:}\vec{E} \in M \quad \mathscr{D}, v \rightsquigarrow e \notin \vec{E}}{M \vdash [\![B]\!]^{\mathscr{D}}\ v \longrightarrow [\![B]\!]^{\mathscr{D}, v}} \text{(E-B2)}$$

$$\frac{B{:}\vec{E} \in M \quad \mathscr{D}, t \rightsquigarrow e \in \vec{E} \quad l = B : \mathscr{D}, t}{M \vdash [\![B]\!]^{\mathscr{D}}\ [t] \xrightarrow{l} e} \text{(E-B3)} \qquad \frac{B{:}\vec{E} \in M \quad \mathscr{D}, t \rightsquigarrow e \notin \vec{E}}{M \vdash [\![B]\!]^{\mathscr{D}}\ [t] \longrightarrow [\![B]\!]^{\mathscr{D}, t}} \text{(E-B4)}$$

$v \succ e_p : \sigma$ 　　　　　　　　　　　　　　　Pattern matching data constructors.

$$\frac{}{v \succ v : \cdot} \text{(U-ID)} \qquad \frac{}{v \succ x : x \mapsto v} \text{(U-VAR)} \qquad \frac{v \succ e :: \sigma \quad v' \succ \sigma\,e' : \sigma'}{v\,(v') \succ e\,(e') : \sigma, \sigma'} \text{(U-CON)}$$

Figure 3.6: Dynamic semantics of λAIR

This judgment claims that a term e reduces in a single step to e' in the presence of a model M that interprets the base terms in a signature. The security-relevant reduction steps are annotated with a trace element l, which is useful for stating our security theorem.

Following a standard approach for interpreting constants in a signature [85], we define a model M by axiomatizing the reductions of base-term function applications. The syntax of the model is shown M is shown at the top of Figure 3.6. A model M contains

equations $B : \mathscr{D} \rightsquigarrow e$, where $\mathscr{D}$ is a sequence of types and values. A simple example of an

equation is plus $1\ 2 \rightsquigarrow 3$, indicating that an application (plus $1\ 2$) reduces to 3 at runtime.

Base term functions in λAIR are also used to perform runtime checks that provide

evidence for the release conditions in an AIR policy. For example, a λAIR program can

perform a test (*leq x y*) to attempt to construct evidence of type *LEQ x y*. The model

equations for *leq* need to generate valid proof certificates for tests that succeed and throw

runtime errors for those tests that fail. Handling failures is relatively straightforward—

we simply include equations of the form *leq* : 4, $3 \rightsquigarrow \perp$ indicating that the expression

(*leq* 4 3) reduces to $\perp$, i.e., a runtime error. However, we also need a way to construct

proof certificate values that inhabit types like *LEQ* 3 4. Our solution is to introduce

special values $[\![B]\!]^{\mathscr{D}}$ to represent these certificates. For example, $[\![LEQ]\!]^{3,4}$ will be the

representation of a value that inhabits the type *LEQ* 3 4, and model equations of the form

leq : 3 4 $\rightsquigarrow [\![LEQ]\!]^{3,4}$ will serve to indicate that the application (*leq* 3 4) reduces at runtime

to a valid proof certificate. In practice, if we are in a purely type-safe setting, we could

choose an arbitrary value (like unit) to represent a proof certificates. However, a concrete

runtime representation for proof certificates can be of practical use if proofs need to be

checked at run time, e.g., when interfacing with type-unsafe code.

The values and evaluation contexts in λAIR are also defined in Figure 3.6. Note that

the base-term data constructors D are treated as values; e.g., constructors Succ and Zero

are both treated as values. Constructor applications like Succ (Zero) are also values. The

other values include certificates, abstractions, and new automaton instances.

The rules in the reduction relation $M \vdash e \xrightarrow{l} e'$ from (E-CTX) to (E-CASE) are

entirely standard. The pattern matching judgment $v \succ e_p : \sigma$ follows a similar judgment in

FABLE. The remaining rules manipulate base-term functions. In (E-DELTA), we reduce a base term B to a certificate $[\![B]\!]$ that serves a proof that a term inhabits the type given to B in the signature.

In (E-B1), we show how the application of a base term B to a sequence of types and terms $\mathscr{D}, v$ is reduced using an equation that appears in the model. The security-relevant actions in a program execution are the reduction steps that correspond to automaton state changes. As indicated earlier, each transition and release rule in a policy will be translated to a function-typed base term like Conf_coalition. Thus, every time we reduce an expression e using a base-term equation $B : \mathscr{D} \rightsquigarrow e'$, we record $l = B : \mathscr{D}$ in the trace: i.e., $M \vdash e \xrightarrow{B : \mathscr{D}} e'$.

(E-B3) handles the application of a type to a polymorphic base-term function and is identical in structure to (E-B1). One point to note about (E-B3): although we allow an equation in M to depend on the type argument t (i.e., it is free to perform an intensional analysis on t, potentially violating parametricity), our soundness theorem places constraints on the form of the model equations to ensure type safety.

Finally, the rules (E-B2) and (E-B4) handle partial applications of base terms. For example, the partial application reduces in several steps as shown below:

$$M \vdash leq\ 3 \longrightarrow [\![leq]\!] \longrightarrow [\![leq]\!]^3$$

An implementation of λAIR would, of course, take a less abstract approach to the semantics of base terms. For example, we could use enforcement policy functions in FABLE to produce unforgeable certificates for runtime tests (using FABLE's labeling and

unlabeling constructs). However, the abstract presentation here both keeps the presentation simple and allows us to prove a standard type-soundness theorem.

Theorem (Type soundness). *Given an environment $\Gamma = S, \alpha_1::\mathbb{N}, ..., \alpha_n::\mathbb{N}$ that only binds type names, such that $\Gamma; \cdot \vdash_{\mathsf{term}} e : t; \varepsilon$, and an interpretation M such that M and S are type-consistent, then $\exists e'.M \vdash e \xrightarrow{l} e'$ or e is a value. Moreover, if $M \vdash e \xrightarrow{l} e'$ then $\Gamma; \cdot \vdash_{\mathsf{term}} e' : t; \varepsilon$.*

Notice that the statement of this theorem relies on a hypothesis that the model M and the signature S are type-consistent. That is, patently type-unsafe model equations like $leq : 3\ 4 \rightsquigarrow 17$ are ruled out. Furthermore, in order to prove this theorem, we need a way to type certificates. That is, we need additional type rules that allow certificates like $[\![LEQ]\!]^{3,4}$ to be typed as $LEQ\ 3\ 4$—this is easily done. Appendix B defines these additional rules and contains a detailed proof sketch of this type-soundness theorem.

We have also mechanized the proof of soundness for $\lambda\textsc{air}$ using the Coq proof assistant [17]. Our formalization adapts a proof technique recently proposed by Aydemir et al [7]. In particular, we use a locally nameless approach for representing both term- and type-level bindings and rely on cofinite quantification to introduce fresh names. We rely on a set of libraries distributed by Aydemir et al. that provide basic support for working with environments and finite sets. Our Coq proof is complete, modulo a collection of identities about finite sets and context splitting. The proofs of these identities are beyond the capabilities of the decision procedures in the finite set libraries that we use and, without automation, we have found proofs of these identities in Coq to be tedious and time consuming. However, we expect it will be possible to devise specialized decision proce-

dures to automatically discharge the proofs of these identities. Our development of λAIR in Coq can be obtained from `http://www.cs.umd.edu/projects/PL/selinks`.

3.5 Translating AIR to λAIR

In this section, we show how we translate an AIR class to a λAIR API, describe how that API is to be used, and state our main security theorem.

3.5.1 Representing AIR Primitives

In order to enforce an AIR policy we must first provide a way to tie the policy to the program by protecting data with AIR automata. We must also provide a concrete representation for automata instances and a means to generate certificates that attest to the various release conditions that appear in the policy. These constructs are common to all λAIR programs and appear in the standard prelude S_0, along with the integers and pairs discussed in Section 3.4.1.

Protecting data. As indicated in Section 3.3, we include the following type constructor to associate an automaton with some data: ($Protected$::U $\rightarrow$ N $\rightarrow$ U). A term with type $Protected\ t\ \alpha$ is governed by the policy defined by an automaton instance with type-level name α. We would like to ensure that all operations on protected data are mediated by functions that correspond to AIR policy rules. For this reason, we do not provide an explicit data constructor for values of this type, ensuring that they cannot be destructed directly, say, via pattern matching. Values of this type are introduced only by assigning the appropriate types to functions that retrieve sensitive data. For instance, library functions

that read secret files from the disk can be annotated so that they return values with a protected type.

In addition to functions corresponding to AIR class rules, we can provide functions that allow a program to perform computations over protected values while respecting their security policies. We have explored such functions in Chapter 2 and showed that computations that respect a variety of policies (ranging from access control to information flow) can be encoded; we do not consider these further here.

Next, we discuss our representation of an AIR automaton—these include representations of the class that the automaton instantiates and the principal that owns the class.

Principals. The nullary constructor *Prin* is used to type principal constants *P*, i.e., $(Prin::\mathtt{U}), (P:Prin)$. As with integers, we need a way to test and generate evidence for acts-for relationships between principals. We include the dependent-type constructor and run-time check shown below.

$$(ActsFor::Prin \rightarrow Prin \rightarrow \mathtt{U})$$

$$(acts\_for:(x:Prin) \rightarrow (y:Prin) \rightarrow ActsFor\ x\ y)$$

AIR **classes.** A class consists of a class identifier *id* and a principal *P* that owns the class. The type constructors $(Id::\mathtt{U}), (Class::\mathtt{U})$ are used to type identifiers and classes. Classes are constructed using the data constructor $(\mathsf{Class}:Id \Rightarrow Prin \Rightarrow Class)$. The translation of an AIR class introduces nullary data constructors like US_Army_Confidential:*Id*

and US_Army:*Prin*, from which we can construct the class

$$\mathsf{USAC} = \mathsf{Class}\ (\mathsf{US\_Army\_Confidential})\ (\mathit{US\_Army})$$

Finally, we use a dependent-type constructor and run-time check to generate evidence that two classes are equal.

$$(\mathit{IsClass}{::}\mathit{Class} \rightarrow \mathit{Class} \rightarrow \mathtt{U}),$$

$$(\mathit{is\_class}{:}(x{:}\mathit{Class}) \rightarrow (y{:}\mathit{Class}) \rightarrow \mathit{IsClass}\ x\ y)$$

Class instances. Instances are typed using the *Instance*::U type constructor. Each instance must identify the class it instantiates and the current state of its automaton. For each state in a class declaration, we generate a data constructor in the signature that constructs an *Instance* from a *Class* and any state-specific arguments. For example, we have:

$$\mathsf{Init}{:}\mathit{Class} \Rightarrow \mathit{Instance}, \mathsf{Debt}{:}\mathit{Class} \Rightarrow \mathit{Int} \Rightarrow \mathit{Instance}$$

Thus the expression new Init (USAC) constructs a new instance of a class. According to (T-NEW), this expression has the affine type ¡*Instance*$^\alpha$, where the unique type-level name α allows us to protect some data with this automaton. Since we wish to allow data to be protected by automata that instantiate arbitrary AIR classes, we give all instances, regardless of their class, a type like ¡*Instance*$^\alpha$, for some α. This has the benefit of flexibility—we can easily give types to library functions that can return data (like file system objects) protected by automata of different classes. However, we must rely on a

88

run-time check to examine the class of an instance since it is not evident from the type.

The prelude includes the the following two elements to construct and type evidence about the class of an automaton instance:

$$ClassOf::\mathbb{N} \to Class \to \mathbb{U}$$

$$class\_of\_inst{:}\forall\alpha{::}\mathbb{N}.(x{:}\mathord{\textrm{¡}}Instance^{\alpha}) \to (\mathord{\textrm{¡}}Instance^{\alpha} * c{:}Class * ClassOf\ \alpha\ c)$$

The function *class_of_inst* extracts a *Class* value c from an instance named α and produces evidence (of type *ClassOf* α c) that α is an instance of c. The return type of this function is interesting for two reasons. First, because the returned value relates the class object in the second component of the tuple to the evidence object in the third component, we give the returned value the type of a *dependently typed tuple*, designated by the symbol $*$. Although we do not directly support these tuples, they can be easily encoded using dependently typed functions, as shown in Figure 2.2. Second, notice that even though *class_of_inst* does not cause a state transition, the first component of the tuple it returns contains an automaton instance with the same type as the argument x. This is a common idiom when programming with affine types; since the automaton instance is affine and can only be used once, functions like *class_of_inst* simply return the affine argument x to the caller for further use.

The following constructs in the prelude allow a program to inspect the current state of an automaton instance.

$$InState::\mathord{\textrm{¡}}Instance^{\circ} \to Instance \to \mathbb{U}$$

$$state\_of\_inst{:}\forall\alpha{::}\mathbb{N}.(x{:}\mathord{\textrm{¡}}Instance^{\alpha}) \to (z{:}\mathord{\textrm{¡}}Instance^{\alpha} * y{:}Instance * InState\ z\ y)$$

These constructs are similar to the forms shown for examining the class of an instance, but with one important difference. Since the state of an automaton is transient (it can change as transition rules are applied), we must be careful when producing evidence about the current state. This is in contrast to the class of an automaton which never changes despite changes to the current state. Thus, we must ensure that stale evidence about an old state of the automaton can never be presented as valid evidence about the current state.

The distinction between evidence about the class of an automaton and evidence about its current state is highlighted by the first argument to the type constructor *InState*. Unlike the first argument of the *ClassOf* constructor (which can be some type-level name α::N), the first argument of *InState* is an *expression* with an affine type $¡Instance^\circ$ (introduced via subsumption in (T-DROP)) that stands for an automaton instance that has been assigned some name. Using this form of subtyping allows us to use *InState* to type evidence about the current state of any automaton. An alternative would be to enhance the kind language by allowing type constructors to be have polymorphic kinds—we chose this form of subtyping to keep the presentation simpler.

As described further in the next subsection, functions that correspond to AIR rules take an automaton instance a_1 (say, in state Init) as an argument, and produce a new instance a_1' as a result (say, in state Debt(0)). Importantly, both a_1 and a_1' are given the type $¡Instance^\alpha$, i.e., the association between the type-level name α and the automaton instance is fixed and is invariant with respect to state transitions. Since the class of an automaton never changes (both a_1 and a_1' are instances of USAC) it is safe to give evidence about the class of an instance the type *ClassOf* α USAC, i.e., evidence about the class of an automaton can never become stale. On the other hand, evidence about the current

90

$$\boxed{src;dst;\Gamma \models_\rho \overrightarrow{\exists x{:}t.C}; e : t'} \qquad\qquad \text{Translation of an AIR rule}$$

$$\frac{\Gamma \vdash t :: k \qquad src;dst;S;\Gamma,x{:}t \models C : t' \qquad src;dst;S;\Gamma,x{:}t \models_\rho \overrightarrow{\exists x{:}t.C}; e : t''}{src;dst;S;\Gamma \models_\rho \exists x{:}t.C, \overrightarrow{\exists x{:}t.C}; e : (x{:}t) \to t' \to t''} \text{ (TR-COND)}$$

$$\frac{\Gamma, s'{::}N; \cdot \vdash e : Protected \; \alpha \; s; \varepsilon}{s;d;S;\Gamma \models_r \cdot; e : (\text{\textexclamdown}Instance^s \times \text{\textexclamdown}Instance^d \times Protected \; \alpha \; d)} \text{ (R-BODY)}$$

$$\frac{\Gamma, s'{::}N; \cdot \vdash e : t; \varepsilon}{s;d;S;\Gamma \models_t \cdot; e : (!Instance^s \times !Instance^d)} \text{ (T-BODY)}$$

Figure 3.7: Translating an AIR rule to a base-term function in a λAIR signature

state of the automaton can become stale. If we were to type this evidence using types of the form *InStateBad* α Init, then this evidence may be true of a_1 but it is not true of a'_1. Therefore, we make *InState* a dependent-type constructor to be applied to an automaton instance rather than a type-level name.

3.5.2 Translating Rules in an AIR Class

Appendix B defines a translation procedure from an AIR class to a λAIR signature. The key judgment in this translation is shown in Figure 3.7. In this section, we discuss the form of this judgment and describe its behavior by focusing on the translation of the rules in the example policy of Figure 3.2.

Each rule r in an AIR class is translated to a function-typed constant f_r in the signature. Each condition in a rule is represented as an argument to the function f_r—the translation of these conditions is the same for both release and transition rules. Where the translation of release and transition rules differs is in the construction of the final return type of the function f_r.

The translation judgment shown in Figure 3.7 uses a more compact notation for an AIR policy than the syntax of Figure 3.1. In particular, we treat both release and transition rules as a λAIR expression e prefixed by a list of binders and conditions $\overrightarrow{\exists x{:}t.C}$. The judgment $src; dst; \Gamma \models_\rho \overrightarrow{\exists x{:}t.C}; e : t'$, states that in a context where src and dst are the type-level names of the source and destination automata, and where Γ is a standard λAIR typing environment, the rule $\overrightarrow{\exists x{:}t.C}; e$ is translated to a base-term function with the type t'. The index ρ that appears on the turnstile differentiates transition rules ($\rho = t$) from release rules ($\rho = r$).

The rule (TR-COND) shows how a condition is translated. Its index ρ indicates that it applies to both release and transition rules. (TR-COND) peels off a single condition $\exists x{:}t.C$ from the list of conditions associated with a rule. The first premise checks that the type is well-formed. The second premise translates the condition C to the type t' that stands for the evidence of the condition. The third premise recurses through the rest of the release conditions. In the conclusion, we have the type t of the bound variable and the evidence type t' shown as arguments to a function whose return type t'' is the type produced by the recursive call.

The rules (R-BODY) and (T-BODY) translate release and transition rules respectively. We turn to the concrete example of the policy of Figure 3.2 to illustrate the behavior of these rules in detail.

Release rules. At a high-level, release rules have the following form. In response to a request to release data x, protected by instance a_1, to an instance a_2, the programmer must provide evidence for each of the conditions in the rule r. If such evidence can be

produced, then f_r returns a new automaton state a_1', downgrades x as specified in the policy, and returns x under the protection of a_2. As an example, consider the full type of the Conf_coalition rule shown below.

Conf_coalition :
1 $\forall src{::}\text{N}, dst{::}\text{N}, \alpha{::}\text{U}.$
2 $(a1{:}¡Instance^{src}) \rightarrow ¡(x{:}Protected\ \alpha\ src) \rightarrow ¡(a2{:}¡Instance^{dst}) \rightarrow$
3 $¡(e1{:}ClassOf\ src\ \text{USAC}) \rightarrow ¡(cd{:}Class) \rightarrow ¡(e2{:}ClassOf\ dst\ cd) \rightarrow$
4 $¡(e3{:}ActsFor\ (principal\ cd)\ Coalition) \rightarrow ¡(debt{:}Int) \rightarrow$
5 $¡(e4{:}InState\ a1\ (\text{Debt}\ (\text{USAC})\ (debt))) \rightarrow ¡(e5{:}LEQ\ debt\ 10) \rightarrow$
6 $\quad (¡Instance^{src} \times ¡Instance^{dst} \times Protected\ \alpha\ dst)$

The first two lines of this type were shown previously—x is the data to be released from the protection of automaton $a1$ (with type-level name src) to the automaton $a2$ (with type-level name dst). Since the argument $a1$ is affine, we require every function type to the right of $a1$ to also be affine, since they represent closures that capture the affine value $a1$. At line 3, the argument $e1$ is evidence that shows that the source automaton is an instance of the USAC class; cd is another class object, and $e2$ is evidence that the class of the destination automaton is indeed cd. At line 4, $e3$ stands for evidence of the first condition expression, which requires that the owning principal of the destination automaton acts for the *Coalition* principal. Line 5 contains evidence $e4$ that $a1$ is in some state Debt($debt$), where, from $e5$, $debt \leq 10$. The return type, as discussed before, contains the new state of the source automaton, the destination automaton $a2$ threaded through from the argument, and the data value x, downgraded according to the policy and with a type showing that it is protected by the dst automaton.

Transition rules. Each transition rule r in a class declaration is also translated to a function-typed constant f_r in the signature. However, instead of downgrading and co-

ercing the type of some datum *x*, a transition function only returns the new state of

the source automaton and an unchanged destination automaton. That is, instead of re-

turning a three-tuple like Conf_coalition, a transition rule like Conf_init returns a pair

$(\neg Instance^{src} \times \neg Instance^{dst})$, where the first component is the new state of the source

automaton and the second component is the unchanged destination automaton threaded

through from the argument. The full type of Conf_init is shown below.

Conf_init :
1 $\forall src$::N, dst::N, α::U.
2 $(a1{:}\neg Instance^{src}) \rightarrow \neg(x{:}Protected\ \alpha\ src) \rightarrow \neg(a2{:}\neg Instance^{dst}) \rightarrow$
3 $\neg(e1{:}ClassOf\ src\ \mathsf{USAC}) \rightarrow \neg(cd{:}Class) \rightarrow \neg(e2{:}ClassOf\ dst\ cd) \rightarrow$
4 $\neg(e4{:}InState\ a1\ \mathsf{Init}) \rightarrow (\neg Instance^{src} \times \neg Instance^{dst})$

A final point about the translation of an AIR class: It is also possible to translate an

AIR class D to a model that captures the runtime behavior of each rule in the class. We

focus on the signature S_D alone as this suffices for type checking. However, in order to

state our security theorem, we require constraining possible models M_D of S_D, so that M_D

is consistent with the AIR rules. For example, equations in M_D that represent transition

rules must return automaton states that correspond to the next states specified in the AIR

rules. Appendix B defines the consistency of a model M_D with an AIR policy precisely.

3.5.3   Programming with the AIR API

The program in Figure 3.8, a revision of the program in Figure 3.3, illustrates how

a client program interacts with the API generated for an AIR policy.

As previously, the first two lines represent boilerplate code, where we read a file and

its automaton policy and then block waiting for a release request. At line 3, we generate

```
1 let x_a1, a1:¡Instance^src = get_usac_file_and_policy () in
2 let a2:¡Instance^dst, channel = get_request () in
3 let a1,USAC,ca1_ev = class_of_inst [src] a1 in
4 let a2,ca2,ca2_ev = class_of_inst [dst] a2 in
5 let actsfor_ev = acts_for (principal ca2) Coalition in
6 let a1, Debt{USAC}{debt}, a1_state_ev = state_of_inst [src] a1 in
7 let debt_ev = leq debt 10 in
8 let a1',a2,x_a2 = Conf_coalition [src][dst][Int] a1 x_a1 a2
9                            ca1_ev ca2 ca2_ev actsfor_ev
10                           debt a1_state_ev debt_ev in
11   send [Int] [dst] channel x_a2
```

Figure 3.8: A λAIR program that performs a secure information release

evidence a1_class_ev that a1 is an instance of the USAC class and at line 4 we retrieve a2's

class ca2 and evidence ca2_ev that witnesses the relationship between ca2 and a2. At line

5, we check that the destination automaton is owned by a principal acting for the Coalition.

At lines 6 and 7 we check that a1 is in the state Debt{USAC}{debt}, for some value of

debt $\leq$ 10. If all the run-time checks (i.e., calls to functions like leq) succeed, then we

call Conf_coalition, instantiating the type variables, passing in the automata, the data to

be downgraded and evidence for all the release conditions. We get back the new state of

the *src* automaton a1', a2 is unchanged, and x_a2 which has type *Protected Int dst*. We

can give the channel a type such as *Channel Int dst*, indicating that it can be used to send

integers to the principal that owns the automaton *dst*. The send function can be given the

type shown below:

$$\text{send}:\forall\alpha::\text{U}, \beta::\text{N}.\textit{Channel } \alpha \ \beta \rightarrow \textit{Protected } \alpha \ \beta \rightarrow \textit{Unit}$$

This ensures that x_a1 cannot be sent on the channel. If the call to Conf_coalition succeeds,

then the downgraded x_a2 has type *Protected Int dst*, which allows it to be sent.

3.5.4 Correctness of Policy Enforcement

In this section, we present a condensed version of our main security theorem and discuss its implications. The full statement and proof can be found in Appendix B.

Theorem (Security). *Given all of the following: (1) an* AIR *declaration* D *of a class with identifier* C *owned by principal* P*, and its translation to a signature* S_D*; (2) a model* M_D *consistent with* S_D*; (3)* $\Gamma = S_D, src::\mathbb{N}, dst::\mathbb{N}, s:\textit{¡Instance}^{src}$*; (4)* $\Gamma; s \vdash_{\mathsf{term}} e : t; \varepsilon$ *where* $src \notin \varepsilon$*; and (5)* $M \vdash ((s \mapsto v)e) \xrightarrow{l_1} e_1 \ldots \xrightarrow{l_n} e_n$ *where* $v = \mathsf{new}\ \mathsf{Init}\ (\mathsf{Class}\ (C)\ (P))$*. Then the string* $l_1, \ldots, l_n$ *is accepted by the automaton defined by* D*.*

The first condition relies on our translation judgment (discussed in Section 3.5.2) that produces a signature S_D from a class declaration D. The second condition is necessary for type soundness. Conditions (3) and (4) state that e is a well-typed expression in a context with a single free automaton $s : \textit{¡Instance}^{src}$ and two type name constants src and dst. By requiring that $src \notin \varepsilon$ we ensure that e does not give the name src to any other automaton instance. This theorem asserts that when e is reduced in a context where s is bound to an instance of the C class in the Init state, then the trace $l_1, \ldots, l_n$ of the reduction sequence is a word in the language accepted by the automaton of D.

The trace acceptance judgment has the form $A; D \models l_1, \ldots, l_n; A'$, which informally states that an automaton defined by the class D, in initial state A, accepts the trace $l_1, \ldots, l_n$ and transitions to the state A'. Recall that the trace elements l_i record base terms B that stand for security-relevant actions and sets of values that certify that the action is permissible. The trace acceptance judgment allows a transition from A to A' only if each transition is justified by all the evidence required by the rules in the class. This condition

96

is similar to the one used by Walker [139].

3.6 Encoding FABLE in λAIR

The power of dependent types to express a kind of customized type system is well known [144, 17, 49]. The dependent typing features of λAIR and FABLE are no exception—they can also be used to customize a type system. In fact, one view of the policy functions in FABLE (as proposed in Chapter 2) is that they provide a means of introducing axioms into the type system to which an application program can appeal in order to prove the validity of certain type-level invariants. For example, the *sub* function in the policy of Figure 2.9 effectively introduces a subsumption rule into the type system. By calling this function, an application program can prove that a term that inhabits the type $t\{l\}$ also inhabits the type $t\{lub\ l\ m\}$.

λAIR provides a similar ability, while using a slightly different mechanism. Instead of using explicit policy functions, λAIR programs are checked in the presence of a signature S that asserts the existence of terms of a particular type; i.e., S includes elements of the form $B{:}t$ for some base term B and type t. Whereas in FABLE we chose to pay attention to the particular implementation of these terms, in λAIR we take a more abstract view in that the semantics of these base terms is simply axiomatized in terms of some model. For instance, the same *sub* policy function in FABLE can be represented in λAIR as base term with the appropriate type. λAIR programs can appeal to this base term (or axiom) in order to prove the same subsumption relation that can be proved using FABLE. The operational behavior of *sub* is not specified within the language; instead we just model

sub at runtime using a model that axiomatizes the set of possible reductions that result from an application of the *sub* function.

By directly modeling the implementation of these policy functions, FABLE provided us with a way to reason concretely about the correctness of specific policies. In this respect, FABLE is more expressive than λAIR. However, for situations in which we do not necessarily care about specific policy implementations, the more abstract approach in λAIR suffices.

Additionally, λAIR exceeds the expressiveness of FABLE in two important ways. First, the signature in a λAIR program is not limited to defining base terms only. This makes the type language used in a λAIR program customizable via the signature. In particular, the signature S also includes elements of the form $T::K$, which binds the type constructor T to a kind K. Thus, while FABLE "bakes in" specific type constructors like int, lab, and the labeling construct $t\{e\}$, such constructs can be introduced into λAIR by simply plugging in the appropriate bindings in the signature. The second way in which λAIR exceeds FABLE is in its use of affine types (and type-level names).

In the remainder of this section, we develop a λAIR signature, S_{FABLE}, that will allow us to embed FABLE in λAIR. We will argue informally that this embedding is faithful in the sense that all programs typeable in FABLE can be translated to equivalent programs in λAIR. However, this translation does not come for free. In particular, several of the typing rules in FABLE will be translated to base terms in S_{FABLE}. For instance, the rule that allows a term with type lab $\sim e$ to be subsumed to the type lab will be represented by the base term *thide* in S_{FABLE}. Whereas the FABLE type system is able to apply this subsumption nondeterministically, in λAIR, we will expect programs to include explicit

Type constructors	Base terms
$(Int::U)$,	$(0:Int), (S:Int \Rightarrow Int)$,
$(Lab::U)$,	$(Low:Lab), (High:Lab)$,
$(SLab::Lab \to U)$,	$(Tuple:Lab \Rightarrow Lab \Rightarrow Lab), \dots$
$(Labeled::U \to Lab \to U)$	$(tshow : (e:Lab) \to (SLab\ e)$,
	$(thide : (e':Lab) \to (e:SLab\ e') \to Lab$

Figure 3.9: S_{FABLE}: An embedding of FABLE in λAIR

invocations of the *thide* base term whenever subsumption is necessary. Thus, much as our encoding of an information flow type system in FABLE places a greater burden on the application programmer compared to programming in a special purpose system (they must insert the appropriate calls to policy functions), emulating FABLE within λAIR is also somewhat more burdensome than programming in FABLE directly.

3.6.1 S_{FABLE}: A λAIR Signature for FABLE

Figure 3.9 shows S_{FABLE}, the signature that embeds FABLE in λAIR. The standard base types in FABLE include the type int of integers. This is easily mapped to λAIR by including the corresponding nullary type constructor *Int* with the kind U—i.e., integer variables can be used an arbitrary number of times. The term representation of integers follow the standard Peano construction, i.e., a nullary constructor 0 to represent zero and a constructor *S*, which when applied as *S n* will denote the successor of its *Int*-typed argument *n*.

The type of labels in FABLE is represented next. The nullary constructor *Lab* stands for the FABLE type lab. In FABLE we represented label terms by arbitrary applications of constructors *C*, where *C* is a meta-variable ranging over all possible constructors. In λAIR our approach is to define a set of constructors for the *Lab* type, i.e., an approach that

closely resembles the definition of variant types in a language like ML. Thus, we include the nullary term constructors like *Low* and *High* to represent the labels from a standard two-point lattice. We can also represent more complex label constructors using variant types in λAIR. For instance, we include the *Tuple* constructor, which can be applied to two *Lab*-typed values to produce a label e.g., *Tuple Low High* can be given the type *Lab*.

FABLE also includes the type $\mathsf{lab} \sim e$, a singleton type inhabited only by the value v which is a normal form of e. This type is represented in λAIR using the dependent-type constructor *SLab*. This constructor can be applied to any *term argument* that has the type *Lab* and produces a type of unrestricted kind. For example, the type $\mathsf{lab} \sim High$ in FABLE is represented by the type-constructor application *SLab High*.

We also need a way to represent the three typing rules in FABLE that handle labels. These rules are reproduced below.

$$\frac{\Gamma \vdash_c e_i : \mathsf{lab}}{\Gamma \vdash_c C(\overrightarrow{e}) : \mathsf{lab} \sim C(\overrightarrow{e})} \ \text{(T-LAB)} \qquad \frac{\Gamma \vdash_c e : \mathsf{lab}}{\Gamma \vdash_c e : \mathsf{lab} \sim e} \ \text{(T-SHOW)} \qquad \frac{\Gamma \vdash_c e : \mathsf{lab} \sim e'}{\Gamma \vdash_c e : \mathsf{lab}} \ \text{(T-HIDE)}$$

The (T-LAB) rule, the introduction form for labels, is handled in λAIR via each of the type constructors for labels. That is, the kind given to constructors like *Low*, *High* and *Tuple* directly encode the (T-LAB) rule in its kind, e.g., *Tuple* requires that each of its arguments is itself a *Lab*-typed value. However, one difference is that whereas (T-LAB) introduces a term with a singleton type $\mathsf{lab} \sim e$, the type constructors in S_{FABLE} introduce terms with a less precise type *Lab*. However, this precision is easily recovered via an

application of the base terms that correspond to the (T-SHOW) and (T-HIDE) rules.

The (T-SHOW) rule is represented by the base term *tshow*, a dependently typed function. This function represents a subsumption rule that allows any term *e* of type *Lab* to be used at the type *SLab e*, which is identical to the form of the (T-SHOW) rule. The *thide* dependently typed function models the (T-HIDE) type rule. Given a term *e* of type *SLab e'* (for any *e'* specified in the first argument), *thide* allows *e* to be used at the type *Lab*. One point to note here is that the *thide* expects *e'* as its first term argument, even though the runtime behavior of *thide* is independent of *e'*. It is exactly in this kind of situation that phantom variables (introduced in Chapter 2) are useful. An extension of λAIR with phantoms would permit us to express that *thide* is parametric in the type index *e'*.

Finally, we use the *Labeled* constructor to represent FABLE types of the form $t\{e\}$. This constructor is analogous to the *Protected* constructor from Chapter 3, except the here, sensitive resources will be protected by labels *e* rather than the type-level name *N* of some AIR automaton. For example, the FABLE type $\mathsf{int}\{High\}$ is represented in λAIR using the type-constructor application *Labeled Int High*.

The remaining type rules in FABLE can be represented by the built-in type rules of λAIR. For instance, the (T-CONV) rule in FABLE that allows type-level terms to be reduced corresponds to the (T-CONV) rule that also appears in λAIR. (See the full static semantics of λAIR in Figures B.1 and B.2 for a definition of the (T-CONV) rule in λAIR.) Similarly, the rules for abstractions, applications, and pattern matching are subsumed by the corresponding rules in λAIR.

The FABLE constructs that we do not model are the unlabeling and relabeling op-

101

erators. Recall that these operators can only be used within FABLE policy functions. In λAIR we will represent policy functions using base terms in the signature, for which we only give an abstract specification in terms of types—we simply axiomatize the operational behavior of these base terms in the language, which eliminates the need for labeling operators.

3.7   Concluding Remarks

This chapter has presented AIR, a simple policy language for expressing stateful information release policies. We have defined a core formalism for a programming language called λAIR, in which stateful authorization policies like AIR can be certifiably enforced. Additionally, we have shown how the type system of FABLE can be embedded within λAIR. As a result, λAIR can also be used to enforce the access control, provenance and information flow policies that were developed in Chapter 2.

A limitation of λAIR is that its programming model is still purely functional. Although we show how to model purely functional state updates in λAIR using affine types, we would also like to be able to work with a richer programming model that allows the direct manipulation of mutable state. In the next chapter, we extend λAIR with mutable references and show how the resulting calculus, FLAIR, can be used to enforce information flow policies while accounting for side effects.

4.   Enforcing Policies for Stateful Programs

The preceding chapters have demonstrated that a simple, lightweight form of dependent typing can be used to specify and implement the enforcement of a range of security policies. We have illustrated the expressiveness of our approach by showing that several special-purpose security type systems can be encoded within FABLE and its relative, λAIR. We have argued that our approach has several benefits, prominent among which are flexibility and customizability. In particular, FABLE makes it possible to enforce a range of security policies in a manner best suited to the specific requirements at hand. However, our results so far apply only to purely functional programs, undermining the claim that our approach promises to be widely applicable to realistic programs.

Our focus on purely functional programs reveals a fundamental conflict. As functions, a purely functional program is not permitted to have any side effect. But, to write programs that are useful, one needs to cause some form of side effect, e.g., some output has to be printed to the screen. To omit side effects from our model of security is to ignore the obvious—clearly we wish to control what messages a program is permitted to send over the network.

In this chapter, we seek to redress this deficiency by demonstrating how a version of the λAIR language can be used to track and secure a canonical form side effects. In particular, we focus on equipping a core functional calculus with references to a mutable

store and enforcing an information flow policy for programs in this language. Such a policy partitions the set of memory locations into public and secret locations. Intuitively, the correct enforcement of this policy must ensure that no information about secret values is leaked into public locations.

Whereas the FABLE approach can easily be adapted to ensure that secret information is not leaked into a public memory location via direct assignment, this is not sufficient to establish correct enforcement of an information flow policy. Unfortunately, dependences on secret data can be revealed without requiring a direct assignment of secrets to a public location—the leak can take place via a so-called *implicit* or *indirect* flow. The classic example of such a leak is illustrated by the example program below, where h is a high-security boolean and a *lloc* is a low-security location.

$$\text{if } (h) \text{ then } lloc := \text{true}$$
$$\text{else } lloc := \text{false}$$

Although the secret value h is never directly assigned to the public location *lloc*, clearly this program successfully copies h to *lloc*. Plugging this form of leak will be the main challenge faced by our enforcement mechanism. The key idea will be to use affine types in λAIR to provide evidence that the set of implicit dependences at an assignment to a location l is not more secret than the location itself.

This chapter makes the following contributions:

- Section 4.1 defines FLAIR, a straightforward extension of λAIR with references and the final formal calculus of this dissertation. We state a type-soundness theorem for FLAIR. The proof of this theorem, an extension of the corresponding proof for λAIR, is in Appendix C.

104

- Accounting for side effects when enforcing an information flow policy involves a number of subtle issues. In Section 4.2, we present Core-ML, a simplification of a calculus of the same name proposed by Pottier and Simonet [104], as a model for the purely static enforcement of information flow controls in an ML-like, mostly functional language. Core-ML serves as a high-level specification of correct enforcement of information flow, in the familiar language of natural deduction.

- The main result of this chapter (Section 4.4) is an encoding of the Core-ML type system in FLAIR. Our security theorem asserts that every type correct FLAIR program using our encoding enjoys a noninterference property analogous to the corresponding property for Core-ML programs. Although this result demonstrates that FLAIR is expressive enough to enforce policies for stateful programs, our encoding is relatively complex. The difficulty of programming at the source level in FLAIR while enforcing such a complex policy is substantial. Thus, our contention in Chapter 3 that λAIR is likely to be more useful as an intermediate language is only more pronounced with the static enforcement of information flow in FLAIR.

4.1   FLAIR: Extending λAIR with References

In this section, we define FLAIR, an extension of λAIR with references. The extension is mostly standard and is shown in Figure 4.1.

The extensions to the syntax of λAIR include a *Unit* type and the corresponding value (); a value form ℓ which stands for a memory reference literal; a dereferencing operation $!e$; and an assignment operation $e\_1 := e\_2$. References to a value of type t are

Terms	e	$::=$	$\dots \mid () \mid \ell \mid \,!e \mid e_1 := e_2$
Types	t	$::=$	$\dots \mid Unit \mid \mathsf{ref}\ t$
Typing environment	Γ	$::=$	$\dots \mid \Gamma, \ell{:}t$
Store	Σ	$::=$	$(\ell, v) \mid \Sigma, \Sigma \mid \cdot$
Values	v	$::=$	$\dots \mid () \mid \ell$
Eval ctxt	$\mathscr{E}$	$::=$	$\dots \mid \,!\bullet \mid \bullet := e \mid v := \bullet$

$\Gamma; A \vdash_\varphi e : t; \varepsilon$ Extensions to typing judgment

$$\frac{\Gamma(\ell) = t}{\Gamma; \cdot \vdash_{\mathsf{term}} \ell : t; \cdot} \text{(T-LOC)} \qquad \frac{\Gamma; A \vdash_{\mathsf{term}} e : \mathsf{ref}\ t; \varepsilon}{\Gamma; A \vdash_{\mathsf{term}} \,!e : t; \varepsilon} \text{(T-DREF)}$$

$$\frac{\Gamma; A \vdash_{\mathsf{term}} e_1 : \mathsf{ref}\ t; \varepsilon_1 \qquad \Gamma; A \vdash_{\mathsf{term}} e_2 : t; \varepsilon_2}{\Gamma; A \vdash_{\mathsf{term}} e_1 := e_2 : Unit; \varepsilon_1 \uplus \varepsilon_2} \text{(T-ASN)}$$

$\Gamma \vdash t :: K$ Extensions to kinding judgment

$$\frac{}{\Gamma \vdash Unit :: \mathtt{U}} \text{(K-UNIT)} \qquad \frac{\Gamma \vdash t :: \mathtt{U}}{\Gamma \vdash \mathsf{ref}\ t :: \mathtt{U}} \text{(K-REF)}$$

$M \vdash (\Sigma, e) \xrightarrow{l} (\Sigma', e')$ Dynamic semantics in the presence of a model M

$$\frac{M \vdash e \xrightarrow{l} e' \qquad e' \neq \bot}{M \vdash (\Sigma, \mathscr{E} \cdot e) \xrightarrow{l} (\Sigma, \mathscr{E} \cdot e')} \text{(E-PURE-CTX)} \qquad \frac{M \vdash e \xrightarrow{l} \bot}{M \vdash (\Sigma, \mathscr{E} \cdot e) \xrightarrow{l} (\Sigma, \bot)} \text{(E-PURE-BOT)}$$

$$\frac{M \vdash (\Sigma, e) \xrightarrow{l} (\Sigma', e') \qquad e' \neq \bot}{M \vdash (\Sigma, \mathscr{E} \cdot e) \xrightarrow{l} (\Sigma', \mathscr{E} \cdot e')} \text{(E-CTX)} \qquad \frac{M \vdash (\Sigma, e) \xrightarrow{l} \bot}{M \vdash (\Sigma, \mathscr{E} \cdot e) \xrightarrow{l} (\Sigma, \bot)} \text{(E-BOT)}$$

$$\frac{(\ell, v) \in \Sigma}{M \vdash (\Sigma, !\ell) \longrightarrow (\Sigma, v)} \text{(E-DEREF)} \qquad \frac{\Sigma = \Sigma_1, (\ell, v'), \Sigma_2}{M \vdash (\Sigma, \ell := v) \longrightarrow (\Sigma_1, (\ell, v), \Sigma_2, ())} \text{(E-ASN)}$$

Figure 4.1: Syntax and semantics of FLAIR (Extends λAIR with references)

given the type ref t.

For simplicity we do not include a dynamic allocation construct. While adding dynamic allocation to FLAIR is straightforward, developing a policy to control allocation effects is somewhat tedious (although it is still possible). Rather than further complicate

the information flow policy of Figure 4.5, we just omit dynamic allocation.

The typing judgment $\Gamma; A \vdash_\varphi e : t; \varepsilon$ extends the judgment of the same form found in Figure 3.5. In this case, the environment Γ is extended to include bindings of memory locations ℓ to their reference types. Since we do not support dynamic allocation, we expect all memory locations to be bound to their types in the initial typing environment. Recall that the phase index φ distinguishes the typing of term-level from type-level expressions and that ε is a constraint that manages the usage of type-level names—ε plays no significant role in this extension to λ_{AIR}.

The typing judgments are mostly standard. Our one concern is to ensure that effectful expressions (i.e., expressions that use references to manipulate the store) do not appear in type-level expressions. This is standard in a dependent type system. (Hoare type theory [91], a dependent type system with stateful higher-order functions, is a notable exception.)

To enforce the effect-free restriction on type-level expressions, (T-LOC), (T-DREF) and (T-ASN) are all applicable only when typing a term-level expression. (T-LOC) simply looks up the type of the location in the environment. (T-DREF) gives $!e$ the type of the referent of e. (T-ASN) requires the value being written to have the same type as the contents of the location and gives the result of an assignment the *Unit* type. It also propagates the constraints on names ε_1 and ε_2 as enforced in the other rules of the system.

The extensions to the kinding judgment are next. (K-UNIT) is straightforward. In (K-REF) we impose a restriction that values in the store be given an unrestricted type. Allowing affine values to escape into the store is common (e.g., affine types in Cyclone [124] are used primarily to track the usage of the heap-directed pointers). Although standard,

the machinery required to support this feature is somewhat cumbersome, e.g., we would need to ensure that it there are no unrestricted references to affine values in the store. To keep the formalism simple, we exclude this feature. We would expect a practical implementation of FLAIR to allow affine values to be stored in the heap.

Figure 4.1 concludes with an extension to the dynamic semantics of λAIR. In Chapter 3, we defined the operational semantics of (purely functional) λAIR as a relation of the form $M \vdash e \overset{l}{\longrightarrow} e'$, where M is a model defining the reduction of base-term applications. Here, we extend the relation to define a reduction of a configurations (Σ, e), consisting of a store Σ and an expression e.

The rules (E-PURE-CTX) and (E-PURE-BOT) are congruences that reduce the stateful reduction relation to the purely functional relation for sub-terms that do not require the store Σ. (E-CTX) and (E-BOT) are the stateful versions of the first two congruences. The only new base rules in the dynamic semantics are (E-DEREF), which simply looks up the value v stored at a location ℓ when ℓ is dereferenced, and (E-ASN), which updates the store at ℓ with the new value v and reduces the expression to the unit value.

Appendix C extends the soundness result of λAIR to include the store manipulation constructs of FLAIR. The statement of the theorem appears below.

Theorem (Type soundness)**.** *Given all of the following:*

1. *A well-formed environment Γ of type names and memory locations, with signature S, $\Gamma = S, \alpha_1::\mathbb{N}, \ldots, \alpha_n::\mathbb{N}, \ell_1:t_1, \ldots, \ell_m:t_m$*

2. *A type correct expression e such that $\Gamma; \cdot \vdash_{\mathsf{term}} e : t; \varepsilon$, for some t and ε.*

3. *An interpretation M such that M and S are type consistent.*

4. A store Σ such that $\Gamma \models \Sigma$.

Then, $\exists e', \Sigma'. M \vdash (\Sigma, e) \xrightarrow{l} (\Sigma', e')$ or e is a value.

Moreover, if $M \vdash (\Sigma, e) \xrightarrow{l} (\Sigma', e')$ then $\Gamma; \cdot \vdash_{\mathsf{term}} e' : t; \varepsilon$ and $\Gamma \models \Sigma'$.

This theorem is nearly identical to the soundness theorem of λAIR. The only difference is that in FLAIR, we guarantee that the store remains well-typed as a term evaluates.

4.2 A Reference Specification of Information Flow

With the ability to write programs with mutable state in FLAIR, we can begin to address the problem of constructing policies that control information flows that can occur via side effects. But first, we make precise the semantics of static information flow by presenting a simple information flow type system for Core-ML, a core subset of an ML-like language proposed by Pottier and Simonet [104].

4.2.1 Information Flow for Core-ML

Figure 4.2 defines the syntax and static semantics of Core-ML, a minimal core of an ML-like language that enforces static information flow controls. This system (the purely functional fragment of which appears in Appendix A) is a simplification of a type system proposed by Pottier and Simonet [104], and is established as correctly enforcing a standard noninterference property for mostly functional programs that can manipulate a mutable store.

The syntax of Core-ML appears at the top of Figure 4.2. Expressions include variables x and the value forms for units, booleans, memory locations ℓ, and abstractions.

Expressions	e	$::=$	$x \mid () \mid \mathsf{true} \mid \mathsf{false} \mid \ell \mid \lambda x.\mathsf{e}$
		$\mid$	$\mathbf{if}\ (\mathsf{e})\ \mathbf{then}\ \mathsf{e}\ \mathbf{else}\ \mathsf{e} \mid \mathsf{e}_1\mathsf{e}_2 \mid \mathsf{e}_1 := \mathsf{e}_2 \mid !\mathsf{e}$
Types	t	$::=$	$\mathsf{unit} \mid \mathsf{bool}^{\mathtt{l}} \mid (\mathsf{t}_1 \xrightarrow{pc} \mathsf{t}_2)^{\mathtt{l}} \mid \mathsf{ref}^{\mathtt{l}}\ \mathsf{t}$
Labels	$\mathtt{l}, pc$	$::=$	$L \mid H$
Environment	Γ	$::=$	$x{:}\mathsf{t} \mid \ell{:}\mathsf{t} \mid \Gamma,\Gamma \mid \cdot$

$\mathtt{l} \lhd \mathsf{t}$ Guards

$$\mathtt{l} \lhd \mathsf{unit} \qquad \frac{\mathtt{l} \sqsubseteq \mathtt{l}'}{\mathtt{l} \lhd \mathsf{bool}^{\mathtt{l}'}} \qquad \frac{\mathtt{l} \sqsubseteq \mathtt{l}'}{\mathtt{l} \lhd (\mathsf{t} \xrightarrow{pc} \mathsf{t}')^{\mathtt{l}'}} \qquad \frac{\mathtt{l} \sqsubseteq \mathtt{l}'}{\mathtt{l} \lhd \mathsf{ref}^{\mathtt{l}'}\ \mathsf{t}} \qquad \text{where } L \sqsubseteq H$$

$\mathsf{t}\ \mathsf{ok}$ Well-formed types

$$\mathsf{unit}\ \mathsf{ok} \qquad \mathsf{bool}^{\mathtt{l}}\ \mathsf{ok} \qquad \frac{\mathsf{t}\ \mathsf{ok} \quad \mathsf{t}'\ \mathsf{ok}}{(\mathsf{t} \xrightarrow{pc} \mathsf{t}')^{\mathtt{l}}\ \mathsf{ok}} \qquad \frac{\mathsf{t}\ \mathsf{ok} \quad \forall \mathtt{l}'.\mathtt{l}' \lhd \mathsf{t} \Rightarrow \mathtt{l}' \sqsubseteq \mathtt{l}}{\mathsf{ref}^{\mathtt{l}}\ \mathsf{t}\ \mathsf{ok}}$$

$\Gamma; pc \vdash_{ML} \mathsf{e} : \mathsf{t}$ Core-ML typing

$$\Gamma; pc \vdash_{ML} () : \mathsf{unit}\ \text{(ML-UNIT)} \qquad \Gamma; pc \vdash_{ML} x : \Gamma(x)\ \text{(ML-VAR)} \qquad \Gamma; pc \vdash_{ML} \ell : \Gamma(\ell)\ \text{(ML-LOC)}$$

$$\Gamma; pc \vdash_{ML} \mathsf{true} : \mathsf{bool}^{\mathtt{l}}\ \text{(ML-T)} \qquad \Gamma; pc \vdash_{ML} \mathsf{false} : \mathsf{bool}^{\mathtt{l}}\ \text{(ML-F)}$$

$$\frac{\Gamma; pc \vdash_{ML} \mathsf{e} : \mathsf{bool}^{\mathtt{l}} \quad \Gamma; pc \sqcup \mathtt{l} \vdash_{ML} \mathsf{e}_1 : \mathsf{t} \quad \Gamma; pc \sqcup \mathtt{l} \vdash_{ML} \mathsf{e}_2 : \mathsf{t} \quad \mathtt{l} \lhd \mathsf{t}}{\Gamma; pc \vdash_{ML} \mathbf{if}\ (\mathsf{e})\ \mathbf{then}\ \mathsf{e}_1\ \mathbf{else}\ \mathsf{e}_2 : \mathsf{t}}\ \text{(ML-IF)}$$

$$\frac{\Gamma; pc \vdash_{ML} \mathsf{e}_1 : \mathsf{ref}^{\mathtt{l}}\ \mathsf{t} \quad \Gamma; pc \vdash_{ML} \mathsf{e}_2 : \mathsf{t} \quad pc \sqsubseteq \mathtt{l}}{\Gamma; pc \vdash_{ML} \mathsf{e}_1 := \mathsf{e}_2 : \mathsf{unit}}\ \text{(ML-UPD)}$$

$$\frac{\Gamma; pc \vdash_{ML} \mathsf{e}_1 : \mathsf{ref}^{\mathtt{l}}\ \mathsf{t}}{\Gamma; pc \vdash_{ML} !\mathsf{e}_1 : \mathsf{t}}\ \text{(ML-DREF)} \qquad \frac{\Gamma, x{:}\mathsf{t}; pc' \vdash_{ML} \mathsf{e} : \mathsf{t}'}{\Gamma; pc \vdash_{ML} \lambda x.\mathsf{e} : (\mathsf{t} \xrightarrow{pc'} \mathsf{t}')^{\mathtt{l}}}\ \text{(ML-ABS)}$$

$$\frac{\Gamma; pc \vdash_{ML} \mathsf{e}_1 : (\mathsf{t} \xrightarrow{pc'} \mathsf{t}')^{\mathtt{l}} \quad \Gamma; pc \vdash_{ML} \mathsf{e}_2 : \mathsf{t} \quad \mathtt{l} \lhd \mathsf{t}' \quad pc \sqcup \mathtt{l} \sqsubseteq pc'}{\Gamma; pc \vdash_{ML} \mathsf{e}_1\mathsf{e}_2 : \mathsf{t}'}\ \text{(ML-APP)}$$

Subtyping

$$\mathsf{bool}^{\oplus} \qquad\qquad (\odot \xrightarrow{\odot} \odot)^{\oplus} \qquad\qquad \mathsf{ref}^{\odot}\ \odot$$

Figure 4.2: Core-ML syntax and typing

Notice that we do not decorate bound variables with their types; the static semantics of

Core-ML "guesses" these types. The non-value forms include a conditional statement,

function application, assignment, and dereference. Although not strictly necessary, we include booleans and conditionals since they are convenient for illustrating indirect flows. For simplicity, as in FLAIR, we do not support dynamic allocation of memory and assume instead that all memory locations are statically known.

Core-ML types may be decorated with labels l drawn from the two-point lattice, $L \sqsubseteq H$. Unit values carry no sensitive information and so these types are unlabeled. The type bool^H classifies high-confidentiality boolean values. Function types are $(t \xrightarrow{pc} t')^l$. The outer-most label l represents the confidentiality of the function, e.g., if a function literal is constructed in each branch of a conditional expression, then each literal must be at least as confidential as the guard expression. The annotation pc that appears above the function arrow represents a lower bound on the confidentiality of the memory locations that this function may update when it is applied. For example, $(\text{bool}^L \xrightarrow{H} \text{bool}^H)^H$ is the type of a high-confidentiality function from low- to high-confidentiality booleans. Additionally, the pc annotation H indicates that this function does not mutate any memory locations that are public, i.e., have confidentiality level L. The type given to a memory location is of the form $\text{ref}^l\ t$. (Note that although the typical notation for reference in the ML family of languages is t ref, we adopt the ref t for consistency with our notation in FLAIR.)

The static semantics begins by defining the security semantics of each type. The relation $l \lhd t$ (read: l guards t) requires t to have security level l or greater. It is used to record a potential information flow, e.g., we will use this relation to ensure that the values returned by the branches of a conditional have a security level no less than the level of the guard.

Since unit values carry no information, they are guarded by all labels. Booleans and function types are standard—their security level is just defined by the outer-most security label. Reference types are somewhat more subtle. In a language with dynamic allocation, both the address and the contents of the location can carry sensitive information. For instance, if allocation occurs conditional on a high-security boolean value, the address chosen by the allocator can reveal information about the boolean. But, in our setting, since we make the simplifying assumption that all memory locations are known statically, we can adopt a simpler security model for reference types. We will interpret a reference type like $\mathsf{ref}^H\ \mathsf{bool}^H$ as the type of a memory location only visible to a principal with privilege to view high-security values. Under this interpretation, a type like $\mathsf{ref}^L\ \mathsf{bool}^H$ stands for a public memory location that holds a secret boolean value. We will treat such a type as ill-formed, where well-formedness of types is defined by the predicate t ok. The interesting case is the last one, which rules out a type like $\mathsf{ref}^L\ \mathsf{bool}^H$, since $H \lhd \mathsf{bool}^H$ and $H \not\sqsubseteq L$. (One might wonder why we even require a top-level label for a reference type, i.e., why not just have types like $\mathsf{ref}\ \mathsf{bool}^H$? The reason, which will become clear in the next section, is that this model illustrates a particularly close correspondence with the type of labeled references in our FLAIR encoding.)

In the typing judgment $\Gamma; pc \vdash_{ML} \mathsf{e} : \mathsf{t}$, the environment Γ binds variables and memory locations to their types, as usual. The pc element is a label representing the confidentiality of the context in which e occurs. The judgment states that an expression e has the type t in an environment Γ, and that it does not effect memory at a confidentiality level lower than pc.

The first five rules in the judgment (ML-UNIT), (ML-VAR), (ML-LOC), (ML-T)

and (ML-F) are standard. The first interesting rule is (ML-IF), which type checks a conditional expression. When the guard e has confidentiality level l, the branches e_1 and e_2 execute in a context that carries information about the guard e. Thus, the (T-IF) checks each of these in a context where where the program counter is elevated to $pc \sqcup l$ (where $\sqcup$ is the least upper bound operator on the $L \sqsubseteq H$ lattice), i.e., the program counter is at least as secret as the guard. The other rules will ensure that the side effects of e_1 and e_2 are only to locations at least as secret as l. Since the value returned from the branches also depends on the guard expression, the final premise $l \lhd t$ requires that the type of the entire expression also be at least as secure as the guard.

(ML-UPD) type checks assignments through a reference. The first two premises ensure that the type of e_2 matches the type of the values that can be stored in the location e_1. The third premise $pc \sqsubseteq l$ enforces the program counter constraint; it ensures that the location being written to is at least as confidential as the context. (ML-DREF) allows any location to be dereferenced, irrespective of the value of the confidentiality of the context—the well-formedness of reference types ensures that the dereferenced value is at least as secret as l.

Finally, we turn to the rules for functions. (ML-ABS) guesses a type t for the abstracted variable x and type checks the body of the abstraction, e, in a context that includes an assumption for x. Additionally, a label pc' is chosen for the confidentiality of the program counter. This label serves as the lower bound for the memory effects of the body when the abstraction is applied and this bound is recorded above the arrow in the resulting type. Finally, we pick a confidentiality label l for the function itself. As an example, consider the program below, typed in a context $\Gamma = secret{:}\mathrm{bool}^H, hloc{:}\mathrm{ref}^H \mathrm{bool}^H$.

$$\text{if } (secret) \text{ then } \lambda x.\ hloc := \text{true}$$
$$\text{else } \lambda x.\ hloc := \text{false}$$

One legal type for this program is $(\text{unit} \xrightarrow{H} \text{unit})^H$. Since, each branch of the function carries information about the *secret* boolean, we must ensure that the values returned from the branches are at least as secure as *secret*. This explains the outermost label H reflecting the confidentiality of the function itself. The *pc* annotation on the function arrow is H, indicating that it writes only to high-confidentiality memory locations. This is desirable since when the function returned by this program is applied, the value of *secret* is written into the memory location *hloc*. Thus, when *secret*:bool^H, this program is secure only when *hloc* is also a high-confidentiality location.

The final typing rule (ML-APP) handles function application. The first two premises are straightforward—they simply require the type of the formal parameter to match the type of the actual argument. Since the value returned from the application carries information about the identity of the function that was applied, the third premise requires the returned value to be at least as secure as the function itself. Finally, the last premise ensures that the memory effects that occur in the function's body do not leak information about the program context ($pc \sqsubseteq pc'$) or about the identity of the function itself ($1 \sqsubseteq pc'$).

To illustrate how function applications are typed, we revisit our example program, this time typed in a context $\Gamma = secret{:}\text{bool}^H, lloc{:}\text{ref}^L \text{ bool}^L$.

$$\text{if } (secret) \text{ then } \lambda x.\ lloc := \text{true}$$
$$\text{else } \lambda x.\ lloc := \text{false}$$

Since the function bodies in the branches update low-security locations, the type of this program must be of the form $(\text{unit} \xrightarrow{L} \text{unit})^H$, i.e., the *pc* annotation on the function arrow is L, reflecting that this function (call it f), when applied, can effect a low-security

location. However, if we try to apply this function as $f()$, we find that if fails to type check. The final premise of (ML-APP) requires the *pc* annotation on f's type to be at least high as the confidentiality of f itself. This is a good thing—we expect to reject an application of f as insecure because, when f is applied, the value of the *secret* boolean is copied into the low-security location *lloc*.

The last section of Figure 4.2 shows a simple subtyping relation which extends the partial order over labels $L \sqsubseteq H$ to types. The notation, due to Pottier and Simonet, is a compact form for defining covariance ($\oplus$) and invariance ($\odot$). For instance, we have that bool^L is a subtype of bool^H. For simplicity, we do not permit contravariance of function arguments and covariance of function return types. Contravariance of the program counter annotation is also customary. Handling each of these features poses no significant challenge; Appendix A shows how to handle these constructs in the context of FABLE. Finally, one might have expected covariant subtyping on the label associated with a reference type. However, since our attacker model interprets a location of type $\mathsf{t}_1 = \text{ref}^L \text{bool}^L$ to be readable by the attacker, it is not safe to treat this as a subtype of $\mathsf{t}_2 = \text{ref}^H \text{bool}^L$. Again, for simplicity, we limit ourselves to invariance on the labels of reference types.

4.3 Tracking Indirect Flows in FLAIR using Program Counter Tokens

In this section, we describe an encoding of a policy in FLAIR that attempts to prevent illegal information flows through side effects. We present a signature S_{Flow}^0 which illustrates the basic idea behind our encoding, i.e., we use a special runtime value to represent the confidentiality of the program counter. Policy functions receive these program

Type constructors

Bool	::	U
LabeledRef	::	U → *Lab* → U
PC	::	*Lab* → U

Base terms

True	:	*Bool*
False	:	*Bool*
lub	:	$(x{:}Lab) \to (y{:}Lab) \to Lab$
update	:	$\forall \alpha{::}U.(l{:}Lab) \to PC\ l \to (x{:}LabeledRef\ (\text{ref}\ \alpha)\ l) \to (y{:}\alpha) \to Unit$
branch	:	$\forall \alpha{::}U.(l{:}Lab) \to PC\ l \to (m{:}Lab) \to (b{:}Labeled\ Bool\ m) \to$
		$\quad (t{:}PC\ (lub\ l\ m) \to \alpha) \to (f{:}PC\ (lub\ l\ m) \to \alpha) \to (Labeled\ \alpha\ m)$

Figure 4.3: S^0_{Flow}: An attempt to statically enforce information flow in FLAIR

counter values in their arguments, and, by giving these arguments appropriate types, we are able to place constraints on the indirect flows that occur in the program. In Section 4.4, we elaborate upon this basic idea to develop a complete signature S_{Flow}, which we then prove to successfully enforce a noninterference property for FLAIR programs.

4.3.1 S^0_{Flow}: A Sketch of a Solution

Figure 4.3 shows S^0_{Flow}, to be read as an extension of S_{FABLE}, which appears in Figure 3.9. The signature begins with a type constructor for booleans and the corresponding term constructors. We also include a *lub* function that computes the least upper bound of two labels. We then include a type constructor *LabeledRef*—we use this constructor to protect references by giving them types like *LabeledRef* (ref *t*) *High*. As usual, application programs cannot manipulate labeled references directly. Instead, they must call base-term functions that mediate all operations on these references.

The *update* function controls assignments through references. To prevent illegal direct flows, we simply require that the type α of the referent of x is the same as the type of the new value y to be stored in that location—this corresponds to the constraints specified in the first two premises of (ML-UPD) in Figure 4.2. The constraints of the third premise of (ML-UPD) (in order to protect against indirect flows) are captured by the first two arguments of *update*. We require the application to pass in a label l and a program counter token of type $PC\ l$. This token is a runtime value that is to be used as a proof that the program counter (at the point where *update* is called) is only as confidential as l. If we can ensure that such proofs are always constructed properly, we can be sure that a low-security reference is never updated in a high-security context.

The *branch* function mediates all conditional expressions where the guard is a labeled boolean value. The first two arguments of *branch* show a program counter token of type $PC\ l$, indicating that the program counter just before the conditional is executed is only as confidential as l. The next two arguments show the boolean guard b, which is labeled m—this much captures the first premise of (ML-IF). The arguments t and f represent the true and false branches, respectively. Since FLAIR is a call-by-value language, we need to suspend the execution of t and f when passing them as arguments to *branch*— this explains why both t and f are given function types. To ensure that the effects of the branches do not leak information about the guard, the third premise of (ML-IF) checks each branch in a context where the program counter is at least as secure as the guard. The constraint is captured in *branch* by the arguments of t and f, i.e., a program counter token of type $PC\ (lub\ l\ m)$. The idea is that given a token of type $PC\ (lub\ l\ m)$, t and f can never modify a location that is less secret than b since they cannot pass in the appropriate token

Γ = S^0_{Flow}, *initpc*:*PC Low*, *h*:*Labeled Bool High*, *l*:*Labeled Bool Low*,	
hloc:*LabeledRef* (ref *Bool*) *High*, *lloc*:(*LabeledRef* (ref *Bool*) *Low*),	

if (*h*) then *hloc* := true else *hloc* := false	let *tbranch* (*pc*:*PC High*) = *update pc hloc* true let *fbranch* (*pc*:*PC High*) = *update pc hloc* false *branch initpc h tbranch fbranch*
if (*h*) then *lloc* := true else *lloc* := false	let *tbranch* (*pc*:*PC High*) = *update initpc lloc* true let *fbranch* (*pc*:*PC High*) = *update initpc lloc* false *branch initpc h tbranch fbranch*
if (*l*) then *lloc* := true else *hloc* := false	let *tbranch* (*pc*:*PC Low*) = *update pc lloc* true let *fbranch* (*pc*:*PC Low*) = *update pc hloc* false *branch initpc l tbranch fbranch*

Figure 4.4: Attempting to track effects in some simple example programs

as an argument to *update*. Finally, the value returned by *branch* is the value of type α returned by either branch. This value is labeled *m* to reflect the dependence on the guard *b*, i.e., the return type captures the last premise of (ML-IF).

4.3.2 Example Programs that use S^0_{Flow}

To illustrate how S^0_{Flow} works, we consider three example programs in Figure 4.4. These programs are checked in the context Γ where *initpc* is an initial program counter token of type *PC Low*, *h* is a *High* boolean value, *l* is a *Low* boolean value, *hloc* and *lloc* are *High* and *Low* locations respectively. To the left, we show (in pseudo-code resembling Core-ML) programs that manipulate these variables, and at the right, we show equivalent

programs in FLAIR. Throughout the remainder of this chapter, our examples will use a more readable ML-like syntax rather than the primitive notation of FLAIR.

In the top-most section of the figure we have a secure program that examines the h value and, based on the result, writes to the secret location. The FLAIR program at the right, starting from the bottom, calls the *branch* function, passing in *initpc* as a token representing the initial program counter. Next, we pass in the boolean guard h, and two functions, *tbranch* and *fbranch*, representing each branch of the conditional. Both *tbranch* and *fbranch* expect program counter arguments of type *PC High*, indicating that they execute in a context control dependent on a *High* confidentiality value. In the bodies of these functions, we pass the *High* program counter, the location to be updated *hloc*, and the value to be stored to the *update* function. The entire program can be given the type *Labeled Unit High*, which reflects that the value computed by this program depends on the boolean expression h, which is labeled *High*.

Although S^0_{Flow} illustrates our basic strategy of tracking indirect flows by passing program counter tokens between the policy and application, it is flawed in two important ways. First, an application program can spoof the policy by re-using a stale program counter token, making information flow tracking unsound. Second, in some contexts, S^0_{Flow} prevents a program from causing side effects to high-security locations, even though such effects are always secure, i.e., in this respect, the policy enforced by S^0_{Flow} is more restrictive than a system like Core-ML. The next two examples in Figure 4.4 illustrate these two problems and hint at possible solutions.

The program in the middle part of Figure 4.4 is insecure because it updates *lloc*, a public location, in a context depending on the value of h, thereby exhibiting an indirect

119

flow from *High* to *Low*. However, the types in S^0_{Flow} are not precise enough to reject a transcription of this program to FLAIR (at the right) as type-incorrect. The trouble is that in *tbranch* (and *fbranch*) the application presents *initpc* as a program counter token to *update*. The *initpc* token represents the confidentiality of the program counter only at the start of the program. In the context of *tbranch*, the *initpc* token is no longer valid. We need a way to ensure that functions like *tbranch* only use the program counter tokens they receive as arguments, rather than re-using stale tokens. In Section 4.4 we show how affine types in FLAIR can be used to accomplish exactly this.

The final program in Figure 4.4 illustrates the second problem. At the left, we have a secure program that updates either *lloc* or *hloc* based on the value of *l*. However, using S^0_{Flow}, it is not possible to translate this to a type-correct FLAIR program. In *fbranch*, we have a function that expects a *PC Low* token as an argument, reflecting that it is control dependent only on the *Low*-security boolean value *b*. In the body of *fbranch*, we need to update *hloc*; but, the call to *update* fails to type check because the label of the program counter does not exactly match the label of *hloc*. To solve this problem, in Section 4.4, we will develop an encoding that allows an application program to use a program counter token of type *PC l* to produce capabilities that allow it to update any location that is labeled *l* or higher.

4.4   Enforcing Static Information Flow in FLAIR

In this section, we develop S_{Flow}, a signature that incorporates affine types and capabilities into S^0_{Flow} to accurately enforce an information flow policy in the presence of

side effects. We begin by providing a brief overview of our solution, which consists of the following main elements.

Representing a program counter with an affinely typed value. The main problem with our attempt to track indirect flows in S_{Flow}^0 was that the program counter token could be freely duplicated by the application program. This meant that the policy could not correctly ensure that the program always presented a program counter token that represented the sensitivity of the context in which the *update* policy function was called. This problem can be overcome by using affine types. Our encoding will give the *PC* type constructor the kind $Lab \to \text{A}$ (rather than $Lab \to \text{U}$, as was the case in S_{Flow}^0). So, a value of the affine type *PC e* will represent a proof that the program counter is no more secret than the label *e*. Since this value is affine, the type system will ensure that any given program point, there is only a single value that represents the current state of the program counter.

Generating capabilities from program counter tokens. In order to modify a location with label *l*, an application program must present a capability that demonstrates that the program counter at that point is not greater than *l*. We will use a value of the type *Cap l m* as this capability—a *Cap l m* value represents a proof that the label *l* is not less than *m*, the program counter label at that program point. In order to construct such capabilities, we provide a function *pc2cap* which, when given a *PC m* value, produces a capability of type *Cap (lub l m) m*, for some label *l*, while consuming the *PC m* value. Conversely, the program can call *cap2pc* to consume a capability *Cap l m* and retrieve a program counter token of type *PC m*.

Representing the *pc* bound on function types with an extra parameter. In a Core-ML

function type $(\text{bool}^L \xrightarrow{H} \text{bool}^H)^L$, the H *pc*-annotation over the arrow is a static guarantee that the function's effects are limited to the fragment of memory with security level at least H. So, our representation in S_{Flow} of this Core-ML function type will be a function with a formal parameter (*PC High* × *Labeled Bool Low*). That is, the argument of the function is a pair that includes the *PC High* value, from which it can generate capabilities that authorize it to update only *High*-confidentiality memory locations. However, since *PC e* is an affine type, when a value of type *PC High* is passed as an argument to a function, the caller can no longer use this value to generate capabilities to modify some other memory location (or even to call some other function). The solution to this problem is a standard idiom for programming with affine types—every function that receives a *PC e* value as an argument also returns this value back to the caller for further use. (This style of "threading" affine values through a function was introduced in Chapter 3.) Thus, the FLAIR representation of our example Core-ML function type is therefore

$$\textit{Labeled} \; ((\textit{PC High} \times \textit{Labeled Bool Low}) \rightarrow (\textit{PC High} \times \textit{Labeled Bool High})) \; \textit{Low}$$

4.4.1 S_{Flow}: A Signature for Static Information Flow

Figure 4.5 defines the signature for a policy that can statically control information flows in a FLAIR program. It is intended to be read as an extension of the S_{FABLE} signature (which encodes some FABLE primitives) presented in Figure 3.9. This signature, in effect, defines an interface that any correct *implementation* of a policy must satisfy. However, many possible implementations exist, of which only some are correct. In Section 4.4.4 we

Type abbreviation

Boxed l α ≡ $(PC\ l \times α)$

Type constructors

Bool	::	U
LabeledRef	::	$U \rightarrow Lab \rightarrow U$
$\times$	::	$A \rightarrow U \rightarrow A$
PC	::	$Lab \rightarrow A$
Cap	::	$Lab \rightarrow Lab \rightarrow A$

Base terms

True : *Bool*

False : *Bool*

Low : *Lab*

High : *Lab*

lub : $(x{:}Lab) \rightarrow (y{:}Lab) \rightarrow Lab$

Pair : $\forall α{::}A, β{::}U.\, α \Rightarrow β \Rightarrow α \times β$

join : $\forall α{::}U.(l{:}Lab) \rightarrow (m{:}Lab) \rightarrow$
$(x{:}Labeled\ (Labeled\ α\ l)\ m) \rightarrow Labeled\ α\ (lub\ l\ m)$

sub : $\forall α{::}U.(l{:}Lab) \rightarrow (m{:}Lab) \rightarrow (x{:}Labeled\ α\ l) \rightarrow Labeled\ α\ (lub\ l\ m)$

initpc : ¡$((l{:}Lab) \rightarrow PC\ l)$

pc2cap : $(l{:}Lab) \rightarrow (m{:}Lab) \rightarrow PC\ l \rightarrow Cap\ (lub\ l\ m)\ l$

cap2pc : $(l{:}Lab) \rightarrow (m{:}Lab) \rightarrow Cap\ l\ m \rightarrow PC\ m$

update : $\forall α{::}U.(l{:}Lab) \rightarrow (m{:}Lab) \rightarrow (cap{:}Cap\ l\ m) \rightarrow$
¡$(x{:}LabeledRef\ (\mathrm{ref}\ α)\ l) \rightarrow$ ¡$(y{:}α) \rightarrow Boxed\ m\ Unit$

deref : $\forall α{::}U.(l{:}Lab) \rightarrow (x{:}LabeledRef\ (\mathrm{ref}\ α)\ l) \rightarrow (Labeled\ α\ l)$

branch : $\forall α{::}U.(l{:}Lab) \rightarrow (pc{:}PC\ l) \rightarrow$ ¡$(m{:}Lab) \rightarrow$ ¡$(b{:}Labeled\ Bool\ m) \rightarrow$
¡$(t{:}Boxed\ (lub\ l\ m)\ Unit) \rightarrow Boxed\ (lub\ l\ m)\ α) \rightarrow$
¡$(f{:}Boxed\ (lub\ l\ m)\ Unit) \rightarrow Boxed\ (lub\ l\ m)\ α) \rightarrow$
$Boxed\ l\ (Labeled\ α\ m)$

apply : $\forall α{::}U, β{::}U.(l{:}Lab) \rightarrow (pc{:}PC\ l) \rightarrow$ ¡$(m{:}Lab) \rightarrow$
¡$(f{:}Labeled\ ((Boxed\ (lub\ l\ m)\ α) \rightarrow Boxed\ (lub\ l\ m)\ β)\ m) \rightarrow$
¡$(x{:}α) \rightarrow Boxed\ l\ (Labeled\ β\ m)$

Figure 4.5: S_{Flow}: A FLAIR signature to statically enforce an information flow policy

discuss the form of specific policy implementations (in terms of a model *M* for a FLAIR program) that satisfies this signature. This is in contrast to the approach in FABLE, where we focused directly on providing a concrete semantics for an enforcement policy.

Our encoding will make repeated use of packaging a value of type *t* along with a program counter value that, in the case of function's argument, will represent a lower bound on the function's side effects. Functions will also package their result with a program counter and return the pair to the caller. Throughout the remainder of this chapter, we will use the type abbreviation *Boxed e t* to stand for the tuple type $(PC\ e \times t)$. Since our policy encoding provides a purely static guarantee, an optimizing compiler can chose to erase the program counters and choose a runtime representation for values of type *Boxed e t* that is the same as the representation chosen for values of type *t*.

The binary type constructor $\times$ is used to give a type to a tuple consisting of a affine value and an unrestricted value and will be used to package a program counter with some other value. The corresponding term constructor is *Pair*. (However, we will use the more intuitive notation of (e_1, e_2) to construct pairs, and functions like *fst* and *snd* to destruct pairs.) Notice that *Pair* is polymorphic in α and β, the types given to each component of the tuple that it constructs. Since the first component of the tuple is affine, we require the tuple itself to be affine to ensure that the first component is not projected from it repeatedly.

The type constructor *LabeledRef* is used to protect memory locations; *Labeled* is used to protect all other values. The kinds of both constructors are identical. By distinguishing protected locations from other protected values, we will be able to define subsumption rules that apply only to labeled values, not to labeled references. (Recall that in Core-ML, labeled reference types were invariant in their labels, whereas covariant subtyping on the labels were permissible on other labeled values.) The terms *join* and *sub* define these subsumption rules. The *join* functions allows a type with multiple labels *l*

and *m* to be coerced to a type with a single label *lub l m*. As previously, *sub* is intended to encode a subsumption relation which takes as arguments a term *x* with type *Labeled α l* and a label *m* and allows *x* to be used at the type *Labeled α (lub l m)*. This is a restatement of the covariant subsumption rule, as $l \sqsubseteq m$ implies $l \sqcup m = m$. (Of course, as in Chapter 2, to simplify the construction of source programs we could use phantom label variables in the types of functions like *join* and *sub*. We omit phantom variables here for simplicity.)

Unlike S_{Flow}^0, in S_{Flow} we type the program counter token using the dependent-type constructor *PC* that constructs an affine type (a type with kind A) from a label. Rather than provide data constructors for the program counter token, the signature includes a function *initpc* that an application program can call to construct an initial program counter token. The type of *initpc* is affinely qualified (prefixed by '¡'). This ensures that the application program can construct an initial program counter token only once. For example, the application can call *initpc Low* to create a token of type *PC Low*. Recall that the program counter serves as a lower bound on the memory effects of a program. Thus, the application is free to initialize the program counter as *initpc High* as this only further restricts the set of permissible memory effects of the program.

The type of a capability that authorizes a program to update a memory location will be formed from the binary dependent-type constructor, *Cap*. Specifically, *Cap High Low* is a capability that states that the current program counter is *Low* and the program is authorized to modify a location with the label *High*. A program can trade in a program counter value *PC l* for a capability *Cap (lub l m) l* using the *pc2cap* function. Notice that *lub l m* is always greater than *l*, ensuring that the only capabilities that can be generated at a program point are to modify memory at least as high as the current program counter.

$\boxed{\Gamma = S_{\text{Flow}}, hloc\text{:}LabeledRef\ Bool\ High, lloc\text{:}LabeledRef\ Bool\ Low}$

hloc := true; *lloc* := false	let *pc* = *initpc Low* in let *hcap* = *pc2cap Low High pc* in let *pc1*,() = *update High Low hcap hloc True* in let *lcap* = *pc2cap Low Low pc1* in *update Low Low lcap lloc False*

$\boxed{Boxed\ Low\ Unit}$
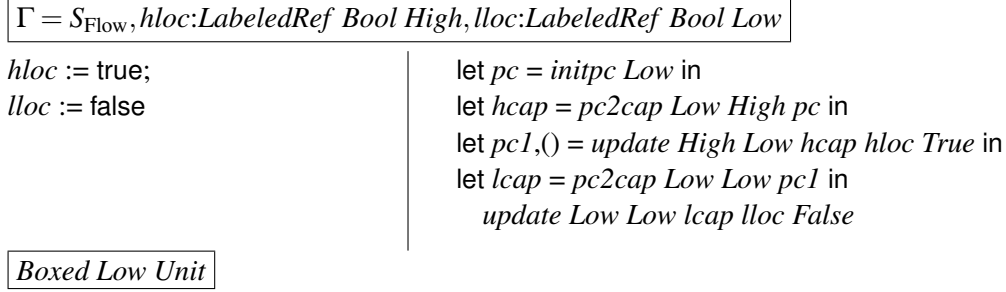
Figure 4.6: Translating a simple Core-ML program to FLAIR

Importantly, the *Cap m l* type is also affine, ensuring that a program cannot duplicate capabilities and use them when they are no longer consistent with the program counter. In order to retrieve a program counter from a capability, the program can call the function *cap2pc*.

The *update* base term corresponds refines the function of the same name in S_{Flow}^0 to account for the affine tokens and capabilities. As before, the arguments *x* and *y* represent the reference and the value to be stored therein, respectively. However, instead of preventing indirect flows by requiring an argument of type *PC l*, the first three arguments of *update* show a capability *Cap l m*. This capability proves that the label *l* of the reference is not less than the confidentiality *m* of the current program counter. Since the program counter token and capability *cap* are affine, *update* must return a token to the caller to allow it to make subsequent calls to the policy. So, the return type of *update* includes a value of type *PC m* (packaged as a *Boxed m Unit*). One remaining point to note about the type of *update*: as with our translation of AIR release rules in Chapter 3, since the argument *cap* is affine, we require every function type to the right of *cap* to also be affine, since they represent closures that capture an affine value.

Figure 4.6 contains a simple example program that illustrates how program coun-

ters, capabilities, and the *update* policy term interact. The top of the figure shows the environment Γ that records the types of the *hloc* and *lloc* as *High* and *Low* locations, respectively. Next, we show a Core-ML program (at left) and its corresponding FLAIR program (at right). In FLAIR, we obtain the initial program counter token of type *PC Low* by calling the *initpc* function. Since this function is affine, it can never be called again in the rest of the program. In order to update *hloc*, we must construct a capability of type *Cap High m* (for some *m*). So, we call *pc2cap* to produce a value *hcap* of type *Cap High Low* from the initial program counter. We then pass *hcap* to the *update* function, along with the reference *hloc* and the value to be stored. The *update* function updates the location *hloc*, consumes the capability *hloc*, and returns a program counter value *pc1* of type *PC Low* back to the caller, along with the unit value that results from the assignment. Finally, in order to update *lloc*, we must present a capability of type *Cap Low Low* to *update*. We construct such a value by applying *pc2cap* to *pc1* and then passing the result to *update* as before. The type of the entire program is shown in the box at the bottom, i.e., a pair consisting of a *PC Low* token and a *Unit* value.

Returning to the signature S_{Flow} of Figure 4.5, we have the term *deref*, which mediates access to the dereferencing operation. A location can be dereferenced at any point in the program, irrespective of the program counter. However, we must be careful to ensure that we do not allow the program to read out of a secret (*High*) location and write the contents to a public (*Low*) location. The *deref* function ensures this by labeling the value read out of the location with the same label as the location itself. The result is that the value is at least as secret as the location in which it was stored.

We turn now to the *branch* function, which corresponds to the (ML-IF) rule in the

Core-ML semantics. As in S_{Flow}^0, the arguments show a program counter token at level l, the boolean guard labeled m, and the "thunkified" branches t and f. The branches receive as arguments program counter tokens at level *lub l m*. Since the tokens are now affine, the types guarantee that the branches only use this token of type *PC lub l m* in their bodies (and not some other stale token in scope, like *initpc*). As with all other functions, the branches thread the tokens back to their callers along with the value of type α that they compute. Finally, as before, the return type of *branch* labels the result of type α with the label m of the guard. In addition, *branch* includes the program counter token in the boxed type that it returns.

The *apply* function corresponds to the type rule (ML-APP) in Core-ML and is similar in structure to *branch*. It allows a labeled function f to be applied to an argument x. The type of f ensures that its body executes in a context where the program counter is labeled with the confidentiality of f itself (which corresponds to the final premise of (ML-APP), $pc \sqcup \mathbf{l} \sqsubseteq pc'$). Additionally, the return type of *apply* ensures that the value returned from the function is also as secret as the function itself (corresponding to the third premise of (ML-APP)).

4.4.2 Simple Examples using S_{Flow}

In this section, we revisit the example programs of Figure 4.4 and show how they can be checked in FLAIR using S_{Flow}. The top-most part of Figure 4.7 shows a secure Core-ML program that updates a *High*-location based on a *High* guard. In the FLAIR program at the right, as before, we have a call to the *branch* function, passing in the

128

Γ = S_{Flow}, *h:Labeled Bool High*, *l:Labeled Bool Low*,
hloc:LabeledRef (ref *Bool*) *High*, *lloc:*(*LabeledRef* (ref *Bool*) *Low*),

if (*h*) then *hloc* := true
else *hloc* := false
	let *tbranch* (*x:Boxed High Unit*) =
	   let *hcap = pc2cap High High* (*fst x*) in
	     update High High hcap hloc True
	
	let *fbranch* (*x:Boxed High Unit*) =
	   let *hcap = pc2cap High High* (*fst x*) in
	     update High High hcap hloc False
	
	branch Low (*initpc Low*) *High h tbranch fbranch*

if (*h*) then *lloc* := true
else *lloc* := false
	let *tbranch* (*x:Boxed High Unit*) =
	   let *cap = pc2cap High Low* (*fst x*) in
	     update Low High cap lloc True
	     #require cap:Cap Low Low
	
	let *fbranch* (*x:Boxed Low Unit*) =
	   let *lcap = pc2cap Low Low* (*fst x*) in
	     update Low Low lcap lloc False
	
	branch Low (*initpc Low*) *High secret tbranch fbranch*
	   #1st arg of fbranch must be Boxed High Unit

if (*l*) then *lloc* := true
else *hloc* := false
	let *tbranch* (*x:Boxed Low Unit*) =
	   let *cap = pc2cap Low Low* (*fst x*) in
	     update Low Low cap lloc True
	
	let *fbranch* (*x:Boxed Low Unit*) =
	   let *hcap = pc2cap Low High* (*fst x*) in
	     update High Low hcap hloc False
	
	branch Low (*initpc Low*) *Low l tbranch fbranch*

Figure 4.7: Tracking effects using S_{Flow}

guard *h* and the branches. However, this time we call the *initpc* function to construct

an initial program counter—since *initpc* is affine in S_{Flow}, it cannot be called elsewhere

in the program. In *tbranch* (and in *fbranch*) we receive a token of type *PC High* as an

argument (reflecting the dependence on the guard *h*). Before updating *hloc*, we construct

a capability *hcap* of type *Cap High High* by calling *pc2cap*. We then pass this capability to *update* along with *hloc* and the value to be stored. Both branches return values of type *Boxed High Unit*, and the *branch* itself returns *Boxed Low Unit*.

In the middle part of Figure 4.7 we have a Core-ML program that is insecure (and untypable) because it has an indirect flow from *High* to *Low*. We might try to write a similar program in FLAIR—the right of the figure shows one such attempt with two typing errors. For instance, if we were to were to try to *update* the location *lloc* in the body of *tbranch*, we must pass in evidence that the program counter is not more secret than the contents of *lloc*. We try to construct such a capability by calling *pc2cap High Low*, but we get back a value of type *Cap (lub High Low) High*, which is equivalent to *Cap High High*. Thus the type checker rejects the call to *update* as incorrect, since in order to update *lloc*, *update* requires a *Cap Low Low* capability. In *fbranch*, the argument *pc* is given a type that allows the body of the function to type check. However, *fbranch* cannot be passed to *branch* as an argument, because the type of *branch* dictates that the first argument of both branches include program counters that are at least as secret as the guard *secret*—in this case, at least *PC High*.

The final program in Figure 4.7 shows how capabilities can be used to modify all locations more secret than the current program counter. At the left, in the else-branch, we update *hloc* in a context that is dependent on *l.* At the right, *fbranch* receives a token of type *PC Low* as an argument. To update *hloc*, we can construct a capability of type *Cap High Low* and pass this to *update*. In contrast, when using S^0_{Flow} in Figure 4.4, we could only update *Low* locations in contexts that were dependent on *Low*-security values.

4.4.3 Examples with Higher-order Programs

We now turn to some examples of higher-order functions and show how they can be checked in FLAIR using S_{Flow}. The top-most part of Figure 4.8 shows a Core-ML program to the left, where, instead of modifying a location in each branch depending on the value of a boolean, we construct closures that modify the locations when they are applied. In FLAIR, the top level is as in Figure 4.7—we call *branch* passing in the true and false branches, *tbranch* and *fbranch* respectively. Each branch returns a pair where the first component is just the program counter token received in the argument and the second component is the closure. In each case, the closure itself takes an argument that includes a program counter token of type *PC High*, indicating statically that this function's effects are only to the *High* fragment of memory. In the body of the closure, we project out the program counter token from the argument *y*, generate a capability *hcap*, and call the *update* function. We call the whole program on the right *p2* and can give it the type:

$$\textit{Boxed Low } (\textit{Labeled } (\textit{Boxed High Unit} \rightarrow \textit{Boxed High Unit}) \textit{ Low})$$

That is, a pair consisting of a *Low* program counter, and a *Low*-security function from *Unit* to *Unit* which is guaranteed to only have an effect (if at all) on the *High* fragment of memory.

The next part of Figure 4.8 shows how the function *p2* can be applied. Since this is a boxed value, we first project out each component—the program counter token *pc* and the labeled function *f*. We then call the *apply* function, passing in the program counter, the function *f* and the argument (). However, our construction of *p2* requires that the function be called with a program counter that has the type *PC High*. But, at the call

$\Gamma = S_{\text{Flow}}, l{:}Labeled\ Bool\ Low, hloc{:}LabeledRef\ Bool\ High$

let *p2* =
  if (*l*) then $\lambda x.\ hloc$:= true
  else $\lambda x.\ hloc$:= false
	let *tbranch* (*x*:*Boxed Low Unit*) =
	    (*fst x*, λy:*Boxed High Unit*.
	        let *hcap* = *pc2cap High High* (*fst y*) in
	          update High High hcap hloc True)
	
	let *fbranch* (*x*:*Boxed Low Unit*) = ...
	
	branch Low (*initpc Low*) *Low l tbranch fbranch*

$\Gamma = \ldots, p2{:}Boxed\ Low\ (Labeled\ (Boxed\ High\ Unit \to Boxed\ High\ Unit)\ Low)$

let *p3* =
  p2 ()
	let *pc, f* = *p2* in
	    apply Low pc High (*sub Low High f*) ()

$\Gamma = \ldots, p3{:}Boxed\ Low\ (Labeled\ Unit\ High)$

let *p2* =
  if (*l*) then
    $\lambda x.\ hloc$:= true; true
  else
    $\lambda x.\ hloc$:= false; false in
  p2 ()
	let *tbranch* (*x*:*Boxed Low Unit*) =
	    (*fst x*, λy:*Boxed Low Unit*.
	        let *hcap* = *pc2cap Low High* (*fst y*) in
	        let *pc, ()* = *update High High hcap hloc True* in
	          (*pc*, true)
	
	let *fbranch* (*x*:*Boxed Low Unit*) = ..
	
	let *pc* = *initpc Low* in
	let *pc1, f* = *branch Low pc Low l tbranch fbranch* in
	    apply Low Low pc f ()

$\Gamma = \ldots, p4{:}Boxed\ Low\ (Labeled\ Bool\ Low)$

Figure 4.8: Higher-order programs that contain secure indirect flows

site, the program counter *pc* has type *PC Low* and the function *f* is labeled *Low*. In this context, the type of *apply* requires *f* to be a function from *Boxed* (*lub Low Low*) $\alpha \to$ *Boxed* (*lub Low Low*) β, whereas, our function *f* has the underlying type

$$Boxed\ High\ Unit \to Boxed\ High\ Unit,$$

and so cannot be passed as is to *apply*.

One way allow this application to proceed is to use subtyping to coerce the outer-

most label of *f* from *Low* to *High*—which is what that call to *sub Low High f* achieves. Now, the type that *apply* requires for the underlying function matches the type we have for *f*—the arguments of both are *Boxed* (*lub Low High*) α. This approach illustrates a way in which subsumption can be used. But, this approach has the unfortunate consequence that the returned value is also labeled *High* confidentiality (since it must be as confidential as the function itself). In this case, since the returned value is just (), the fact that it is *High* confidentiality is insignificant. However, if the value is, say, a boolean, spuriously treating the result is *High* security is undesirable.

The final part of Figure 4.8 shows an alternative translation that fixes this problem. Here the closure in *tbranch* is a function that takes a *PC Low* token as an argument, which statically only guarantees that its memory effects are to the *Low* (or higher) fragment of memory. In the body of the function, we use *pc2cap* to generate a capability to modify the *High* security location *hloc* and then package the boolean to be returned along with return the program counter of type *PC Low*. This time, the function *p2* has the type

$$Boxed\ Low\ (Labeled\ (Boxed\ Low\ Unit \rightarrow Boxed\ Low\ Bool)\ Low)$$

In order to call this function, we just have to unbox it and pass the components to the *apply* function. The resulting value has the type *Boxed Low* (*Labeled Bool Low*), as desired.

We turn next to the programs in Figure 4.9 which display insecure indirect flows. In the first section of the figure, we have a program fragment that type checks in both Core-ML and in FLAIR. This is a program that based on a secret value *h*, constructs a closure that, *only when applied* leaks the value of *h* into the public location *lloc*. This program *p2* has the type shown in the box, reproduced below:

$$\boxed{\Gamma = h\text{:}Labeled\ Bool\ High, lloc\text{:}LabeledRef\ Bool\ Low}$$

let *p2* =
   if (*h*) then λ*x. lloc* := true
   else λ*x. lloc* := false

let *tbranch* (*x:Boxed High Unit*) =
    (*fst x*, λ*y:Boxed Low Unit.*
       let *lcap = pc2cap Low Low* (*fst y*) in
         update Low Low lcap lloc True

let *fbranch* (*x:Boxed High Unit*) =
    (*fst x*, λ*y:Boxed Low Unit.*
       let *lcap = pc2cap Low Low* (*fst y*) in
         update Low Low lcap lloc False

branch Low (*initpc Low*) *High h tbranch fbranch*

$$\boxed{\Gamma = \ldots, p2\text{:}Boxed\ Low\ (Labeled\ (Boxed\ Low\ Unit \rightarrow Boxed\ Low\ Unit)\ High)}$$

p2 ()

let *pc, f = p2* in
  apply Low pc High f ()
  #f's argument must be (*Boxed High Unit*)

Figure 4.9: Higher-order programs with insecure indirect flows

p2:Boxed Low (*Labeled* (*Boxed Low Unit* → *Boxed Low Unit*) *High*)

This is the type of a boxed function, where importantly, the function itself is labeled *High*. This reflects that fact that the value of *p2* depends on a *High* security value *h*. If we try to apply this function (in the final section of the figure), we find that the application fails to type check. The reason is that *apply* requires the first argument of the function *f* to include a program counter token that proves that *f*'s effects are only to locations that are at least as secure as *f* itself. In this case, *f* is *High* security, but the program counter argument of *f* is *PC Low*.

4.4.4 Security Theorem

The main security result of this chapter is a proof that FLAIR programs that are type correct with respect to S_{Flow}, the signature of Figure 4.5, enjoy a noninterference property. However, before we can proceed, we must define a model for the base terms in S_{Flow}. This model axiomatizes the reductions of base-term applications by associating a set of equations with each base term. (In FLAIR, each equation is optionally parameterized by a store Σ.) For instance, the desired semantics of the *lub* function is defined by the following set of equations E_{lub}, where the equation $v_1, v_2 \rightsquigarrow e_3$ axiomatizes the reduction of application *lub* v_1 v_2 to the expression e_3.

$$
\begin{array}{rcccl}
E_{lub}: & Low, & Low & \rightsquigarrow & Low \\
& Low, & High & \rightsquigarrow & High \\
& High, & Low & \rightsquigarrow & High \\
& High, & High & \rightsquigarrow & High
\end{array}
$$

Our security theorem will be parameterized by a model for FLAIR programs M, where $M(lub) = E_{lub}$, i.e., all applications of *lub* are defined by the above set of equations. Our proof in Appendix C provides a complete model for S_{Flow}. However, for all the base terms other than *lub*, the types in FLAIR are precise enough that any set of equations that are consistent with the types in the signature are sufficient for noninterference.

To illustrate the sufficiency of type-consistency, we point out first that several of the function-typed base terms in S_{Flow} are just type coercions—operationally, these coercions are the identity function (on one of its arguments). For example, the *join* function has the

135

following type:

$$\forall \alpha::\mathrm{U}.(l{:}Lab) \rightarrow (m{:}Lab) \rightarrow (x{:}Labeled\ (Labeled\ \alpha\ l)\ m) \rightarrow Labeled\ \alpha\ (lub\ l\ m)$$

The only possible implementation of *join* that respects this type is the identity function on the argument *x*. The same is true of the other coercions like *sub*.

The insight that these coercions must be identity functions on one of their arguments is a particular instance of a parametricity theorem [136]. A reading of the types in S_{Flow} with parametricity in mind indicates that a similar theorem applies to most of the base terms. In the case of *deref*, any set of type consistent equations must simply read a value out of the location received as an argument and return the result. The *apply* base term also has only one possible type-correct implementation—it must apply *f* to *x* and return the result.

In the case of *branch* and *update*, the types in the signature admit more than one possible definition. For *branch*, while parametricity guarantees that an implementation must apply one of the branches, our types are not precise enough to guarantee that the branch executed correctly reflects the value of the guard *b*. For *update*, since the returned type is *Unit*, one possible type correct implementation is to simply return (). However, any implementation that mutates the store must do so as intended. That is, it must assign the value *y* to the location *x* (and not to any other). Clearly, the choice we make for defining the operational behavior *branch* and *update* has a profound impact on the semantics of the FLAIR program that uses these terms. However, purely from a security perspective, the specific implementation that is chosen does not matter. For instance, an ill-chosen

(but type-correct) definition of *branch* may cause the else-branch to be executed instead of the then-branch; but, the types guarantee that even in this case, no high-confidentiality information is leaked to low-confidentiality outputs.

While we do not formalize these parametricity arguments, an interesting direction of future work would be to investigate the validation of a policy specification (in the form of a signature) with respect to the theorems that can be deduced from the types in the specification. Rather than prove a security result (as we did in FABLE) by considering specific implementations (in the source language) of a policy, reasoning with parametricity at the meta-level may lead to simpler and more abstract proofs.

Definition 4 (Low-equivalence of stores). *Two stores Σ_1 and Σ_2 are low-equivalent with respect to an environment Γ if and only $dom(\Sigma_1) = dom(\Sigma_2)$ and $\forall \ell.\Sigma_1(\ell) \neq \Sigma_2(\ell) \Rightarrow \Gamma(\ell) = LabeledRef\, t\, High$*

Theorem (Noninterference for FLAIR, with S_{Flow}). *Suppose, for well-formed Γ, the signature S_{Flow}, a model M type-consistent with S_{Flow}, such that $M(lub) = E_{lub}$, we have $\Gamma; initpc \vdash_{\text{term}} e : t; \cdot$. Then, for any two low-equivalent stores Σ and Σ', such that $\Gamma \models \Sigma$ and $\Gamma \models \Sigma'$, if we have*

$$M \vdash (\Sigma, e) \longrightarrow (\Sigma_1, e_1) \longrightarrow \ldots \longrightarrow (\Sigma_n, e_n)$$
$$M \vdash (\Sigma', e) \longrightarrow (\Sigma'_1, e'_1) \longrightarrow \ldots \longrightarrow (\Sigma'_m, e'_m)$$

Then, the sequences $\Sigma, \Sigma_1, \ldots \Sigma_n$ and $\Sigma', \Sigma'_1, \ldots, \Sigma'_m$ are low-equivalent up to stuttering.

This timing- and termination-insensitive noninterference property is similar to an analogous property for Core-ML programs. Our proof is based on a technique due to

Pottier and Simonet that allows two program executions to be embedded in the syntax of an extended calculus. Since S_{Flow} embeds Core-ML in FLAIR, the terms structure of a FLAIR program essentially mirrors the structure of a Core-ML typing derivation, e.g., each application of the *sub* function in FLAIR corresponds to an application of a subtyping judgment in the Core-ML derivation. However, FLAIR programs can make use of (first-class) polymorphism, but the Core-ML subset that we have defined is strictly monomorphic. Rather than extend Core-ML with polymorphism, we simply assume that all the polymorphism in FLAIR is removed via code replication. The blow-up in code size is quadratic—for n FLAIR functions at m call sites, we can produce $n \cdot m$ Core-ML function definitions.

4.5   Concluding Remarks

This chapter concludes a development in which we have shown how a general purpose type system, as embodied by FLAIR, has an expressive power with regard to security policy enforcement that, to our knowledge, is matched by no other single programming formalism. In this chapter, we have demonstrated how an information flow policy can be enforced with purely static controls for programs that manipulate mutable references to memory. Although we have focused on memory effects, our encoding of information flow in S_{Flow} can easily be generalized to account for other kinds of side effects, e.g., sending messages over the network, or printing output to terminal.

We have focused so far on the expressive power of our type-based approach. We have repeatedly dismissed concerns of usability by positioning FLAIR as the kernel of

an intermediate representation rather than a source-level language for use by a human programmer—particularly for the more complex policies that we have explored. In subsequent chapters make the claim that for many simple policies of interest, the basic idea of a customizable security label model that is interpreted by a user-defined enforcement policy is in fact practical for real-world programs. The main evidence for this claim: SELINKS, a new programming language for building secure web applications.

5. Enhancing LINKS with Security Typing

Multi-tier web applications are becoming the *de facto* standard for programs that need to share sensitive information across a wide community of users. To recapitulate the discussion from the Chapter 1, we would like verify that such applications correctly enforce fine-grained security policies. For a program like Intellipedia for instance, which makes classified documents available to the U.S. intelligence community using a Wikipedia-like interface, we would like to protect fragments of documents with access control and provenance tracking policies. On-line stores, web portals, e-voting systems, and online medical record databases have similar needs. This chapter and the next set out to show that by applying FABLE to the design of a new programming language, user-defined security policies can be reliably and efficiently enforced in multi-tier web applications.

There are two main approaches to enforcing fine-grained policies in a multi-tier web application. A *database-centric* approach relies on native security support provided by the DBMS. For example, Oracle 10g [97] supports a simple form of *row-level* security in which security labels can be stored with individual rows, and the security semantics of these labels is enforced by the DBMS during database accesses. A similar approach is possible with views backed by user-defined functions [97]. A customized row-level security label is hidden by the view, and the label's semantics is transparently enforced

by the DBMS via invocations to user-defined functions as part of query processing.

Alternatively, a *server-centric* approach is to enforce application-specific policies in the server. For our example, the programmer could define a custom format for access-control labels, store these with rows as above, and then perform access control checks explicitly in the server prior to security-sensitive operations. This is the basic approach taken by J2EE [56] and other application frameworks.

Neither approach is ideal. The database-centric approach is attractive because highly-optimized policy enforcement code is written once for the database for all applications, rather than once per application, improving efficiency and trustworthiness. On the other hand, DBMS support tends to be coarse-grained and/or too specialized. For example, most DBMSs provide only simple access control policies at the table level, and Oracle's relatively sophisticated per-row labels only apply to totally-ordered multi-level security policies [42]. Even customized support based on views, or further native security extensions, will only go so far: some policies simply cannot be enforced entirely within the database. For example, an end-to-end information flow policy [111] requires tracking data flows through the server to ensure, for instance, that the server does not write confidential data to a publicly-viewable web server log.

The server-centric approach has the opposite characteristics: it can enforce highly-expressive application-specific policies, but is potentially far less efficient and less trustworthy. In the worst case the server must load entire database tables into server memory to access and interpret the custom security labels associated with each row. And because the application performs security checks explicitly, programming errors can create security vulnerabilities.

As a remedy to this state of affairs, this chapter proposes an extension to the LINKS web-programming language [35] that can be used to build secure, multi-tier applications by combining the best features of the server-centric and database-centric enforcement strategies. Our extension is called *Security-Enhanced* LINKS, or SELINKS, and employs a server-centric programming model for maximum policy expressiveness, and uses compilation and verification techniques to make performance and trustworthiness competitive with the database-centric approach.

We have used SELINKS to implement two applications that enforce interesting security policies. However, this chapter focuses on the features of the SELINKS language. Chapter 6 discusses our example applications as well as some aspects of the implementation of SELINKS in detail.

5.1   Overview

We begin in Section 5.2 by illustrating several of the features of LINKS via a simple multi-tier example program. We then consider the security issues that arise for such multi-tier programs and indicate how these issues might be addressed through the use of label-based security policies.

Our extensions to LINKS consist of two main components. The first is an implementation of a FABLE-like type system for LINKS, which can be used to verify that application programs correctly enforce their policies. The second component of SELINKS is a novel compilation procedure that aims to make the enforcement of policies in database code more efficient. This chapter focuses on a description of the main feature of the SELINKS

type system. The next chapter motivates and describes our cross-tier policy compilation strategy.

Policy enforcement in SELINKS works much as it does in FABLE. The SELINKS programmer specifies a policy by associating customizable security labels with sensitive data in the program. The usage modes of labeled data are defined via specially privileged enforcement policy functions. The type checker ensures that application programs include the appropriate calls to the enforcement policy to ensure that all usages of sensitive data is mediated by the policy. Section 5.3 sketches the main elements of how this works.

Although the basic concepts of FABLE translate directly to SELINKS, programming with label-based security policies at the source level requires several additional constructs. Many of these constructs are standard extensions, but their inclusion in SE-LINKS required addressing some subtle details. For example, we include built-in support for dependently typed records, rather then encoding them with higher-order functions, as we did in Chapter 2. However, adding this constructs to LINKS required adapting techniques from the theory of existential types to manage names bound within the scope of a record. Other constructs involve small theoretical advances. Notable among these is our use of of *phantom variable polymorphism*. When coupled with inference, we have found this feature to significantly reduce the annotation burden of programming with dependent types.

Sections 5.4, 5.5 and 5.6 catalog each of our SELINKS-specific constructs in detail. Where the theory is novel, as with phantom variable polymorphism, we sketch formal definitions of the semantics. However, for the most part, we rely on informal descriptions of the implementation of these features in the current version of the SELINKS compiler.

143

As such, one purpose of this chapter is to serve as a reference manual for the intrepid programmer intent on experimenting with the research prototype that is SELINKS.

We should note that the current version of SELINKS does not support the enforcement of policies in the style of λAIR or FLAIR. As we have already observed, we expect working with FLAIR's combination of affine and dependent types at the source level to require too much effort on the part of the programmer. In Chapter 8 we suggest directions for future work that aim to address this limitation.

5.2 An Introduction to LINKS

Modern web applications are typically designed around a three-tier architecture. The part of the application related to the user interface runs in a client's web browser. The bulk of the application logic typically runs on a web server. The server, in turn, interacts with a relational database that serves as a high-efficiency persistent store. Oftentimes, this architecture is generalized to n-tiers. For instance, one might split the web server into a tier that processes HTTP requests and handles the presentation logic, and a so-called *application server* that runs the core application logic. Multiple web and application servers are also possible, for better load distribution.

Programming such an application can be challenging for a number of reasons. First, the programmer typically must be proficient in a number of different languages—for example, client code may be written as JavaScript; server code in a language like Java, C#, or PHP; and data-access code in SQL. Furthermore, the interfaces between the tiers are cumbersome—the data submitted by the client tier (via AJAX [54], or from an HTML
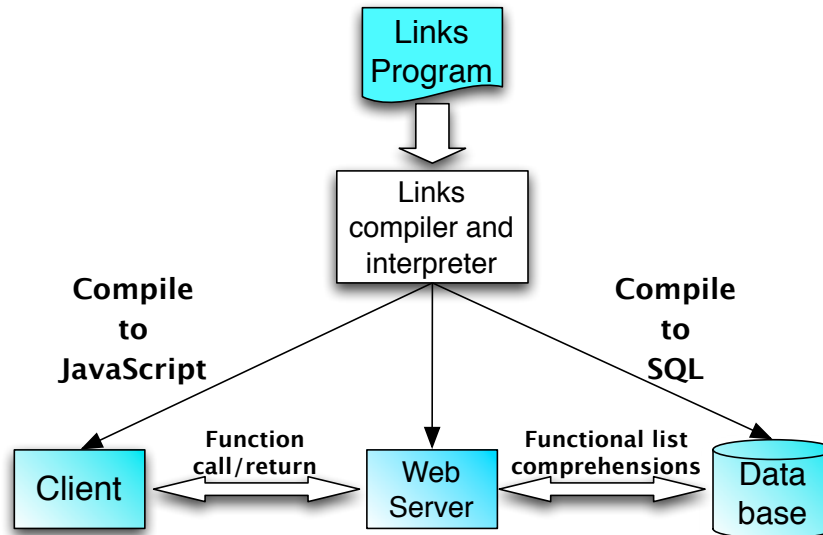
Figure 5.1: An overview of the execution model of LINKS

form) is not always in a form most suitable for processing at the server or database. These factors are elements of the so-called *impedance mismatch* in web programming.

LINKS aims to reduce this impedance mismatch by making it easier to synchronize the interaction between the tiers of a web application. LINKS is a strict, typed, mostly-functional language, with syntax resembling that of JavaScript and employing ideas from other languages, including XQuery [146], Erlang [44], Kleisli [142], Scheme [71], and others. The principal novelty of LINKS is that it brings together a diverse set of proposals in single language in a manner that enables a unique execution model. Rather than construct the multiple tiers of a web application in separate languages (and glue them together via non-standard interfaces), a LINKS programmer writes a *single* program that expresses the entirety of a multi-tier web application, from client to server to database. Figure 5.1 illustrates the execution model of a LINKS program graphically.

A LINKS program consists of a series of function definitions followed by some ini-

tialization code to start the application. Each function is annotated with qualifiers, either **client** or **server**, to indicate where it is supposed to run. LINKS provides a code generator that translates client-side functions to JavaScript to run in the browser; additionally, an interpreter runs server functions at the web server. Function calls may traverse the client/server gap—LINKS automatically translates such calls into synchronous remote procedure calls (RPCs) using AJAX [54]. LINKS also allows data access code to be integrated with server-side functions by representing database operations as list comprehensions in the style of Kleisli [142] and LINQ [83]. The server-side interpreter translates list comprehensions to SQL expressions and dispatches these to be run at the database. Thus programs are expressed at a fairly high-level while the low-level details are handled transparently by the compiler.

The original LINKS paper [35] provides a comprehensive discussion of the various features of the language. Here, we just attempt to provide the reader with a feel for LINKS programming, with an eye towards the issues that arise when attempting to enforce fine-grained custom security policies.

5.2.1 Programming in LINKS

Figure 5.2 shows a simple, but fairly typical, LINKS program. At a high level, this program provides a web-based interface to a database of employee records. The database contains a table that associates an employee's name with her salary. The program allows the user to enter a minimum salary and the program selects all records in the database for which the salary exceeds the minimum and renders the result in the browser as HTML.

```
 1   var employeeTab = table ''Employee'' with
 2                          (name : String, salary : Int)
 3                          from (database ''EmpDB'');
 4
 5   fun getRecords(minSalary) server {
 6     for (var row ← employeeTab)
 7       where (row.salary > minSalary)
 8       [row]
 9   }
10
11   fun showRecords(minSalary) client {
12     var recs = getRecords(minSalary);
13     var tableBody = for (var r ← recs)
14       <tr>
15        <td>{stringToXml(r.name)}</td>
16        <td>{intToXml(r.salary)}</td>
17       </tr>;
18     <html>
19      <body>
20       <table>{tableBody}</table>
21      </body>
22     <html>
23   }
24
25   fun main() client {
26     <html>
27      <body>
28       <form method=''POST''
29             l:action=''{showRecords(minSalary)}''>
30        Enter minimum salary:
31         <input type=''text'' l:name=''minSalary''/>
32         <input type=''submit'' value=''Get records!''/>
33        </form>
34      </body>
35     </html>
36   }
37
38   main()
```

Figure 5.2: A LINKS program that renders the contents of an employee database in a web browser

147

The program begins by defining a schema for the database table that stores the employee records (lines 1-3). The remainder of the programmer shows the functions getRecords, showRecords, and main. Notice that each of these are annotated with a location qualifier (**client** or **server**), indicating on which tier they are intended to run. Finally, (line 38) we have a call to the main function—this is code that will be run on the client in order to start the program.

The database table in this case is called "Employee" and is defined as a relation in the database called "EmpDB." Each row in this table has two fields (columns). The first, name, stores the employee's name as a *String*, and the salary field is an *Int* (the type of integers). A handle to this table is bound to the variable employeeTab which is in scope for the remainder of the programmer. All operations on this table (such as querying or updating) will be performed using this handle.

LINKS does not currently allow database operations (like queries) to be performed directly from client code. Instead, an interface to these tables are exposed to client functions by server functions that encapsulate the application logic. In this case, we have a single server-side function getRecords that allows the Employee table to be queried for all records where the salary exceeds the argument minSalary.

The LINKS view of a database table is simply a list of records. Under this model, a database query is a *list comprehension* [135]. The body of getRecords is a single list comprehension that selects data from the Employee table. In particular, for each row in the table (the syntax **for**(**var** row ← employeeTab)) for which the **where** clause is true, the row is included in the final list to which the comprehension evaluates (the syntax [row]). Since the LINKS view of each row is a record, the **where**-clause projects out the salary

148

field and checks if it is greater than the argument minSalary. The list computed by this comprehension is returned by the function. (LINKS functions simply return the value computed by their last expression—as in most functional languages, there is no explicit return keyword.)

A database list comprehension is checked against the type signature provided as the table schema. In this case, both *String* and *Int* are primitive types in LINKS. Therefore, comparing the *Int*-typed salary field against a *String* constant in the **where**-clause of the query would be flagged by LINKS as a type error. Additionally, as far as the programmer is concerned, the types given to the columns of the table are independent of the underlying representation of these types in the database—the translation between the database representation of these types and the LINKS representation is taken care of by the LINKS runtime. However, if no such translation is possible, the current implementation of LINKS will signal a runtime error. However, it should be straightforward to parameterize the LINKS type checker with a database schema and statically check that the LINKS types given to a table's columns can always be translated to corresponding types in the database.

We now turn to the client functionality, beginning with the main function. This function constructs the initial web page of the application. Its body is an HTML page (LINKS allows XML literals to be embedded within the source) which contains a form to collect the user's input. The two input fields in the form are, first, a text field named minSalary (the name minSalary is in scope throughout the enclosing form element), and a form submission button. When the user enters an integer value in the text field and presses the submit button, the l:action handler specified in the enclosing form element is called. In

this case the handler is a local call to the client function showRecords where the argument is the contents of the text field named minSalary. The LINKS runtime takes care of input validation—in case the user enters a non-integer value in the text field, the runtime will fail to parse the value and refuse to dispatch the function call. (A more graceful failure mode that, say, prompts the user to enter a different value is not yet provided.)

The showRecords function makes a remote call to the server for the function getRecords passing in the user input minSalary as input. There is no distinction at the source level between a local and a remote call. The LINKS runtime, running in the web browser as a JavaScript library, dispatches this call to the server via a synchronous AJAX call. The returned value is a list recs of database rows that matched the query. The name **rec** is bound in the remainder of the function (i.e., the notation **var** x = e1; e2 is LINKS notation for the more familiar **let** x = e1 **in** e2). The function showRecords then iterates through these rows (using the same list comprehension syntax), but this time constructing an HTML representation of the matched rows—each is an HTML table row (the <tr>) element with two columns (the <td> element) containing the name and salary fields coerced to their XML (equivalently, HTML) representations. Finally, showRecords returns a new HTML page that contains a <table> element, where the body of the table is the list of XML rows constructed by the list comprehension and bound to the tableBody variable.

Finally, in order to explain the examples that appear in the rest of this chapter it is important to note that, by default, functions are not curried in LINKS. The type of the showRecords function, for instance, is $(Int) \longrightarrow Xml$, indicating that it is a function that takes a tuple containing a single Int-typed field as an argument and returns an Xml value. Functions that take multiple arguments usually do so by accepting multiple fields in the

argument record. For example, a version of showRecords that took both a minSalary and a maxSalary as arguments is typically defined as **fun** showRecords(min, max) { ... } and is a given the type (*Int*, *Int*) —→*Xml*. It is possible to explicitly define a function as being curried, by using the notation **fun** showRecords (min) (max) { ... }. This function would be given the type (*Int*) —→(*Int*) —→*Xml*, the type of a function that expects a tuple with a single integer as an argument and returning a function that expects a tuple with a single integer which in turn returns some *Xml*.

5.2.2   Fine-grained Security with Links

It is natural to want to enforce application-specific security policies for programs like the example of Figure 5.2. For instance, we might want to limit access to an employee's salary information only to certain principals—for instance, the employee herself, her managers, and maybe certain other privileged actors like members of an organization's human-resources team.

One way to apply such a security policy would be to partition the table into multiple tables where all rows in a given table have identical access control requirements. The database can enforce access protection at the level of the table itself preventing a user from accessing a table when she does not have the right set of privileges. But, for a large organization with a complex managerial hierarchy, such an approach can lead to a proliferation of tables. Managing a large number of tables can easily become unwieldy. Furthermore, the privilege of creating tables and setting access controls is often restricted to users with administrative rights. This makes it difficult for ordinary users to apply

151

```
var employeeTab = table ''Employee'' with
                     (acl : String, name : String, salary : Int)
                     from (database ''EmpDB'');

fun getRecords(credential, minSalary) server {
  for (var row ← employeeTab)
    where (accessAllowed(credential, row.acl) &&
           (row.salary > minSalary))
    [row]
}

fun selectAll() server {
  for (var row ← employeeTab)
    [row]
}
```

Figure 5.3: Enforcing a fine-grained access control policy in LINKS

discretionary controls to their data with table-level protection. Additionally, indexing data in multiple tables can be difficult or impossible, which can degrade the performance of query execution.

An alternative approach is to associate some metadata with each row in the employee table that identifies the set of users that can access the record. Queries of the table can be expected to examine this metadata against the credentials of the user issuing the query and return the result only if the access check succeeds.

For instance, one might define the "Employee" table as shown in Figure 5.3. Each row now contains three columns; name and salary are as before, and the new acl field holds some string metadata that represents an access control list. We can then revise our function getRecords to take two arguments, credential and minSalary. The new argument credential is some token that represents the identity of the user on whose behalf the query is to be executed. The query itself is similar to what we had before, except now, in the where-clause, we include an access control check. This check is a call to the function

accessAllowed, passing in the user's credential and the access control list on the row being examined. We only include the row in the list of results if the access control check succeeds.

Of course, we would like to ensure that LINKS programs are always correct with respect to their security policies. For instance, we would like to ensure that access control checks like accessAllowed are always present at the right places in the program. One definition of correctness might be that the program examines the salary field of a row in the database only after it has performed an access check of the corresponding acl field in the same row. Under this definition, using the getRecords function of Figure 5.2 with the table declaration of Figure 5.3 is deemed insecure, since it does an integer comparison on the salary field (thereby examining it) without checking the acl field of the row.

On the other hand, consider the function selectAll shown at the bottom of Figure 5.3. The query in this function simply selects every row in the Employee table and returns it. On its own, we might consider this program to be secure since it certainly does not inspect the salary field of any row in the table. However, clearly the list of rows returned by selectAll contains sensitive data. So, we would also like to ensure that such sensitive data does not flow to a location where it can be inspected by an unprivileged user.

These examples illustrate the two main concerns that our extensions to LINKS must address.

- First, we aim to ensure *complete mediation* of the security policy. By augmenting the type language of LINKS with security labels in the style of FABLE, and modeling functions like accessAllowed as FABLE enforcement policy functions, we can check

153

that the appropriate policy checks are always present in an SELINKS program.

- Second, we seek to ensure that all cross-tier data flows in the program are consistent with the level of trust we have in those tiers. In particular, our trust model considers code that runs in the client tier to be untrusted, since we cannot easily assure that the client runs code sent by the LINKS compiler to the web browser. In the context of the example of Figure 5.3, this trust model means that the list of rows returned by selectAll are not allowed to flow directly to a client function—a policy check must intervene to authorize the release of this data to the client.

Additionally, we would like to ensure that database queries that contain calls to potentially complex enforcement policy functions (like accessAllowed) can still be executed efficiently within the database. We defer addressing this concern to Chapter 6, where we show how enforcement policy functions and database list comprehensions can be compiled for good performance.

5.3   SELINKS Basics: Enforcing Policies with Static Security Labels

We begin our presentation of SELINKS by considering how to enforce particularly simple security policies. For pedagogical reasons, we will begin with simple policies specified using static security labels. Subsequent sections will illustrate how to specify and enforce policies using dynamic labels in SELINKS.

Whether static or dynamic, specifying and enforcing a security policy in SELINKS typically proceeds in three steps.

- First, the policy designer chooses a language of security labels. For example, for the simplest form of information flow policy, we might use the labels Low and High.

- Next, we identify the sensitive resources in our program and label their types with security labels that protect them from unrestricted usage by the application program. For instance, we might give sensitive values in the program, such as passwords, types such as *String*{High}, indicating that these will be treated as High confidentiality. Additionally, library functions that are significant from a security perspective are also given types to reflect their intended usage. For instance, a library function print that prints strings to a user's terminal might be given a type such as (*String*{Low}) —→(), indicating that only Low-security strings can be printed to the terminal.

- Finally, we write *enforcement policy* functions that give an interpretation to the security labels. Without the enforcement policy, the labels that decorate types are entirely uninterpreted in the program. There is no way, for instance, to allow a Low-security integer to be treated as a High-security one. As in FABLE, the enforcement policy is granted special privileges to interpret label types by defining the conditions under which labeled data can be used, or how type of labeled data can be coerced from one type to another.

Under the assumption that the enforcement policy is correct, and given that we have assigned proper types to protected data and sensitive library functions, the SELINKS type checker can be used to ensure that an application program meets a set of high-level security goals. As with FABLE, the type system ensures that an application program always

relies on the enforcement policy to construct and destruct protected data. Additionally, as we will see in Section 5.5, the SELINKS type system also ensures that protected data is never sent directly to the untrustworthy client tier.

In the remainder of this section, we illustrate each of these three basic steps towards security enforcement in SELINKS.

5.3.1 Defining a Language of Security Labels

Specifying a security policy in SELINKS begins by choosing a language of security labels. In FABLE, we restricted terms in this language to be applications of constructors from an algebraic datatype. While this was adequate in the formal setting, for practical policies, we would like to be able to construct labels that include values other than just the data constructors of an algebraic datatype. For instance, it would be much more convenient to represent an access control list as a list of tuples, where each tuple contains a user's integer UID and the user's name (say, for pretty printing). This would allow us to manipulate access control lists using all the standard list library functions like searching through the list for an element, folding over it etc. For this reason, SELINKS generalizes the language of security labels to include arbitrary data values (with some caveats, discussed shortly). An example label type in SELINKS is shown below.

typename UserRec = (username: *String*, uid: *Int*);
typename *Acl* = *List* (UserRec) **is** *lab*;

This declaration defines a type alias called *Acl*, intended to represent an access control policy. This is an alias for the type of a *List*, where each element of the list is a record with two fields—the first, a *String*-typed field called username and the second an *Int*-typed field

156

called uid. Here, *List* is a type constructor defined in the standard library, and the notation *List*(t) represents the application of this type constructor to the type t.

Notice that this type declaration concludes with an assertion "**is** *lab*". This assertion serves as a type annotation which signals the programmer's intention to use values of the *Acl* type as security labels—we call such types *label types*. One way to the think of the "**is** *lab*" annotation is that it asserts that the type *Acl* is a member of the *lab type class* [137]. In our current implementation, the semantics of this type class is utterly trivial. We permit any type declared with the **is** *lab* assertion to be treated as a member of the *lab* type class—i.e., membership in this type class does not demand any particular constraint of the underlying datatype. However, we expect this to change in the near future to accommodate the two features discussed below. In the meantime, we allow the "**is** *lab*" annotation to be elided for convenience. We expect future versions of SELINKS to be more strict with this requirement, in order to satisfy the following two properties.

1. **Ensuring the purity of type-level expressions.** First, since expressions of label type can appear at the type level, we should to ensure that these expressions are pure—i.e., that they have no side effect. Although LINKS is primarily a functional language (unlike a language like ML, LINKS programs cannot manipulate memory via references), programs can have side effects by altering the database. Our current prototype permits type-level expressions to include database operations although attempting to give a reasonable semantics to such expressions at the type-level appears to be unwise. One enhancement that we anticipate is to enrich the LINKS type system so that we can lock all side-effecting computation within a monad [87]. We

could then ensure that the only members of the *lab* type class are types whose values are computed by purely functional code. Recall that we adopted a similar restriction with FLAIR in Chapter 4, where we tracked memory effects in the type system and forbade effectful expressions from appearing at the type level.

2. **Ensuring the serializability of label values.** Since LINKS targets multi-tier applications, data values are required to be communicated across tiers. For instance, in Section 5.4.2 (and in greater depth in Chapter 6), we will argue that reliable enforcement of security policies requires label-typed values to be stored in the database. With this in mind, we envisage limiting membership in the *lab* type class to types whose values can be readily serialized for storage in the database. This would, for instance, exclude function types since serializing code to the database is unlikely to be efficient.

$$\textbf{typename } \textit{LatticeLab} = [|\ \mathsf{Low}\ |\ \mathsf{Med}\ |\ \mathsf{High}|];$$

$$\textbf{sig } \mathsf{foobar}: (\textit{LatticeLab } \textbf{is } \textit{lab}.\mathsf{High}) \longrightarrow ()$$
$$\textbf{fun } \mathsf{foobar}\ (\mathsf{h})\ \{\ ()\ \}$$

Figure 5.4: An example illustrating the syntax of singleton label types in SELINKS

In addition to the *lab* type, FABLE provides a precise singleton type of labels $lab \sim e$. The latter type is only inhabited by the value to which e evaluates (if one exists). An SELINKS version of this construct is shown in Figure 5.4. The type alias *LatticeLab* that stands for a variant type consisting of three constructors, Low, Med or High. We then define a type for the function named foobar, using the **sig** construct from standard LINKS. The type we give to this function shows that it expects a single argument a value of the type *LatticeLab*,

but the label type assertion "**is** *lab*.High" asserts that not only is the argument to be used as a label, but additionally that it must be the value High. That is, the **is** *lab*.High refines the variant type *LatticeLab* to just the single data constructor High. The type checker ensure that the foobar function is only ever called with the argument High. In this case, the body of foobar is trivial (it just returns the unit value), but if foobar were to perform some security sensitive operation, we would be able to assume that its argument h is High throughout the body of the function. We permit arbitrary label-typed expressions *e* to be used in the *lab*.*e* construction.

5.3.2   Protecting Resources with Labels

Security labels are only useful insofar as they can be used to protect sensitive resources with a policy. This kind of security labeling is a central feature of FABLE, and it translates naturally to SELINKS. The SELINKS type $t\{e\}$ is the type of some data of underlying type t, protected by the security label in the expression e.

sig sock_send : (*Socket*) $\longrightarrow$ (*String*) $\longrightarrow$ ()

sig sock_send_Low : (*Socket*{Low}) $\longrightarrow$ (*String*{Low}) $\longrightarrow$ ()
fun sock_send_Low (sock) (data) { ... }

Figure 5.5: Protecting a socket interface with simple security labels

The code in Figure 5.5 illustrates a particularly simple usage of labeled types in SELINKS. This snippet begins with a type signature for the function sock_send, a curried function that represents a function from an API that allows data to be sent on a network socket. This is a function that takes two arguments, the socket and the string data to be sent on the socket, and returns a unit.

159

In the event that we wish to control what data is sent on which socket, we can protect sockets with security labels indicating the security level of data which they are allowed to carry. An instance of such a protection policy is defined in the types of the next function in the snippet, sock_send_low. In this case, the first argument is a *Socket* value that is protected by the static label Low, indicating that it is only cleared to carry data that is marked as being Low security. The next argument is a *String*, but one that is labeled Low security. These types ensure that the security requirements of the socket interface are observed. An application program cannot call the sock_send function with a protected socket since the types do not match, and must call the sock_send_low function with a socket and data that are both tagged with the label Low.

5.3.3 Interpreting Labels via the Enforcement Policy

Enforcement policy functions in FABLE translate directly to SELINKS in that certain functions can be tagged with the **policy** keyword, indicating that they are privileged. These policy functions then have access to two special built-in operators, unlabel and relabel, that permit them to manipulate labeled data. The type checker ensures that application programs (i.e., code that does not have the privilege conferred by the **policy** keyword) treat labeled data abstractly.

The example program in Figure 5.6, adapted from our previous example, illustrates a usage of enforcement policy functions. As before, the sock_send function is from the socket API and does not pay any particular attention to the security level of sockets or the data that is allowed to be sent on a socket. The new_socket function is also a library

160

```
sig sock_send : (Socket) ⟶ (String) ⟶ ()

sig new_socket : (String) ⟶ Socket{Low}

sig sock_send_Low : (Socket{Low}) ⟶ (String{Low}) ⟶ ()
fun sock_send_Low (sock) (data) policy {
    sock_send (unlabel(sock)) (unlabel(data))
}

sig concat_LH : (String{Low}) ⟶ (String{High}) ⟶ Int{High}
fun concat_LH (l) (h) policy {
    relabel((unlabel(l) ++ unlabel(h)), High)
}
```

Figure 5.6: An enforcement policy to restrict data sent on a socket

function which provides the only way to construct a new socket. Its type ensures that, by

default, new sockets are tagged with the Low label, indicating that they are cleared only to

carry Low-security data. (If we were implementing a lattice-based policy, a complete im-

plementation would presumably also provide some way to also construct sockets labeled

High.)

Since the sock_send function cannot be called directly by an application program

with a new socket, it is forced to use the sock_send_low function. This time, we show how

to implement this as an enforcement policy function. The **policy** keyword that is associated

with the function definition gives the sock_send_low function the privilege to use the unlabel

operation in its body. In this case, it simply unlabels the sock and data arguments (coercing

their types to *Socket* and *String*, respectively) and calls the library function, sock_send.

To illustrate a usage of the relabel operator, the program in Figure 5.6 concludes

with a policy function that defines how labeled strings can be concatenated. The function

concat_LH is specialized to the concatenation of a Low string with a High string, although, as

subsequent examples will show, polymorphism in SELINKS can be used to avoid having

to specialize policy functions in this manner. In the body of the function, we first unlabel each argument before adding them—the type of the ++ operator ensures that we cannot use it with labeled integers. Then, we use the relabel operator to return a value of the *Int*{High}.

Finally, a note about the **policy** keyword: The attentive reader will have noticed from Section 5.2 that LINKS functions are usually tagged with qualifiers (like **client** or **server**) that indicate the tier on which they are to be executed. In SELINKS, the **policy** qualifier is overloaded—all policy functions are pinned to the server.

5.4   Enforcing Policies with Dynamic Labels

In this section, we show how SELINKS can be used to specify an enforce policies specified using dynamic labels [149]. We provide two mechanisms to express dynamic label relationships. First, as in FABLE, SELINKS contains dependently typed functions. Second, SELINKS provides built in support for dependently typed tuples, rather than requiring the programmer to encode them using functions.

5.4.1   Dependently Typed Functions

Figure 5.7 shows an example of dynamic labels using a dependently typed function. The policy function sock_send_dyn is an elaboration of the simpler sock_send_low function from Figure 5.6. The sock_send_low function was specialized to controlling data sent over sockets, where both the data and the sockets were statically known to be labeled as Low. Here, we want to enforce a policy where the labels of the socket and data are represented

by some program value at runtime. Prior to sending the data over the socket, we must check that the label of the data is not more secure than the label of the socket.

We want to give sock_send_dyn a type that captures the labeling relationships among its arguments. In this case, we want to write a type for a function of four arguments, where the first argument l is a label that labels the second argument sock, and where the third argument m labels the fourth argument data. In FABLE, we would write such a type as $(x{:}lab) \rightarrow Sock\{x\} \rightarrow (y{:}lab) \rightarrow String\{y\} \rightarrow \textbf{unit}$. However, parsing conflicts with existing LINKS notation prevents us from reusing the FABLE notation in SELINKS source programs. Instead, we use the notation $\textsf{Pi}x{:}t\longrightarrow t'$. Here, the term variable x is bound to the formal parameter of type t and is in scope all the way to the right of the arrow, in the type t'. That is, this is the SELINKS version of the FABLE type $(x{:}t) \rightarrow t'$.

To understand the type of sock_send_dyn shown on line 3, recall (from Section 5.2) that every argument of a function in LINKS is a tuple, i.e., a record where the field names are "1", "2", etc. So, in $\textsf{Pi}\ \textsf{x}{:}(LatticeLab) \longrightarrow (Socket\{\textsf{x.1}\}) \longrightarrow ...$ we have the name x is bound to the type of the first formal parameter, a tuple that contains a single element of type *LatticeLab*. The name x is in scope all the way to the right. So, the second argument is a tuple containing a *Socket* labeled by the *LatticeLab* provided in the first argument— x.1 projects out the first component of the first formal parameter. Similarly, the third and fourth arguments show a tuple y containing a *LatticeLab* and a string labeled with the contents of y.

In the body of sock_send_dyn, we check that label of the data is not greater than the label of the socket. If the check succeeds, we unlabel the socket and the data and call the sock_send library function. Otherwise, we simply return a unit.

163

```
 1   sig sock_send : (Socket) ⟶ (String) ⟶ ()
 2
 3   sig sock_send_dyn: Pi x:(Lab) ⟶ (Socket{x.1}) ⟶ Pi y:(Lab) ⟶ (String{y.1}) ⟶ ()
 4   fun sock_send_dyn (l) (sock) (m) (data) policy {
 5       if (less_than_eq (l, m) ) {
 6          sock_send (unlabel(sock)) (unlabel(data))
 7       } else {
 8          ()
 9       }
10   }
```

Figure 5.7: An enforcement policy for sockets using dependently typed functions

As another example of a dependently typed function, consider the type of the relabel operation as given in the SELINKS standard library.

$$\text{Pi } x{:}(\alpha, \beta) \longrightarrow \alpha\{x.2\}$$

This type states that relabel is a function (polymorphic in the type variables α and β) that takes a tuple of an α and β as an argument, where α is the type of the data to be labeled and β is the type of the label to be used. In this case, we bind x to the formal parameter, a record containing the data in its first component and the label in its second component. So, the return type of this function, $\alpha\{x.2\}$ shows that it returns a value of the same underlying type as the α argument passed in, but now, the value is protected by a label. In particular, the label that is used is the second component of the argument x that was passed in, i.e., x.2 projects out the second component of the input argument.

When type checking a function application, as in FABLE, we substitute the actual argument for the formal parameter in the return type. For instance, the function call, relabel(uid, Grant) from our previous example, in fact has the type s*Int*{(uid, Grant).2}. Clearly, a record projection like (uid, Grant).2 is not a value. We would like this function call to have the type *Int*{Grant}. Happily, the type reduction relation as defined in FABLE,

164

```
sig sock_send_dyn : (l←LatticeLab, Socket{l}) ⟶ (m←LatticeLab, String{m}) ⟶ ()
fun sock_send_dyn (l, sock_l) (m, data_m) policy {
    if (leq (m, l)) {
        sock_send (unlabel(sock)) (unlabel(data))
    } else {
        ()
    }
}

sig sock_send_bad : (l←LatticeLab, Socket{l}) ⟶ (l←LatticeLab, String{l}) ⟶ ()
fun sock_send_bad (l, sock_l) (l, data_l) policy { ... }
```

Figure 5.8: An enforcement policy for sockets using dependently typed records

translates naturally to SELINKS. We are able to reduce the expression (uid, Grant).2 to the

value Grant, as desired. Section 5.6 describes this type reduction process in further detail.

5.4.2   Dependently Typed Records

SELINKS provides special constructs to declare and directly manipulate depen-

dently typed records, rather than encoding them in terms of functions. In our experience,

dependently typed records have been the most common way of specifying dynamic label-

ings in SELINKS.

Figure 5.8 shows a program that uses dependently typed records. The function

sock_send_dyn is a revision of the function of the same name from Figure 5.7. Instead of

requiring the label and data to be passed to the policy as separate arguments, here, we can

package the label and data as a record and pass them together as a single argument.

The signature declares sock_send_dyn to be a curried policy function whose first

argument is dependently typed tuple, containing a lattice label l and a socket sock_l that

is protected by that label. The notation l←LatticeLab is a binding construct—it binds the

name l to the value stored in the first field of the tuple, and the name l is in scope for the remainder of the record declaration. To indicate that the socket is protected by the label l, we give it a dependent type *Socket*{l}, which makes clear the relationship between the fields of the tuple. Similarly, the next argument of sock_send_dyn is another pair, containing a label m and data_m, some data protected by m.

The function definition begins on line 2 where we define patterns (l, sock_l) and (m, data_m) that match the tuples provided to the function as arguments. In the body of the policy function, we can inspect the labels and only permit the data to be sent after checking that m is less than, or equal to, l.

Figure 5.8 concludes with a variation on sock_send_dyn that illustrates a tricky issue when programming with dependent types: shadowing of names can be problematic. The type signature of sock_send_bad shows its first argument as a dependently typed pair in which the first field is bound to the name l. As we've pointed out before, the scope of this name is for the remainder of the record—i.e., it is not in scope in the second argument of the function. In the second argument, we have another dependently typed pair, where we bind the first field to the name l. This much is fine, it is clear from the scoping rules that the string in the second argument is protected by the label that it is tupled with—there is no name l that is being hidden by the name binding in the second dependently typed pair.

The situation in the function definition is different. Here, the arguments (l, sock_l) and (l, data_l) pattern match the tuples, and in the second pattern, the name l shadows the name in the previous pattern. If we give sock_l the type *Socket*{l} in the body of the function, and allow l to be shadowed, then we have inadvertently severed the association between the socket and its label and mistakenly associated it with the label of the string.

166

There are many possible solutions to this problem. For instance, we could explicitly α-convert all terms using fresh names before type checking them. Or, we could use a nameless representation such as de Bruijn indices to represent variable bindings [21]. Or some combination of the two, like the recently proposed locally nameless approach [7]. However, the easiest solution, in terms of compatibility with the implementation of LINKS, is to forbid shadowing of variables that may appear in type-level expressions. In effect, this no-shadowing approach rules out programs such as our example, while complaining that the second binding of l shadows the first.

Dependently typed records in table types. Dependently typed records are not limited to function arguments. We can also use them to give types to database tables that store secret data (among other things). Returning to the employee database example from Section 5.2.2, we would like to make explicit the relationship between the access control list and the data that it protects in each row. Given that LINKS models a database row as a record, a natural model for this relationship is in terms of dependently typed record.

Figure 5.9 shows a small policy to protect salary data stored in our example employee database. Line 1 reproduces the type declaration for access control lists shown previously. At line 3, we use a dependently typed record to type each row in the "Employee table. The first field, acl, stores data of type *Acl*. The notation l←*Acl* (as in Figure 5.8) binds the name l to the value stored in the acl field of the record, and the name l is in scope for the remainder of the record declaration. The next field is the name field—we chose not to protect the name with a label. The sensitive data in each row is the salary of the employee. So, we give the salary field a dependent type *Int*{l}, indicating that it is protected

```
 1  typename Acl = List((username:String, uid:Int)) is lab;
 2  var employeeTab = table ``Employee'' with
 3                        (acl : l←Acl, name : String, salary : Int{l})
 4                        from (database ``EmpDB'');
 5
 6  typename EmployeeRec=(acl:l←Acl, name:String, salary:Int{l});
 7  typename Maybe (α) = [|Nothing | Just:α|];
 8
 9  sig releaseSalary: (Credential, EmployeeRec) ⟶ Maybe(Int)
10  fun releaseSalary (u:Credential, x:EmployeeRec) policy {
11      unpack x as (acl=m, name=_, salary=s_m);
12      if (member(u, m)) {
13          Just(unlabel(s_m))
14      } else { #Authorization failure
15          Nothing
16      }
17  }
```

Figure 5.9: A policy to protecting salary data in an employee database

by the label l; i.e., the contents of the acl field. The rest of the example uses the type alias

EmployeeRec to stand for this record.

Explicit scopes for names using existential packages. Before proceeding to the rest

of this example, we need to clarify a subtle issue in working with dependently typed

records—we need to ensure that names bound within a record never escape their scope.

For instance, consider the following (incorrect) program.

fun foo(x:*EmployeeRec*) { x.salary }

Here, we have a function that accepts an *EmployeeRec*, x, as an argument. This type is a

dependently typed record, where the salary field is protected by the contents of the acl

field. Now, since x is a record, in the body of the function, we could try to project out

the salary field from the record. While attempting to do so is certainly reasonable, giving

a type to the expression x.salary is problematic. The salary is field is protected by the

acl field, but there is no valid name in the current scope that can be given to this label expression. Clearly, giving this program the type $(\mathit{EmployeeRec}) \longrightarrow \mathit{Int}\{l\}$ is nonsensical—the label variable l is free. We could try to give x.salary the type $\mathit{Int}\{x.\mathsf{acl}\}$ (which would be accurate), but does not solve the problem of giving a type to the return type of the function because the pattern variable x is not in scope in the return type.

Our solution to this problem is standard. We view dependently typed records as a kind of existential package [149, 85, 101]. Under this view, we read the *EmployeeRec* type as follows:

> *EmployeeRec* is the type of a record of three fields, acl, name and salary, where *there exists a constant l* of type *Acl* in the acl field, a value of type *String* in the name field, and an integer labeled with l in the salary field.

As is standard when working with existential types, we expect these records to be manipulated using special **pack** and **unpack** operations, that control the scoping of the existentially bound names.

Unpacking a dependently typed record. To illustrate the usage of the **unpack** construct we return to Figure 5.9. At line 9-17, we have a policy function releaseSalary that controls access to the salary field of an employee record. This function takes a record with two fields as an argument. The first, u, is some representation of a user credential (say, some unforgeable representation of the UID of the user currently logged in to a system). The next argument, x is our dependently typed employee record. The goal of this policy function is to release the salary field to the caller, but only after checking that the credential u presented is mentioned in the access control list that protects the salary. However, as

illustrated before, projecting the salary field out of the record x is not permissible, since the existentially bound name l escapes its scope. The solution here is to "unpack" the record x to introduce the name l into the scope, before using the salary field.

At line 11, we use the syntax **unpack** x **as** $p; e$, for some record pattern p and expression e. We check that the names bound by pattern variables in p are distinct, and that they do not shadow any other names that can appear with a type-level expression. The names bound by the pattern are in scope for the expression e, and we check that no name bound in the pattern p escapes e. In this particular case, we bind the acl field to the name m, which allows us to give s_m, the salary field, the type $Int\{m\}$. In the remainder of the body, we check that the credential u is mentioned in the acl, and if it is, we unlabel the salary and expose it to the user. We package the result as an option type (Maybe(Int)), returning Nothing if the authorization check fails. Thus, the body of the **unpack** operation (and, as a consequence, the value returned by the function) can be given the type Maybe(Int), which does not leak the existentially bound variable m.

The SELINKS type checker ensures that fields in records whose types include existentially bound names can never be projected out of the record. They must always be accessed by unpacking the record. However, fields that do not include such names, like name, or even acl in our example, can both be projected out using the standard dot notation.

Constructing a dependently typed record with pack. The counterpart of the **unpack** operation (the destructor for a dependently typed record) is the **pack** operation (the introduction form). The example in Figure 5.10 illustrates its use. Here, we have a trusted login

```
1   typename Auth = [| Grant | Deny |];
2   typename Credential = (tag:l← Auth, userid:Int{l});
3
4   typename Maybe (a) = [|Nothing | Just:a|];
5   sig checkpw : (String, String) ⟶ Maybe(Int)
6
7   fun login (uname, password) policy {
8       switch(checkpw(uname, password)) {
9           case Nothing → error(''Failed login'')
10          case Just(uid) →
11              var cred =
12                  pack
13                      (tag=Grant, userid=relabel(uid, Grant))
14                  as Credential;
15              cred
16      }
17  }
```

Figure 5.10: A policy to construct unforgeable user credentials

function that produces an unforgeable user credential for a user after checking a username

and password against some password database. Our representation of a credential is the

type *Credential*, a dependently typed pair consisting of a tag of type Auth and a userid field

that is an integer labeled by the value stored in the tag field. Since only policy functions

can construct values with a labeled type, we can ensure that application programs cannot

forge *Credential* values.

In the body of the login function, we check the supplied username and password by

calling some library function checkpw and returns an option type Maybe(*Int*), containing

the user id of the user if the password check succeeds. We pattern match the result using

LINKS' **switch** construct, and if the check succeeds, we have to return a *Credential* value.

Lines 11-15 show a use of the **pack** construct. The syntax in general is of the form

var $x =$ **pack** e **as** t; e', where x is some variable, e and e' are expressions and t is a type.

The semantics is for e to be a record expression, that is to be packed into the existential

package (equivalently, the dependently typed record) described by the type t. In our case, we have e as (tag=Grant, userid=relabel(uid, Grant)). On its own, this expression can be given the type (tag=Auth **is** *lab*.Grant, userid=*Int*{Grant}), which although a valid (and extremely precise) type, fails to capture the relationship between the tag and userid fields. The type annotation t in the pack construct is a hint to the type checker to generalize the type given to e so as to introduce the relationship between the fields as prescribed by the *Credential* type. In this case the generalization succeeds and e is bound to the variable cred (of type *Credential*) in the remainder e'—in this case, cred is just returned.

Ad hoc inference for dependently typed records. The last example illustrates that the **pack** construct is simply an annotation that indicates how the type checker should generalize the type of a record expression. Fortunately, such a hint is only very rarely needed. Usually, the type checker is able to infer enough information from the context to decide how to generalize the type appropriately. For instance, if the programmer provided a signature for the login function (*String*, *String*) $\longrightarrow$ *Credential*, then there is sufficient information for the type checker to choose the right type without the need for the **pack** construct. Alternatively, if the record was to be passed as an argument to a function that expected a *Credential* argument the type checker would again generalize the type appropriately.

Similarly, the way in which we type check a function's arguments often allows the programmer to avoid writing explicit **unpack** operations. For example, in sock_send_dyn, the tuple patterns that appear in the function's declaration are type checked exactly as if they were the patterns that unpack a dependently typed record, with the scope of the **unpack** being the entire function body. This syntactic sugar for a function's arguments

```
1   sig getRecords: (Credential, Int) ⟶ List(String, Maybe(Int))
2   fun getRecords(cred, minSalary) server {
3     for (var row ← employeeTab)
4       where (switch (releaseSalary(cred, row)) {
5               case Just(salary) ⟶ salary > minSalary
6               case Nothing ⟶ false
7             })
8       [(row.name, releaseSalary(cred, row))]
9   }
```

Figure 5.11: An example program that enforces a policy in a database query

has proved to be very helpful in keeping the notation of our larger example programs relatively lightweight.

Securing a database query in SELINKS. We conclude this section by combining the programs of Figures 5.9 and 5.10 to apply access controls to our employee database. Figure 5.11 revises the getRecords server function first shown as a LINKS program in Section 5.2.1. Our goal remains to select only the records in the database for which the salary field exceeds the minSalary threshold. The SELINKS type checker ensures that we do the appropriate policy check before examining the salary field. In this case, the check amounts to a call to the releaseSalary function in the **where**-clause, passing in the user credential and a relevant row in the table. If the check succeeds, the option value returned contains the exposed salary field which we can then test.

This function returns a list of tuples, where each tuple contains the name and the salary from the rows that matched the query. Notice that on line 8, (the expression that computes the value returned by the comprehension) we have to perform an additional authorization check by calling releaseSalary again. Clearly, this is less than optimal. However, the scoping rules of list comprehensions in LINKS prevent us from simply re-using

the result of the authorization query performed in the **where**-clause. Another source of concern is the efficiency of the query. If as we have said before, **policy** functions like releaseSalary are pinned to the server, is it possible to compile this list comprehension to SQL in a manner that it can still be executed efficiently (and securely) within the database? The next chapter speaks primarily to the issue of efficiently enforcing security policies that span the server and the database.

5.5 Refining Polymorphism in SELINKS

Like most strongly typed functional languages, the type system of LINKS provides for ML-style let-polymorphism. In extending LINKS with security typing, this kind of polymorphism presents us with a useful opportunity. The parametricity results of Reynolds [81] and Wadler [136] guarantee that code that is polymorphic in the type of some data must view that data abstractly. This allows us to safely pass protected data to well-typed polymorphic code and rest assured that the data remains protected.

In Section 5.5.1 we show how the power of type polymorphism can be extended to polymorphism over terms that appear at the type level. The result, *phantom variable polymorphism*, confers two main benefits on SELINKS programs. First, as with standard polymorphism, we can derive useful parametricity results about programs that use phantom variable polymorphism. Additionally, we show how source programs can be simplified substantially through the use of phantom variables, both through the re-use of code (by avoiding over-specialization) as well as enabling a simple and tractable form of type inference.

```
1   sig add : (l←LatticeLab, Int{l}) ⟶ (m←LatticeLab, Int{m}) ⟶ Int{lub l m}
2   fun add (l, x_l) (m, y_m) policy {
3       relabel((unlabel(x_l) + unlabel(y_m)), lub l m)
4   }
5
6   fun addcaller (x:Int{High}, y:Int{Low}) {
7       add(High, x)(Low, y)
8   }
```

Figure 5.12: A lattice-based policy for integer addition

However, enhancing polymorphism in SELINKS by unleashing phantom variables
is only half the story. We must also rein in the power of standard type polymorphism to
cope with the cross-tier execution model of LINKS. Since we have no way of guaran-
teeing that code that runs at the client respects the abstractions specified in its types, we
need a way to control the degree of polymorphism that can be used in client code. In
Section 5.5.2, we show how to refine polymorphism in SELINKS by stratifying the lan-
guage of types into a family of kinds. This allows us to ensure that abstraction violations
in client code do not compromise the security of protected data.

5.5.1   Phantom Variables: Polymorphism over Type-level Terms

To illustrate the need for phantom variables, consider the sample program in Fig-
ure 5.12. This is a policy function that defines the semantics of integer addition under a
lattice-based information flow policy. As in our other examples, this policy function takes
two dependently typed pairs of a label and a protected integer as arguments. In the body,
we unlabel each integer, add them together, and then relabel the result with a label that is
the least upper bound of the two labels.

Even though this function receives the labels l and m as arguments, the runtime

175

```
1  sig add : phantom l.(Int{l}) ⟶ phantom m.(Int{m}) ⟶ Int{lub l m}
2  fun add (x_l) (y_m) policy {
3      relabel((unlabel(x_l) + unlabel(y_m)), lub l m)
4  }
5
6  fun addcaller (x:Int{High}, y:Int{Low}) {
7      add(x)(y)
8  }
```

Figure 5.13: A lattice-based policy for integer addition, with phantoms

behavior of this function is entirely independent of the concrete values chosen for the labels. To see why, recall that both unlabel and relabel operations are erased at runtime—they serve only as type coercions. After erasing these operations, we see that the body of the function is simply x_l + y_m. The only reason l and m are mentioned in the arguments is because we need to provide names for the labels of the integer arguments. Unfortunately, just because we need place-holders for the names of the labels, we force a caller of this function to pass in concrete label terms as arguments. In the function addcaller, these labels are particularly simple, but in practice, constructing these label terms often be cumbersome. We would much prefer a way of providing some constructs that allows the label names in the arguments of add to be bound, without requiring that the exact label terms be passed in as arguments.

The revised version of add in Figure 5.13 makes use of phantom label polymorphism and solves exactly this problem. The type signature of add states that the first argument is an integer labeled with a label l, for some label l. The notation **phantom** l. serves as a binder for l and the name is in scope all the way to the right. Similarly, the next argument is an integer labeled m, for some label m. The return type is the same as before. In the definition of add, notice there are no explicit term arguments for the label

Expression	e	$::=$	$\dots \mid$ phantom $\overrightarrow{y}.\lambda x{:}t.e \mid \dots$	abstraction with phantom variables $\overrightarrow{y}$
Types	t	$::=$	$\dots \mid \overrightarrow{y}.x{:}t_1 \to t_2$	function type
Environment	Γ	$::=$	$\dots \mid x\hat{:}t \mid \dots$	phantom variables bindings
Phase index	φ	$::=$	term $\mid$ type	phase distinction

$\Gamma \vdash_\varphi e : t$ Extensions to static semantics of FABLE

$$\frac{\overrightarrow{y} = FV(t) \setminus \text{dom}(\Gamma) \qquad \Gamma, \overrightarrow{y}\hat{:}lab \vdash t \qquad \Gamma, \overrightarrow{y}\hat{:}lab, x{:}t \vdash_\varphi e : t'}{\Gamma \vdash_\varphi \text{phantom } \overrightarrow{y}.\lambda x{:}t.e : \overrightarrow{y}.x{:}t \to t'} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash_\varphi e_1 : \overrightarrow{y}.x{:}t_1 \to t_2 \qquad \Gamma \vdash_\varphi e_2 : t_1' \qquad \sigma(t_1) \doteq t_1' \qquad \sigma' = (\sigma, x \mapsto e_2)}{\Gamma \vdash_\varphi e_1 e_2 : \sigma'(t_2)} \quad \text{(T-APP)}$$

$$\frac{x{:}t \in \Gamma}{\Gamma \vdash_\varphi x : t} \quad \text{(T-VAR)} \qquad\qquad \frac{x\hat{:}t \in \Gamma}{\Gamma \vdash_{\text{type}} x : t} \quad \text{(T-PHANTOM)}$$

Figure 5.14: Extending FABLE with phantom variables

variables l and m—which explains why we call them phantom variables. Since add does not receive the labels as concrete arguments, a result that concludes that the runtime behavior of add is parametric with regard to label values l and m is trivial—a result that is useful when reasoning about the correctness of the policy implementation.

Not having to pass in explicit term witnesses for these labels simplifies the code of the caller. For example, in the snippet below, we call the add policy function with a High and Low integer respectively. Notice that the caller does not even have to explicitly instantiate the phantom variables l and m—the type checker is able to infer the instantiations as High and Low, respectively, and compute the return type of this function as *Int*{lub High Low}. In the remainder of this section, we sketch an extension to the static semantics of FABLE that supports this form of phantom variable polymorphism.

Static semantics of phantom variables in SELINKS. Figure 5.14 begins with an extension to the syntax of FABLE (which mirrors the concrete syntax for phantoms in SE-LINKS). Term abstraction, $_{phantom}\overrightarrow{y}.\lambda x{:}t.e$ now binds two kinds of variables: the λ-bound variable x is standard, while the $_{phantom}$-prefixed list $\overrightarrow{y}$ binds *phantom label variables*. These represent label terms that require no run-time witness, and will be used to express the just-described flavor of polymorphism over the label expressions that appear in the first argument's type. Whereas previously the type of a function was simply $(x{:}t) \rightarrow t'$, we now record the list of phantom variables that can appear in the argument. In the type $\overrightarrow{y}.x{:}t_1 \rightarrow t_2$, the list $\overrightarrow{y}$ represents the free (phantom) variables in the formal parameter t_1. As before, x names the formal parameter. Both x and $\overrightarrow{y}$ are bound in t_2.

Next, we extend the typing environment Γ to include an additional form of binding for phantom variables: $x\hat{:}t$. Since all phantom variables are implicit parameters that have no runtime witness, we must ensure that these variables are never used in code that may be executed at runtime. Maintaining a separate binding construct in Γ will allow us to enforce this invariant. However, in order to so, we must also parameterize our static semantics with a phase index φ that indicates whether we are type checking a type- or a term-level expression. (We used a similar mechanism in the semantics of FLAIR to rule out side effects for type-level expressions.) Thus, our typing judgment has the form $\Gamma \vdash_{\varphi} e : t$.

The new rules in the system pertain mainly to the typing of abstractions and their applications. (The original semantics of FABLE are in Figure 2.4.) In (T-ABS), the first

premise ensures that the phantom variables $\overrightarrow{y}$ precisely record the free variables of the formal parameter's type, t. When a function is applied we will attempt to infer instantiations for all these free variables by unification. Ensuring that exactly the free variables are mentioned in $\overrightarrow{y}$ allows us to guarantee that such an instantiation, if one exists, can always be computed. The next premise ensures that the ascribed type of the formal is well formed. In particular, since the phantom variables are bound in t, we check t in a context extended with the phantoms. Importantly, the types of the phantoms show that they can only be instantiated with label-typed terms. Finally, the last premise, checks the body of the abstraction e as usual, in a context extended with the formal parameter x, and with the phantom variables. The rest of the type rules will ensure that the phantoms never appear in with a subterm of e that has operational significance.

In (T-APP), the rule for applications, the first two premises are standard. In the third premise, $\sigma(t_1) \doteq t_1'$ we compute a substitution σ of the phantom variables in the formal parameter t_1 that allows it to be unified with the type t_1' of the actual argument. A separate technical report [123] shows that computing such a substitution is decidable, given the constraints of the first premise of (T-ABS). Finally, in the conclusion, we substitute the actual argument e_2 for the formal parameter x in the return type, as is standard. However, we also instantiate all the phantom variables in t' with their substitutions σ.

Finally, we show the rules that ensure that phantom variables are never used in runtime computations. (T-VAR) asserts that variables in the context that are bound using normal bindings can be used in both the term and the type phase. However, according to (T-PHANTOM), phantom bound variables can only be used in the type phase. Ensuring that these variables are only used in the type-phase ensures that a policy function like add

```
1   fun leak() server {
2       var x:String{High} = read_password ();
3       consume(x)
4   }
5
6   sig consume : (α) ⟶ ()
7   fun consume(x) client { () }
```

Figure 5.15: Example illustrating how client code can violate its abstractions

is parametric in its phantom labels l and m.

5.5.2   Restricting Polymorphism by Stratifying Types into Kinds

While we can ensure that both the server and the database run type-correct LINKS

code, such an assurance is not easy to provide for the client tier. This means that we must

ensure that protected data (i.e., data that is given a labeled type) is never sent directly to

the client. However, a naïve use of type polymorphism, as in the example of Figure 5.15,

can cause this invariant to be violated.

The example shows a program with a server function leak and a client function

consume. In the body of leak, we read a High-security string x out of a secret password

file and then pass x to the client function consume. The type of consume shows that it is

parametric in the type of its argument. This ensures that it consume treats its argument ab-

stractly in its body, and indeed it does; it simply returns unit. However, the call to consume

in leak is dispatched across tiers to the client's web browser. Nothing prevents the client

from directly examining the secret argument x. In other words, untrusted client code (or

type-incorrect code) can freely mount abstraction violating attacks that can compromise

the security of protected data.

Our solution to this problem is to stratify SELINKS types into two *kinds*: U-kind and M-kind. A type *t* that inhabits the kind U is assured to contain no labeled types—U is the *unlabeled* kind. In contrast, a type *t* the inhabits the kind M may contain a labeled type—M is the *maybe-labeled* kind. We restrict client code to only manipulate data of types that reside in U-kind.

Examples of types that inhabit U-kind are *Int*, *String*, (*Int*, *String*), etc. Types that reside in M-kind include *Int*{Low}, *String*{High}, (*Int*{Low}, *String*), etc. The last of these types is interesting in that although it is itself unlabeled, since it contains an labeled component, it is considered to be in M-kind. We could also permit function types that have labeled types only in a negative position to reside in U-kind. For instance, the type (*Int*{High}) ⟶ () can be defined as residing in U-kind since it expects a protected data as an argument, rather than producing protected data as a result. However, such a function is useless at the client, since the client has no way to manufacture such an argument. For simplicity, our current implementation deems such a function type as being in M-kind.

We also include a sub-kinding relation—every type that resides in U-kind also resides in M. Additionally, by default, every type variable is considered to be instantiable only with types residing in U-kind. An explicit annotation is required in order to introduce a type variable at M-kind (using the syntax α::M).

Revisiting our example program, the type checker deems it insecure because the type variable in consume is treated as being a U-kinded variable; i.e., α::U. Since the variable x has an M-kinded type, the call to consume in the server function leak is type incorrect since an M-kinded type cannot be used to instantiate a U-kinded variable.

An attempt to circumvent this check by explicitly declaring α to be of M-kind is

shown below:

$$\mathbf{sig}\ \mathsf{consume} : (\alpha{::}\mathsf{M}) \longrightarrow ()$$
$$\mathbf{fun}\ \mathsf{consume}(\mathsf{x})\ \mathbf{client}\ \{\ ()\ \}$$

However, this program is also flagged by SELINKS because M-kinded types are not permitted in client code.

5.6 Expressiveness of Policy Enforcement in SELINKS

A central argument in favor of the FABLE-style of policy enforcement is the degree of expressiveness that it affords. This flexibility in FABLE is derived from two main insights. First, by proposing the notion of an enforcement policy in order to interpret a language of security labels, FABLE can enforce highly customized security policies. We have already seen that this basic idea translates directly to SELINKS.

The second key to the expressiveness of FABLE is its use of a simple but powerful combination of refinement types within a dependent type system. A FABLE policy designer willing to write complex type-level expressions can leverage the power of type-level computation to statically enforce a policy. Where such types become unwieldy, a policy designer can discharge the burden of proof to runtime—type refinements in FABLE allow the result of runtime checks to be incorporated in a flow-sensitive manner in the types of a program.

In this section, we discuss the implementation in SELINKS of these latter two features. Our bias in SELINKS is towards policies that are specified via dynamic labelings. In such a setting, the possibility of purely static enforcement of a security policy is severely limited. With this in mind, our implementation focuses mainly on flow-sensitive

type refinements based on runtime checks, leaving the implementation of type-level computation fairly rudimentary. We speculate that future implementations might benefit from type-level computation via powerful technologies like automated theorem provers.

5.6.1 Type-level Computation

The reduction of type-level expressions in a dependent type system is something of a double-edged sword. Importantly, performing computation at the type level increases the expressiveness on the type system. For example, our ability to enforce purely static information flow controls in FABLE and in FLAIR hinges crucially on the reduction of type-level expressions. But, in a language like SELINKS (or FABLE) which includes general recursion in the form of fixed points, performing computation at the type level leads directly to the undecidability of type checking. What is more, type-level computations may involve open terms and it is not always clear how such terms are to be reduced.

In light of the difficulties due to type-level computation, one might consider forgoing the expressiveness that it offers and settling for a more tractable system in which type-level expressions never need to be reduced. However, for a dependent typing system like FABLE, such an option is not viable. We turn to Altenkirch et al. [2] for a particularly pithy explanation of why this is so:

Let us examine the facts, beginning with the type rule for application:

$$\frac{\Gamma \vdash e_1 : (x{:}t) \to t' \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : t'[e_2/x]}$$

It's clear from the premises that, as ever, to check an application we need

183

to compare the function domain and the argument type. It's also clear from the rule's conclusion that these types may contain expressions. If computation is to preserve typings, then $f (2+2)$ should have the same type as f 4, so $t'[(2+2)/x]$ must be *the same type as $t'[4/x]$. To decide typechecking, we therefore need to decide some kind of equivalence* up to computation.

This argument makes it clear that in order to show that a calculus like FABLE is sound via subject reduction, we must include a type equivalence relation based on the reduction of expressions that appear in types (which is exactly the purpose of the (T-CONV) rule in the semantics of Figure 2.4). However, from the perspective of an implementation like SELINKS, ensuring that computation preserves typing is, at best, pedantic. After type checking a program and allowing it to run, we never actually re-check it after it has taken a step of reduction. Besides, given that we have proved that FABLE is sound, we can rest in the knowledge that if we were to include type-level computations in SELINKS, we would always be able to check that the type of a program is invariant under reduction.

Our current implementation of SELINKS takes a conservative (and practical) view of type-level reduction—we only include as much as is necessary to ensure that our example applications can be type checked. In particular, we concede the full expressive power of FABLE in that we are unable to statically enforce policies like information flow (we must rely on some runtime checks). In return, we profit from the simplicity of our current implementation.

In practice, conceding the expressiveness of static enforcement is not severe handicap. Enforcing a policy without runtime checks demands complete static knowledge of

the policy. For real applications, policies are typically not discovered until runtime. For instance, in our scenario which attempts to protect salary information in an employee database, static information about the labels stored in each row is scant. Even a special-purpose security type system like Jif [31] must rely on runtime checks to enforce this policy.

Our implementation currently supports only the following forms of type-level reduction:

1. **Reducing projections of fields from a record.** This allows us to type examples that use the relabel operator (among others). For instance, we are able to prove $Int\{(\mathsf{uid}, \mathsf{Grant}).2\}$ is equivalent to $Int\{\mathsf{Grant}\}$. We also support reductions that result from pattern matching a record—a variation on projecting a field from a record.

2. **Refinement due to type information.** In a computation where a variable l is free in a context where we have a precise type for l (such as the singleton type lab.High), we permit reduction to proceed by substituting for l with an expression derived from l's type.

Notably, our type equivalence relation does not extend to β-equivalence. If the additional power of type-level computations should become necessary, extending our existing techniques to include β-equivalence is feasible. In a related technical report [123], we discuss a simple (partial) decision procedure that can prove the equivalence of type-level expressions even in the presence of free variables and proves the procedure sound (including β-equivalence). However, if the main motivation for type-level reduction is expressive power, it is unclear that a purely syntactic equivalence algorithm, with ad hoc

```
1    sig print : (String{Low}) ⟶ ()
2
3    sig dynprint: (l←LatticeLab, String{l}) ⟶ ()
4    fun dynprint (l, x) {
5        switch (l) {
6          case Low ⟶ print (x)
7          case _ ⟶ ()
8        }
9    }
```

Figure 5.16: Refining a type based on the result of a runtime check

techniques to cope with free variables, is the way forward. A more promising approach

might be to interface with a more powerful formal tools (such as a automated first-order

SMT solver like Z3 [40]) in order to prove $\beta\eta$-equivalence of expressions.

5.6.2 Refining Types with Runtime Checks

In the absence of a complete type-reduction relation, the need to trade off static

enforcement in favor of dynamic enforcement is critical in SELINKS. In FABLE, we

supported a form of type refinement based on the results of pattern matching operations

performed at runtime. The SELINKS type checker reproduces this behavior by accumu-

lating equality constraints in each branch of a pattern matching statement. In deciding the

equivalence of types, SELINKS can appeal to the set of equality constraints to show that

two type-level expressions are equivalent (without needing to reduce them).

An example of this behavior is shown in Figure 5.16. At line 1 of this example,

we define an interface for a library function print which states that only strings labeled

Low are allowed to be printed to the terminal. Next, we have a function dynprint, whose

argument is dependently typed pair consisting of some label l and a string x labeled with

l. Statically, we only know that l inhabits the *LatticeLab* type, and thus x could be a High-security string. So, before we can print x, we must establish that l is Low. The body of dynprint does exactly this: it pattern matches l and in the case where it is Low, we call the print function.

The typechecker checks the print(x) function call in a context that includes the equality constraint l $\doteq$ Low. Given that the declared type of x is *String*{l}, in the presence of the equality constraint, the type checker is able to prove that *String*{l} is in fact equivalent to *String*{Low}—which is sufficient to type check the call to print.

5.7 Concluding Remarks

This chapter has described our efforts in adapting the core formalism of FABLE to a full-fledged programming language like LINKS. The result, SELINKS, offers a variety of constructs that aim to make programming with a dependently typed security-oriented programming language practical. However, as ever, the proof of the pudding remains in the eating. We defer a verdict on the practicality of SELINKS to the next chapter, wherein we describe our experience putting SELINKS to use in the construction of two secure web applications.

6. Building Secure Multi-tier Applications in SELINKS

We have used SELINKS to implement two applications. The first is SEWIKI, an on-line document management system that allows sensitive documents to be shared securely across a community of users. SEWIKI implements a combination of a fine-grained access control policy and a data provenance policy [22]. We have also implemented SEWINE-STORE, an e-commerce application that implements a fine-grained access control policy. We were able to reuse much of the policy code across the applications, suggesting that SELINKS promotes the modular enforcement of security policies.

Critical to ensuring reasonable performance for these applications is a novel compilation strategy for SELINKS code. Recall that a security policy in SELINKS is enforced by requiring application programs to include calls to privileged *enforcement policy* functions that guard access to protected resources. The naïve approach to compiling data access code that includes calls to these policy functions results in performance that is comparable to the server-centric approach to policy enforcement. Rather than insisting that policy functions execute only in the web server, our approach is to translate enforcement code to *user-defined functions* (UDFs) stored in the database. These functions can be called directly from queries running within the database. Performance experiments (Section 6.4) show that this cross-tier enforcement mechanism in SELINKS substantially improves application throughput when compared to server-only enforcement.

188

This gain in performance does not come at the expense of expressiveness. Enforcement functions can also be called as necessary within the server to enforce more expressive, end-to-end policies, e.g., for tracking information flow. Nor must we compromise on the benefit of protecting multiple applications with a common database-level policy. By associating the policy UDFs with views on database tables, multiple applications can be protected by a uniform policy. As such, cross-tier enforcement in SELINKS retains many of the best features of both the database and server-centric approaches while minimizing the drawbacks of each.

Furthermore, SELINKS makes secure applications more portable. Security policy enforcement relies only on common DBMS support for user-defined functions, and not on particular security features of the DBMS. Because programmers write enforcement functions in SELINKS' high-level language, they need not write variants of their application for different UDF languages. At the moment our implementation (Section 6.3) targets only PostgreSQL, but we believe other DBMSs could be easily supported.

In summary, the core contribution of this chapter is a demonstration that SELINKS is well-suited to building multi-tier applications that enforce expressive security policies in an efficient, reliable, and portable manner.

6.1 Application Experience with SELINKS

This section illustrates that SELINKS can support applications that enforce of fine-grained, custom security policies. We present two examples we have developed, a blog/wiki SEWIKI, and an on-line store SEWINESTORE. Demos of both applications can be found

189

at the SELINKS web-site, `http://www.cs.umd.edu/projects/PL/selinks`.

6.1.1 SEWiki

Our design for SEWIKI was motivated by Intellipedia, discussed in Chapter 1. As such, we aim to satisfy two main requirements:

Requirement 1: Fine-grained secure sharing. SEWIKI aims to maximize the sharing of critical information across a broad community without compromising its security. To do this, SEWIKI enforces security policies on *fragments of a document*, not just on entire documents. This allows certain sections of a document to be accessible to some principals but not others. For example, the source of sensitive information may be considered to be high-security, visible to only a few, but the information itself may be made more broadly available.

Requirement 2: Information integrity assurance. More liberal and rapid information sharing increases the risk of harm. To mitigate that harm, SEWIKI aims to ensure the integrity of information, and also to track its history, from the original sources through various revisions. This permits assessments of the quality of information and audits that can assign blame when information is leaked or degraded.

As discussed in the introduction, these requirements are germane to a wide variety of information systems, such as on-line medical information systems, e-voting applications, and on-line stores.

The implementation of SEWIKI consists of approximately 3500 lines of SELINKS code. It enforces a combined group-based access control policy and provenance policy.

```
typename Group = [| Principal: Int | Auditors | Admins |];
typename Acl = (read:List(Group), write:List(Group));
typename Op = Create | Edit | Del | Restore | Copy | Relab
typename Prov = List(oper:Op, user:String, time:String)
typename DocLabel = (acl: Acl, prov: Prov)
```

Figure 6.1: The representation of security labels in SEWIKI

As discussed in Chapter 5, implementing a security policy in SELINKS proceeds in three steps. First, we must define the form of *security labels* which are used to denote policies for the application's security-sensitive objects. Second, we must define the *enforcement policy* functions that implement the enforcement semantics for these labels. Finally, we must modify the application so that security-sensitive operations are prefaced with calls to the enforcement policy code. We elaborate on these three steps in the context of SEWIKI.

Security labels. Policies are expressed as security labels having type *DocLabel*, the record type shown in Figure 6.1. Documents are protected with security labels with the type *DocLabel*, which is a record type with two fields, acl and prov, representing labels from the access control and provenance tracking policies, respectively.

The access control part is defined by the type *Acl*, which is itself a record containing two fields, read and write, that maintain the list of groups authorized to read and modify a document, respectively. At the moment, we have three kinds of groups: Principal(uid), stands for the group that contains a single user uid; Auditors, is the group of users that are authorized to audit a document; and Admins, which include only the system administrators.

We also address information integrity by maintaining a precise revision history in the labels of each document node—this is a form of data provenance tracking [22]. This

191

part of a label, having type *Prov*, is also shown in Figure 6.1. A provenance label of a document node consists of a list of operations performed on that node together with the identity of the user that authorized that operation and a time stamp. Tracked operations are of type *Op* and include document creation, modification, deletion and restoration (documents are never completely deleted in SEWIKI), copy-pasting from other documents, and document relabeling. For the last, authorized users are presented with an interface to alter the access control labels that protect a document.

This provenance model exploits SELINKS' support for custom label formats. This policy does not directly attempt to protect the provenance data itself from insecure usage. We have shown in Chapter 2 that protecting provenance data is an important concern and is achievable in SELINKS without too much difficulty.

SEWIKI label-based policies can be applied at a fine granularity. In what follows we discuss SEWIKI's document model and the three policy elements of a *DocLabel*.

Document structure. An SEWIKI document is represented as a tree, where each node represents a security-relevant section of a document at an arbitrary granularity—a paragraph, a sentence, or even a word. Security labels are associated with each node in the tree. When manipulating documents within the server, the document data structure is implemented as a variant type. To store these trees in a relational database, we define a database table ''documents'' as shown in Figure 6.2.

The first column in this table, docid, is the primary key. The second column stores the row's security label, having type *DocLabel*. The third column's data has labeled type *String*{l}, i.e., it is protected by the label in the doclab field.

192

```
var doc_table_handle = table ''documents'' with
  (docid : Int, doclab : l← DocLabel,
   text : String{l}, ischild: Boolean
   parentid: Int, sibling: Int,
) from database ''docDB'';

fun access_text (cred, row) policy {
  unpack row as (doclab=dl, text=x | _);
  if (member(cred, dl.acl.read)) { Just(unlabel(x)) }
  else { Nothing }
}
```

Figure 6.2: A document model and enforcement policy for SEWIKI

The parentid field is a foreign key to the docid of the node's parent, the sibling field
is an index used to display the sub-documents in sequential order, and the ischild field is
used to indicate whether this node is a leaf (containing text) or a structural node (con-
taining sub nodes). To retrieve an entire document, we fetch the parent, look up all the
immediate children (by searching for nodes with a parentid of the parent), then recursively
look up all the children's children, until we retrieve all the leaf nodes. (Although other
representations of *n*-ary trees are possible, our choice is a fairly typical choice when trees
have to stored in a relational database.)

Enforcement Policy. Authorization checks in SEWIKI are implemented with an en-
forcement policy similar to the function access_text shown at the bottom of Figure 6.2.
The first argument cred is the user's login credential, and has type *Group*; the second ar-
gument, row is a record representing a row in the documents table. (LINKS type inference
infers the types of the first two arguments.) The function returns a value of the option
type Maybe(*String*). This function is marked with the **policy** qualifier to indicate that it is a
part of the enforcement policy.

```
1   fun getSearchResults(cred, keyword) server {
2     for(var row ← doc_table_handle)
3     where (var txtOpt = access_text(cred, row);
4             switch(txtOpt) {
5             case Just(data) → data ∼/.*{keyword}.*/
6             case Nothing → false
7             })
8       [row]
9   }
```

Figure 6.3: A function that performs a keyword search on the document database

In the body of the function, we first **unpack** the dependently typed record that represents the row (Section 5.4.2 explains this construct), binding the doclab field to dl and the text field to variable x (the syntax |_ allows the rest of the fields to be ignored). Since x has a labeled type *String*{dl}, prior to releasing x, access_text checks whether the user's credential is a member of dl's read access control list (using the standard member function, not shown). If access is granted, the released text is wrapped within the option-type constructor Just; otherwise, Nothing is returned.

Mediate actions. Figure 6.3 shows a function that performs text search on the document database. The getSearchResults function runs at the server (as evinced by the **server** annotation on the first line), and takes as arguments the user's credential cred and the search phrase keyword. The body of the function is a single list comprehension that selects data from the documents table. In particular, for each row in the table for which the **where**-clause is true, the row is included in the final list to which the comprehension evaluates. The **where**-clause is not permitted to examine the contents of row.text directly because it has a labeled type *String*{row.doclab}. Therefore, at line 3, we call the access_text policy function, passing in the user's credential and the row containing the security label and the protected text data. If the user is authorized to access the labeled text field of the row, then

access_str reveals the data and returns it within a Maybe(*String*). Lines 4-7 check the form of txtOpt. If the user has been granted access (the first case), then we check if the revealed data matches the regular expression. If the user is not granted access, the keyword search fails and the row is not included.

6.1.2 SEWineStore

We also extended the "wine store" e-commerce application, distributed with LINKS, with security features. We defined labels to represent users and associate these labels with orders, in the shopping cart and in the order history. This helps ensure that an order is only accessed by the customer who created it.

Order information in SEWINESTORE is represented with the following type:

typename Order = (acl:*Acl*, items:*List*(CartItem){acl})

An order is represented by a record with two fields. The acl field stores a security label while the items field contains the items in the shopping cart. The *Acl* type is the same as that used in SEWIKI, and many of the enforcement policy functions are shared between the two applications. In general, we found that access control policies were easy to define and to use, with policy code consisting of roughly 200 lines of code total (including helper functions). Our experience also indicates that it is possible for security experts to carefully program policy code once, and for several applications to benefit from high-reliability security enforcement through policy-code reuse.

6.2 Efficient Cross-tier Enforcement of Policies

LINKS compiles list comprehensions to SQL queries. Unfortunately, for queries like getSearchResults that contain a call to a LINKS function, the compiler brings all of the relevant table rows into the server so that each can be passed to a call to the local function. (The compiler essentially translates the query to SELECT * FROM documents.) This is one of the two main drawbacks of the server-centric approach: enforcing a custom policy may require moving excessive amounts of data to the server to perform the security check there. In this section, we present an overview of our cross-tier enforcement technique that seeks to remedy this shortcoming.

In order to remedy the inefficiency of pure server-side enforcement of a security policy, SELINKS compiles enforcement policy functions that appear in queries (like access_text) to *user-defined functions* (UDFs) that reside in the database. Queries running at the database can call out to UDFs during query processing, thus avoiding the need to bring all the data to the server. Our implementation currently uses PostgreSQL but should just as well with other DBMSs.

We implement this approach with three extensions to the LINKS compiler (in addition to the type system changes described in Chapter 5). First, we extend it to support storing complex LINKS values (most notably, security labels like those of type *DocLabel*) in the database. Prior to this modification, LINKS only supported storing base types (e.g., integers, floating point numbers, strings, etc.) in database tables. Second, we extend the LINKS code generator so that enforcement policy functions can be compiled to UDFs and stored in the database. Finally, we extend the LINKS query compiler to include calls to

Figure 6.4: Cross-tier Policy Enforcement in SELINKS

UDF versions of enforcement policy functions in generated SQL. Each respective step is labeled (1), (2), and (3) in Figure 6.4.

Representing complex SELINKS **data in the database.** The simplest way to encode a LINKS value of complex type into a database-friendly form would be to convert it to a string. The drawback of doing so is that UDFs would have to either directly manipulate the string encoding or else convert the string to something more usable each time the UDF was called. Therefore, we extend the LINKS compiler to construct a PostgreSQL *user-defined type* (UDT) for each complex LINKS type possibly referenced or stored in a UDF or table [102]. To define a UDT, the user provides C-style `struct` declaration to represent the UDT's native representation, a pair of functions for converting to/from this representation and a string, and a series of utility functions for extracting components from a UDT, and for comparing UDT values. UDT values are communicated between the server and the database as strings, but stored and manipulated on the database in the native format. In SELINKS, UDTs are produced automatically by the compiler.

At the top of the DBMS tier in Figure 6.4, we show the three columns that store SEWIKI documents. The doclab column depicts storage of a complex *DocLabel* record. This value is compiled to a C struct that represents this label. Section 6.3.1 discusses our custom datatype support in detail.

Compiling policy code to UDFs. So that enforcement policy functions like access_text can be called during query processing on the database, SELINKS compiles them to database-resident UDFs written in PL/pgSQL, a C-like procedural language. (Similar UDF languages are available for other DBMSs.) SELINKS extends the LINKS compiler with a code generator for PL/pgSQL that supports a fairly large subset of the SELINKS language; notably, we do not currently support higher-order functions. The generated code uses the UDT definitions produced by the compiler in the first step when producing code to access complex types. For example, LINKS operations for extracting components of a variant type by pattern matching are translated into the corresponding operations for projecting out fields from C structs. Section 6.3.2 describes the compilation process.

Figure 6.4 illustrates that UDFs are compiled from LINKS policy code in the file policy.links. We note that policy code can, if necessary, be called directly by the application program, in file app.links, running at the server.

Compiling LINKS queries to SQL. The final step is to extend the LINKS list comprehension compiler so that queries like that in getSearchResults can call policy UDFs in the database. This is fairly straightforward. Calls to UDFs that occur in comprehensions are included in the generated SQL, and any LINKS values of complex type are converted to their string representation; these representations will be converted to the native UDT

```
typedef struct Value {
  int32 vl_len_;
  int32 type;
  union {                          Variant* variant_in(cstring);
    Variant variant;              cstring variant_out(Variant*);
    Record record;                boolean variant_eq(Variant*, Variant*);
    int32 integer;
    text string;                   Variant* variant_init(text, anyelement);
    ...                            text variant_get_label(Variant*);
  } value;                         Record* variant_get_record(Variant*);
} Value;                           Variant* variant_get_variant(Variant*);
                                   int32 variant_get_integer(Variant*);
typedef struct Variant {          text variant_get_string(Variant*);
  int32 vl_len_;
  char* label;                     Record* record_in(cstring);
  Value value;                     cstring record_out(Record*);
} Variant;

typedef struct Record {           Record* record_init(anyelement);
  int32 vl_len_;                   Record* record_set(Record*, int32, anyelement);
  int32 num_args;                  text record_get_string(Record*, int32);
  Value value;
  Record rest;
} Record;
```

Figure 6.5: PostgreSQL User-Defined Types

representation in the DBMS. Section 6.3.3 shows the precise form of the SQL queries produced by our compiler.

6.3 Implementation of Cross-tier Enforcement in SELINKS

In this section, we present the details of the cross-tier policy-enforcement features of the compiler, overviewed in Section 6.2. We describe our data model for storing SELINKS values in PostgreSQL using user-defined types, illustrate how we compile SELINKS functions to user-defined functions, and explain how we compile SELINKS queries to make use of these functions and manipulate complex SELINKS data.

6.3.1   User-defined Type Extensions in PostgreSQL

User-defined types (UDTs) in PostgreSQL are created by writing a shared library in C and dynamically linking it with the database. For each UDT, the library must define three things: an in-memory representation of the type, conversion routines to and from a textual representation of the type, and functions for examining UDT values. Our in-memory representation for SELINKS values is centered around the **Value**, **Variant**, and **Record** structures, shown in Fig. 6.5.

The **Value** type defines a variable-length data structure that represents all SELINKS values. The first field vl_len_ (used by all the structures) is used to store the size (in memory words) of the represented SELINKS value. The remainder of the structure defines a tagged union: the field **type** is a tag denoting the specific variant of the value field that follows. All the possible forms of SELINKS values are recorded in the value union, including variants (like *Group*), records (like *Acl*), integers, and strings.

The **Variant** type represents an SELINKS value that inhabits a variant type. Every instance of a **Variant** type consists of a single *constructor* applied to a **Value** (stored in the value field of the **Variant** structure). For example, a SELINKS value like Principal(''Alice'') is represented in the database as an object of type **Variant** where the label field contains the zero-terminated string ''Principal'', and the value field is a **Value** whose **type** field indicates it is a string, with the string's value stored in the string field of the value union.

The **Record** type represents a record that can hold an arbitrary number of SELINKS values of different types. In particular, it is used to store the values of multi-argument labels; for example, ActsFor(''Alice'',''Bob'') is a **Variant** whose value field contains a **Record**-

typed value, ("Alice", "Bob"). A record's field names are omitted (the name is implied by position).

Some of the functions which work on these data types are listed in Fig. 6.5. The string conversion functions end with the suffixes _in and _out. These are used internally by PostgreSQL to translate between a UDT's in-memory and string representation. Since our composite types allow embedded values, the *_in functions must be able to recursively parse subexpressions (e.g., in "Principal("Alice")", the "Alice" subexpression must be parsed as a string).

The variant_eq function compares two **Variant** types for equality; in PostgreSQL, it is called by overloading the "=" operator. The variant_eq function implements a special pattern matching syntax, where the value "_" is treated a wild card, and will match any subexpression. For example,

$Acl(\texttt{"Alice"}) = Acl(\_)$ is true.

The variant_get_label function returns the text label of a **Variant**, while the variant_get_* functions get the value of the **Variant**; if the type does not match, a run-time error occurs. We require a different accessor function for each type because PostgreSQL requires return variables to have a type. On the other hand, the variant_init function, which creates a new **Variant** type, takes an argument of type **anyelement**. This is a PostgreSQL "pseudo-type" that accepts any type of argument; the actual type can be determined dynamically. This allows us to create user-defined functions that take a polymorphic type (such as access, described in the next section).

The **Record** functions are similar to the **Variant** functions. The record_get_* functions take a record x and a (zero-based) integer index i as arguments and returns the ith

201

component of the record *x*, if such a component exists and is of the proper type. If either condition is unsatisfied, then a run-time error results. record_init creates a new single record with the given value, while record_set sets a record's value, possibly extending the record by one element as a result.

In the remainder of this section we show how these types are used both within our compiled UDFs as well as in the body of SQL queries.

6.3.2   Compilation of SELinks to PL/pgSQL

To compile SELINKS functions to UDFs, we built a new LINKS code generator that produces PL/pgSQL code, one of PostgreSQL's various UDF languages. Prior to our extension the LINKS code generator could only generate JavaScript code for running on the client. PostgreSQL supports several different UDF languages, but PL/pgSQL is the most-widely used. It has has a C-like syntax and is fairly close to Oracle's PL/SQL.(Note that, unlike most database systems, PostgreSQL makes no distinction between stored procedures and user-defined functions.)

Code generation is straightforward, so we simply show an example. Figure 6.6 shows the (slightly simplified) code generated for an enforcement policy function called access, a generalization of the function access_text shown in Figure 6.2, that can take any type of argument (which is useful when labels annotate values of many different types, since we can write a single access function rather than one per type). A function definition in PL/pgSQL begins with a declaration of the function's name and the types of its arguments. Thus, line 1 of Figure 6.6 defines a UDF called access that takes three

202

```
1.   CREATE FUNCTION access(text,record,anyelement)
2.   RETURNS variant AS $$
3.   DECLARE
4.     cred   ALIAS FOR $1;
5.     doclab ALIAS FOR $2;
6.     x      ALIAS FOR $3;
7.   BEGIN
8.     IF member(cred,record_get_rec(
                   record_get_rec(doclab, 0),0)) THEN
9.       RETURN variant_init('Just', x);
10.    ELSE
11.      RETURN 'Nothing';
12.    END IF;
13. END;
14  $$ language 'plpgsql'
```

Figure 6.6: Generated PL/pgSQL code for access

arguments of built-in type text, a custom type record, and the special built-in "pseudo-type" **anyelement**. The **anyelement** type allows us to (relatively faithfully) translate usages of polymorphic types (as in the argument of our generalized access function) in SE-LINKS to PL/pgSQL. At line 2, we define the return type of access to be variant, since it is supposed to return an option type.

At lines 4, 5, and 6, we give names to the positional parameters of the function by using the ALIAS command (a peculiarity of PostgreSQL). That is, the first argument is named cred to represent the credential; the second argument is doclab to represent the security label of *DocLabel* type; the final argument x, is protected data of any type.

In the body of the function, lines 8-12, we check if the user's credential cred is mentioned in the doclab.acl.read field. Accessing this field requires first projecting out the record doclab.read, using record_get_rec(doclab, 0) and then the read field using a similar construction. The authorization check at line 8 relies on another UDF (member) whose definition is not shown here.

```
1. SELECT docid, doclab, text FROM
2.  (SELECT
3.    S.doclab as doclab,  S.docid as docid,
4.    S.text as text,
5.    access('Alice', S.doclab, S.text) AS tmp1,
6.  FROM documents AS S
7.  ) as T
8. WHERE
9.   CASE
10.    WHEN ((T.tmp1 = 'Just((_))'))
11.      THEN (variant_get_str(T.tmp1)
             LIKE '%keyword%')
12.    WHEN (true)
13.      THEN false
14.  END
```

Figure 6.7: SQL query generated for getSearchResults

If this authorization check succeeds, at line 9 we return a value corresponding to

the SELINKS value Just(x). Notice that the unlabel operator that appears in SELINKS is

simply erased in PL/pgSQL—it has no run-time significance. If the check fails, at line 10

we return the nullary constructor Nothing.

6.3.3 Invoking UDFs in Queries

The last element of our cross-tier enforcement strategy is to compile SELINKS

comprehension queries to SQL queries that can include calls to the appropriate policy

UDFs. This is built on infrastructure provided by Dubochet [43] (based on work in

Kleisli [142]). Prior to our extensions, the LINKS compiler was only capable of handling

relatively simple queries. For instance, queries like our keyword search with function

calls and case-analysis constructs were not supported.

Figure 6.7 shows the SQL generated by our compiler for the keyword search query

in the body of getSearchResults. This query uses a sub-query to invoke the access policy

204

UDF and filters the result based on the value returned by the authorization check. We start by describing the sub-query on lines 2–5. Lines 3 and 4 select the relevant columns from the documents table; line 5 calls the policy function access, passing in as arguments the user credential (here, just the username 'Alice', but, in practice, an unforgeable authentication token); the document label field S.doclab; and the protected text S.text, respectively. The result of the authorization check is named tmp1 in the sub-query.

Next, we describe the structure of the where-clause in the main query, at lines 8–14. We examine the the value returned by the authorization check; if we have obtained a Some(x) value, then we search x to see if it contains the keyword, otherwise the where-clause fails. Thus, at line 10, we check that T.tmp1, the result of the authorization check for this row, matches the special variant pattern Some((_)). In this case, the test on line 10 is satisfied if the value T.tmp1 is the variant constructor Some applied to *any argument*. If this pattern matching succeeds, at line 11, we project out the string argument of variant constructor using the function variant_get_str. Once we have projected out the text of the document, we can test to see if it contains the keyword using SQL's LIKE operator. Lines 12–13 handle the case where the authorization check fails.

Finally, we turn to line 1 of this query which selects only a subset of the columns produced by the sub-query. The reason is efficiency: we do not wish to pass the temporary results of the authorization checks (the T.tmp1 field) when returning a result set to the server.

Although our code generators are fairly powerful, there are some features that are not currently supported. First, our current label model requires storing a security label within the same row as the data that it protects. Next, our support for complex join queries

as well as table updates is still primitive. We anticipate improving our implementations to handle these features in the near future. Finally, as mentioned earlier, we do not allow function closures to be passed from server to the database; however, we do not foresee this being a severe restriction in the short term.

6.4   Experimental Results

In this section, we present the results of an experiment conducted to compare the efficiency of server-side versus database-side policy enforcement. We also examine other factors that come into play when running database applications, such as the number of rows being processed by the query, and the location of the database (local host or network). We also benchmark SELINKS against a simple access control program written in C. We show that, in the case of SELINKS, running a policy on the database greatly reduces the total running time compared to running the same policy on the server when tables are large (up to a $15\times$ speed-up). In addition, our C implementation highlights the high current overhead of SELINKS programs, while at the same time showing that our PostgreSQL implementation is comparable in speed (and shows a slight improvement when network latency is considered).

6.4.1   Configuration

Our system configuration is shown in Figure 6.8. We ran two different system configurations: a single-server mode (local) where the server and database reside on Machine A, and a networked version where the server runs on Machine B.

	Machine A	**Machine B**
CPU:	Intel Quad Core Xeon 2.66 GHz	Intel Quad Core Xeon 2.0 GHz
RAM:	4.0 GB	2.0 GB
HDD:	7,200 RPM SATA	7,200 RPM EIDE
Network:	100 Mbit/s Ethernet	100 Mbit/s Ethernet
OS Kernel	Linux 2.6.9	Linux 2.6.9
OS Distribution:	Red Hat Enterprise Linux AS 4	Red Hat Enterprise Linux AS 4
DBMS:	PostgreSQL 8.2.1	N/A

Figure 6.8: Test platform summary

For our test, we used the getSearchResults query presented in Fig. 6.7, which checks if a user has access to a record and, if so, returns the record if it contains a particular keyword. We generated two tables of random records (1,000 and 100,000 records), each comprised of 5–20 words selected from a standard corpus. Each record has a 10% probability of containing our keyword, and each record is labeled by a random access control label, which grants access 50% of the time. Thus, the query should return approximately 50 results and 5,000 results for the 1,000-record and 100,000-record tables, respectively.

In running our tests on SELINKS, we varied the number of records in the table (1,000 or 100,000), whether the policy was enforced on the server or the database, and the locality of the server (e.g., same or networked machine). We also created a C program that queries the database, manually performs the access control check, and searches for the keyword. The C program operates in one of three modes; no access control, server-side access control, or database-side access control, using the same SQL query as generated SELINKS program, including the database-level UDF function.We compare this program against our SELINKS implementation for all the tests above. All running times are the mean of five runs.

Keyword search on 1,000 rows

Keyword search on 100,000 rows

Figure 6.9: Throughput of SELINKS queries under various configurations

6.4.2 Results

The results of our experiment are summarized in Figure 6.9, which illustrate the time required to run the query on 1,000 and 100,000 records, on the left and right respec-

tively. The horizontal axis illustrates the language (C or SELINKS) and policy enforcement location (None, Database, or Server) used. For each language/policy pair, we show two bars representing the local or networked database configurations, respectively.

The highlight of both figures is the significant improvement shown in running an SELINKS policy on the database rather than the server. For the 100K-record example running over the network we see a $16\times$ improvement; for the 100K-record case with a local database the improvement is $7.5\times$; and for local and network queries on 1,000 records the improvement is $4\times$.

The current incarnation of SELINKS, however, is an interpreter language with few optimizations. Our C program results illustrate some more general results with regard to this technique. Consider the 100K-record results in Figure 6.9. First, running our C program with no policy enforcement takes a little over one second; this gives us a baseline for how long it takes to retrieve the full data set; this illustrates time that could possibly be saved by reducing the result set at the database. Our C implementation of server access control is *much* faster than our SELINKS implementation ($\approx 12.5\times$); this illustrates the lack of optimization in SELINKS. That said, the SELINKS database policy implementation is comparable to the C version on a single machine, (only 27% slower) and is marginally faster when network transmission is taken into account. It is interesting to note that, when running the database-policy versions, the SELINKS implementation actually slightly out-performs the C implementation; this indicates that the C implementation may not be as optimized as possible.

In summary, running SELINKS policies on the database instead of the server greatly improves performance, particularly for large queries. Based on the comparison with C, we

209

note that the SELINKS server component could benefit greatly from more optimization, while database-side enforcement is quite efficient.

6.5   Concluding Remarks

This chapter has demonstrated that feasibility of constructing realistic multi-tier web applications in SELINKS. We have implemented a wiki application that demonstrates multiple security properties, and have extended an existing LINKS e-commerce application with simple security protection. In general, we have found that SELINKS's label-based security policies are neither lacking nor burdensome, and the modular separation of the enforcement policy permitted some reuse of policy code between the two applications.

We have also argued that a multi-tier approach to security is necessary for expressing rich application policies while maintaining efficiency and trustworthiness. We have shown how SELINKS can model and enforce a variety of secure application policies, and have described how SELINKS implements such policies in the database by compiling the enforcement policy to user-defined functions in the database. Finally, we have shown that enforcing policies in the database, versus in the server, improves throughput in SELINKS by as much as an order of magnitude.

7. Related Work

This chapter describes various threads of related work. We begin with a discussion of security-typed languages. FLAIR is distinct from existing languages primarily in that it is intended to be extensible with custom policy enforcement mechanisms. In that respect, our work follows a long tradition of extensible programming languages, and we discuss these next. More recently, researchers have noted that dependent types are useful for extensibility; so, we discuss a variety of languages that include dependent types, whether for extensibility, program verification, or for security. We then turn to a discussion of the various kinds of security policies that we have explored. These include stateful authorization policies, policies for declassification, and data provenance policies. As far as the practical aspects of this work are concerned, the main related works are other projects that target multi-tier web applications—we discuss these next. The final section of this chapter ties together some loose ends and mainly places miscellaneous technical aspects of our work in context.

7.1   Security-typed Languages

Broadly speaking, *security-typed programming languages* augment the types of program variables and expressions with annotations that specify policies on the use of the typed data. These policies are typically enforced at compile-time by a type checker,

although some reliance on runtime checks is not uncommon. As such, FABLE, λAIR and FLAIR all fit this description. The basic idea of security typing is usually attributed to Volpano, Smith and Irvine [132]. Sabelfeld and Myers [111] provide comprehensive survey of a large body of work in this field.

Much of the work on security typing has focused primarily on information flow policies. We have sought to extend security typing beyond information flow. This appears to be a trend—a number of works concurrent with ours have also begun exploring security typing for other kinds of policies, and we discuss these elsewhere.

7.1.1 FlowCaml

Pottier and Simonet's FlowCaml language [104, 118] statically enforces an information flow policy for ML programs. Our encoding of information flow in Chapter 4 closely follows their type system—in fact, our correctness proof is via a translation to a subset of Core-ML, the underlying formal system of Core-ML. We also give a direct proof of correctness for the purely functional information flow policy of Chapter 2—this proof relies heavily on a syntactic proof technique also due to Pottier and Simonet.

Aside, of course, from extensibility, our work is distinct from FlowCaml in two main respects. First, FlowCaml extends Hindley-Milner-style type inference to include security types. We make no attempt to infer security-type annotations. However, Flow-Caml makes the simplifying assumption that security labels are always known statically. Although static labelings are permissible in our formal languages (and in SELINKS), our main focus is dynamically specified policies.

7.1.2 Jif

Jif [31], an extension of Java, is probably the most full-featured implementation of a security typed language. Unlike FlowCaml, Jif does not support full inference of security-type annotations. However, Jif does include dynamic labels (more on this below) and is thus more expressive that FlowCaml. Despite the lack of type inference, programming with an information flow policy in Jif is significantly easier than programming *with a similar policy* in SELINKS. In SELINKS, we require programmers to explicitly insert calls to enforcement policy functions to construct evidence that no illegal information leaks occur. This is the price of generality in SELINKS—Jif effectively "bakes in" these policy checks in its type system, so programmers need not insert these checks manually.

Zheng and Myers [149] formalize the use of *dynamic security labels* in Jif and show how data values can be used as security labels to express information flow policies. The technical machinery for associating labels to terms in their system is similar to ours—we both use forms of dependent types. There are two main differences. First, the security policy in Jif—an information flow policy in the decentralized label model [88]—is expressed directly in the type system whereas in λAIR both the security policy and the label model are customizable. As discussed in Section 2.2.4, dynamic labels for information flow policies can be encoded in FABLE. Second, we allow non-values to appear in types, e.g., lub l m in Figure 2.9. This is a more powerful form of dependent typing and allows us to encode a combination of static and dynamic policy checking. However, we need to take special care to ensure that type checking remains decidable—Section 7.3 contains a more detailed discussion of dependent types in FLAIR and SELINKS.

7.2 Extensible Programming Languages

Loosely, extensible languages aim to allow the user to modify the features of a language to suit his changing needs and purposes. This fits the description of FABLE and FLAIR, at least as far as security enforcement goes. In FABLE, extensions are defined using the enforcement policy and, in FLAIR using the type signature.

7.2.1 Classic Work on Extensible Programming Languages

Research in extensible programming languages dates back nearly 50 years. Standish surveys some of the early results [120] and provides a useful taxonomy of extensions. In his terminology, an extension is a *paraphrase* when a new construct is exchanged for an existing definition; e.g., by macro expansion. An enforcement policy function, like the apply function of Section 2.2.3, can be thought of as a paraphrase, in that it defines the application of labeled functions (which is not possible directly in the base language) in terms of existing constructs in the language (i.e., those used in the body of apply). Extensions can also be *orthophrases* where, an entirely new construct is added to a language. An example of an orthophrase in our context is the inclusion of a base term constant in a FLAIR signature. The third and final class of extensions are *metaphrases*, where new interpretations are given to existing constructs in the language. These are perhaps the most interesting aspect of our extensions. For example, the sub function in the policy of Figure 2.9 effectively introduces a subsumption rule into the type system. By using this function, an application program can give a new *type-level* interpretation to a term—i.e., a term of type t can be used at its sub-type t', where the subtype relation is defined by the

sub extension.

7.2.2 Extensible Type Systems

Researchers have explored how user-defined type systems can be supported directly via customizable *type qualifiers*. For example, CQual [50] is a framework that allows qualifiers to be added to the C programming languages. Qualifiers in CQual are arranged in a lattice and CQual uses various dataflow analyses to enforce properties like taintedness. Unlike a language like FlowCaml, which also tracks lattice-based qualifiers, CQual only tracks direct data flows and not implicit flows of the form described in Chapter 4. By focusing on direct data flows (among other reasons), CQual has seen broad practical applicability. For example, Shankar et al. [116] have used taint tracking in CQual to detect format string vulnerabilities and buffer overruns in C programs. Zhang et al. [148] and Fraser et al. [53] have used qualifiers to check complete mediation in access control systems.

Millstein et al [28, 3] have developed a qualifier-based approach in which programmers can indicate data invariants that custom type qualifiers are intended to signify. This is contrast to CQual, where one does not generally attempt to prove that the user-defined qualifiers correctly establish some property of interest. In some cases, Millstein et al. are able to automatically verify that these invariants are correctly enforced by the custom type rules. While their invariants are relatively simple, we ultimately would like to develop a framework in a similar vein, in which correctness properties for FABLE's enforcement policies can be at least partially automated. Marino et al. [82] have proposed using proof

assistants for this purpose, and we have begun exploring this idea in the context of FABLE policies.

While our security labels and type qualifiers share many similarities, our approach to type extensions is substantially different. In FABLE and FLAIR, the qualifier language is the same as the term language, i.e., qualifiers are specified using dynamic labels, where the labels themselves are program expressions. In FABLE, the semantics of labels are also given using constructs that are directly in the language. In contrast, Millstein et al's JavaCop includes a separate domain-specific language for introducing new type rules in the system. Because dependent types in FABLE and FLAIR essentially conflate the term language and the type language, we are able to describe typing constructs directly in the types of enforcement policy terms. But, the power of dependent types is not always necessary for extensibility—we discuss some alternatives next.

7.2.3   Extensions Based on Haskell's Type System

It has been said that the Haskell programming language is a "laboratory and playground for advanced type hackery" [70]. On occasion, Haskell's type system has been put to use to encode interesting security-related constructs.

Li and Zdancewic show how to encode information flow policies in Haskell [78]. They use type classes [137] in Haskell to define a meta-language of arrows [63] that makes the control-flow structure of a program available for inspection within the program itself. Their enforcement mechanism relies on the lazy evaluation strategy of Haskell that allows the control flow graph to be inspected for information leaks prior to evaluation. While

their encoding permits the use of custom label models, they only show an encoding of an information flow policy. It is not clear their system could be used to encode the range of policies discussed here. Besides, the reliance on a call-by-name evaluation scheme, with all the attendant challenges of handling side-effects and sequential computation, appears to be a considerable handicap of this approach.

Kiselyov and Shan [73] use type classes and higher-rank polymorphism in Haskell to encode a form of dynamic labeling. While their focus is on easily propagating runtime configuration parameters through a program, it appears as though their techniques could also be applied to labeling data with security policies. However, in the absence of true dependent types in Haskell, purely static enforcement of security policies using their method is not possible. Furthermore, by including security labeling as a primitive (rather than a derived construct), SELINKS makes it easier to manipulate labels and labeled data.

In the same work, Kiselyov and Shan provide a mechanism for configuration data to be passed as implicit parameters to functions. This resembles our use of phantom variable polymorphism in Section 5.5.1, but with an important distinction. Phantom label variables have no term level representation (i.e., they are phantom) and so the runtime behavior of a function is parametric with respect to its implicit phantom arguments. In contrast, Kiselyov and Shan's implicit parameters are concrete term-level arguments and can influence the runtime behavior of a function. Adding this to our SELINKS implementation might help reduce the annotation burden further.

7.3 Dependent Typing

Despite being more powerful than Zheng and Myers' dynamic labeling [149], security labels in FABLE and FLAIR still employ only a fairly weak and lightweight form of dependent typing. Traditionally, full-blown dependent types have been used as the basis for theorem provers like Coq [17], Isabelle/HOL [94] etc; for program verification, as in DML [145] and other dependently typed programming languages. In this section, we briefly survey these works. Aspinall and Hoffmann provide a useful introduction to dependent types [4].

7.3.1 Dependently Typed Proof Systems

Dependently typed formalisms like Pure Type Systems [10] and the Calculus of Constructions [36] have been used both as the basis of frameworks to design and formalize type systems as well as to build theorem provers [17, 11]. In these languages, the type and term languages overlap, allowing extremely expressive types to be used as specification. In comparison, dependent typing in FABLE and FLAIR is much simpler. Rather than conflate the language of types and terms, our approach only uses dependent types to express security labelings.

ATS is a programming language and a proof assistant based on a form of dependent types that has extensibility as one of its primary goals [144]. ATS (and its predecessor DML [145]) differs from Pure Type Systems and FLAIR in that the language of types and terms are completely separate. However, types can be indexed by so called *static terms*, essentially a purely functional lambda calculus at the type level. This separation

simplifies a number of issues in ATS—e.g., ATS by definitions rules out side effects and nontermination in type-level terms; in FLAIR, we require a (simple) effect analysis to achieve the same property. But, this separation means that indices in ATS have no runtime representation as they are drawn from a language intentionally kept separate from program expressions. Thus, while some of the statically enforceable policies we explore can be encoded in ATS, enforcing policies based on dynamic labels appears to be difficult since this requires indexing types with expressions drawn from the term language. (It may be possible to encode an indirect form of dynamic labels in ATS using singleton types.) FLAIR, in contrast, is specifically designed to make it easy to express and enforce policies using dynamic labels. ATS also includes linear types, a relative of the affine types we use in FLAIR [140].

Coq, Isabelle, ATS, and similar systems certainly outstrip FABLE in generality and power of static checking. Whereas these other systems target program verification, we have focused on showing that the simple form of dependent typing in FABLE and FLAIR can be used to provide useful assurances about the enforcement of security policies. Thus, while the generality of, say, Coq allows it to be used to define one of our type systems and to construct proofs that the type system is sound (e.g., we proved λAIR sound in Coq), the Calculus of Constructions does not intrinsically facilitate proving that well-typed terms enjoy relevant security properties (since it has no notion of security labels, complete mediation, etc.).

7.3.2 Dependently Typed Programming Languages

Cayenne [6] is a pure language in which the type and term languages coincide, and is possibly the first programming language to include the full power of dependent types. The resulting system is extremely powerful, though type-checking can be undecidable (as it is for the formulation of FABLE in Chapter 2). Cayenne focuses on static verification, while in our languages policies are enforced using a mixture of static and dynamic checks. Cayenne also does not support side effects, as we do in FLAIR.

Epigram [2] is another pure language with dependent types. Unlike Cayenne, Epigram ensures that type level expressions are always total functions, thereby ensuring decidability of type checking (and soundness of the underlying logic). Unlike our extremely simple proposal of ruling out recursion at the type level [123], Epigram employs much more sophisticated reasoning about structural recursion to ensure that functions are total. As with AIR policies in Chapter 3, Epigram also makes extensive use of dependently typed evidence—types like LEQ x y can be used as propositions that stand for integer inequality. In λAIR, certificates that witness these propositions were simply constructed at runtime using trusted function-typed base-terms in the signature. In contrast, the Epigram programmer specifies proof rules to interpret dependently-typed evidence and the compiler checks that these rules are always satisfied when certificates are constructed.

Concurrent with our own work, Nystrom et al. have developed a dependently typed extension to the X10 programming language [96]. They provide a way to associate a "constraint expression" (drawn from the term language) with the type of an X10 object. Although their focus is not security, it appears possible to use this feature to encode a

dynamic security labeling, much like in SELINKS. However, they provide no means of being able to define constraints that control the side effects of a program, or for that matter, to allow constraints themselves to be stateful (as we do in Chapters 3 and 4).

7.3.3 Dependent Types for Security

We are also not the first to use dependent types for security. Walker's "type system for expressive security policies" [139] is also dependently typed. Labels in Walker's language are uninterpreted predicates rather than arbitrary expressions—we are not aware of an earlier use of dependent types for security. Walker's system can enforce policies expressed as security automata—as we shows in Chapter 3, this kind of policy is also enforceable in λAIR. However, in Walker's system, the policy is always enforced by means of a runtime check. In order to recover some amount of static checking, Walker suggests that a user might add additional rules to the type system, though he is not specific about how this would be done. These additional rules would have to be proved correct with respect to a desired security property.

Aura is a programming language that incorporates an authorization logic in its type system using dependent types [64]. Statements from the authorization logic can refer to program values and specify constraints that must be satisfied in order for those values to be manipulated. Aura differs from FABLE and FLAIR in a number of ways. First, Aura focuses on authorization policies and on auditing. Although policies in Aura are user-defined to the extent that the authorization logic is general-purpose, the customizability does not extend to security automata, provenance, information flow and downgrading

policies as they do in FLAIR. Aura also uses dependently typed evidence [130], as we do in the enforcement of AIR policies. However, as with Epigram, Aura uses sound type rules to ensure that evidence objects are constructed properly, where we rely on ad hoc approaches like trusted runtime checks. However, unlike Epigram where the rules for constructing evidence are user-defined, Aura uses a fixed set of rules that capture the requirements of the underlying authorization logic. On a technical note, unlike in FLAIR, dependent types in Aura not quotiented by β-equivalence of type-level expressions, i.e., Aura never reduces type-level expressions using a rule like (T-CONV) in Figure 2.4. Vaughan et al. point out that this is unnecessary for authorization policies; however, we have found this to be useful for statically enforcing information flow policies. Aura is also purely functional and does not account for stateful policies.

Bengston et al's RCF [16] is a language equipped with dependent and refinement type which the authors have used to ensure the proper implementation of cryptographic protocols. Unlike dependent types in FABLE, which are just security policies, refinement types in RCF reflect underlying properties of the values that inhabit them—e.g., the type of access control lists that contain the username "Alice". These extremely precise types are useful in statically enforcing security policies, but in order to type check programs RCF must rely on an SMT solver [40]. (In Chapter 5, we speculated that using SMT solvers in order to prove type equivalences may be of use in SELINKS too.) But, because they always reflect structural properties of the underlying data, refinements in RCF make it difficult to enforce security properties that are not necessarily structural. For example, two strings in the program can be identical, yet have different provenance. In FABLE, it is possible to give these strings different types by associating different provenance labels

with each. In RCF, identical data values always inhabit the same types, so distinguishing between the provenance of these items purely in the types is impossible.

7.4 Security Policies

Here we survey work related to each of the security policy models explored in this dissertation. Information flow policies were discussed in Section 7.1; Bishop is a good reference for various models of access control [18]—we do not discuss these two kinds of policies further.

7.4.1 Security Automata

Schneider proposed using *security automata* to characterize the class of security policies that are enforceable by execution monitors [113]. In Chapter 3, we defined AIR, a language for specifying information release policies using security automata. AIR policies are actually a more general form of security automata called *edit automata* [79], i.e., automata that in the process of deciding whether an input word w is in a language L, may also transform w to some other word w'. AIR classes fit this description, since they may modify data before releasing it. To our knowledge, no prior work has used automata to specify the protection level and release conditions of sensitive data.

The canonical means of enforcing of security automata policies is through the use of reference monitors. Erlingsson and Schneider developed SASI [46] and its successor, PSLang/PoET [45], both inlined reference monitors to enforce security automata policies. Our approach is in contrast with SASI in that we support local policy state—i.e., bits of

automata state are maintained in proximity to the data that it protects and the association between the policy and data is reflected in the types. SASI, however, maintained global automata state and this was identified by Erlingsson as a main obstacle towards making it practical—specifying global policies required cumbersome state management and the runtime overhead of lookup up the relevant part of the global state was prohibitive. PSLang/PoET does support local policy state, but unlike λAIR, PSLang/PoET augments the run-time representation of protected data to include the policy. Dynamic labels in λAIR are more expressive—as discussed in Section 3.3, we can easily enforce secret sharing policies on related data. Local automaton state in λAIR is also likely to be useful when applying policies to concurrent programs—enforcement code does not need to synchronize on some global policy state, thereby allowing greater parallelism. Additionally, by reflecting the association between policy and data in the types, λAIR provides a way to verify that automata and protected data are always correctly manipulated. In the concurrent setting, dynamic labels in λAIR also clearly identify the synchronization requirements on policy state—thus λAIR's type system can improve reliability by helping prevent race conditions. As such, one could imagine putting λAIR to use to certify that IRMs correctly enforce their policies.

Security automata enforcement in λAIR essentially works by tracking the state of objects in types. As such, this is a form of *typestate*, a construct that dates back to Strom and Yemini [122]. The calculus of capabilities [37] provides a way of tracking typestate, using singleton and linear types (a variant of affine types) to account for aliasing. The Vault [41] and Cyclone [67] programming languages implement typestate checkers in a practical setting to enforce proper API usage and correct manual memory management,

224

respectively. λAIR's use of singleton and affine types is quite close to these systems. However, in these systems the state of a resource is a static type annotation, while in λAIR a policy automaton is first-class, allowing its state to be unknown until run time. Walker's type system [139] (discussed in Section 7.3.3) also supports first-class automaton policy state. But, in his system, there can only be a single policy automaton the definition of which is embedded into the type system. In contrast, our approach allows multiple automata policies to be easily defined separately.

As a consequence of dynamically defined policy state, full static verification of a security automaton policy is not possible in λAIR. Instead, we propose certifying the evaluation of policy logic by statically ensuring that proofs that support every authorization decision are constructed at runtime. This form of certified evaluation of authorization decisions has been explored in a number of contexts. For instance, certified evaluation is a feature of the SD3 trust-management system proposed by Jim [65]. Jia et al's Aura language [64] also maintains audit logs to record evidence to justify authorization decisions made at runtime. The architecture we propose for certified evaluation in λAIR is closely related to both these approaches. While more investigation is required, λAIR's ability to accurately track evidence in the presence of state modifications opens the possibility of certified evaluation of a wider class of stateful authorization policies, like those expressible in SMP, a stateful authorization logic recently proposed by Becker and Nanz [15].

7.4.2 Declassification Policies

The specification and enforcement of policies that control information release has received much recent attention. Sabelfeld and Sands [112] survey many of these efforts and provide a useful way of organizing the various approaches. AIR policies address, to varying degrees, the *what*, *who*, *where* and *when* of declassification, the four dimensions identified by Sabelfeld and Sands. Most of this work approaches information release from the perspective of information flow policies. As such, the security properties typically used with declassification are variants of noninterference or related forms of bisimulation. By contrast, the security theorem we show for the enforcement of AIR policies states that the program's actions are in accord with a high-level policy, not that these actions enforce an extensional security property (like noninterference). We believe that the two approaches are complementary. As Chapter 4 shows, λAIR is expressive enough to enforce noninterference-like properties too. By applying an AIR policy in combination with our information flow encoding, we could show a noninterference-like security theorem (e.g., noninterference until conditions [30], or robust declassification [147]) while being able to reason that a high-level protocol for releasing information is correctly followed.

AIR policies are defined separately from programs that use them, allowing them to be reasoned about in isolation. Most related work on declassification embeds the policy within program that uses it, obscuring high-level intent. One exception is work on *trusted declassifiers* [61]. Here, all possible information flows are specified as part of a graph in which nodes consist of either downgrading functions or principals, and edges consists

of trust relationships. Paths through the graph indicate how data may be released. AIR classes generalize this approach in restricting which paths may occur in the graph, and in specifying release conditions in addition to downgrading functions.

Chong and Myers [30] propose declassification policies as labels consisting of sequences of atomic labels separated by conditions c. Initially, labeled data may be viewed with the privileges granted by the first atomic label, but when a condition c is satisfied, the data may be relabeled to the next label in the sequence, and viewed at its privileges. Declassification labels are thus similar to AIR classes, with the main difference that our approach is more geared toward run-time checking: we support dynamically-checked conditions (theirs must be provable statically) and run-time labels (theirs are static annotations).

7.4.3 Data Provenance Tracking

Simmhan et al. [117] define data provenance to be "information that helps determine the derivation history of a data product, starting from its original sources". They also provide a useful survey of various models of provenance and techniques used to track provenance. Their survey proposes a taxonomy based on potential uses of provenance data. Our formal encodings of provenance in Chapter 2 show how to track provenance accurately, but are somewhat agnostic as to how this data is to be used. In the implementation of SEWIKI, (according to their taxonomy) we use provenance primarily for data quality, attribution, and to construct audit trails.

Buneman et al. [22, 23] discuss various approaches to provenance, specifically

in the context of database systems. They propose an alternative means of categorizing provenance in terms of the information it reflects, rather than potential usages of that information. In their terminology, *where*-provenance is information about the location (such as a specific database record) from which some data was retrieved. Alternatively, *why*-provenance refers to the source data that may have influenced the result of a computation. In these terms, we have focused primarily on why-provenance, although tracking where-provenance does not appear to pose serious difficulties.

Our approach to provenance tracking is closely related to computing dynamic program slices [141]. Cheney has also observed this connection and discusses ways in which ideas from slicing can be used to improve provenance tracking in databases [25]. Dependency correctness, the security property we prove for provenance policies, is also due to Cheney et al [26].

7.5   Web Programming

SELINKS expands on the original goal of LINKS [35], which is to reduce the impedance mismatch in programming multi-tier web applications. Our work aims to reduce the impedance mismatch faced when synchronizing the security mechanisms available in the various tiers of a web application. Several other languages also aim to simplify web programming by providing a unified view of the client and server tiers, e.g., Hop [115], the Google Web Toolkit (GWT) [57] and Volta [84]—we could have applied FABLE to any of these instead of LINKS. However, we found LINKS' three-tier solution (spanning client, server, and database) particularly attractive.

SIF [32] and Swift [29] are two Jif-based projects that aim to make web applications secure by construction. The former is a framework in which to build secure Java servlets. The latter, Swift, is a technique that permits a web application to be automatically split according to a policy into JavaScript code that runs on the client and Java code on the server. Being based on Jif, both these projects focus primarily on enforcing information flow policies; SELINKS aims to be more flexible by enforcing user-defined policies. Another distinction is that Swift and SIF target interactions between the client and server, while server-database interactions is the focus of our work with SELINKS. Despite these distinctions, there appear to be a number of ways in which Swift and SIF can complement SELINKS—we discuss some examples below.

In SELINKS, we expect programmers to insert annotations that partition the program into client and server components. The resulting partition is often fairly coarse grained, with more code running at the server than is strictly necessary for security. As such, the basic ideas of Swift could be applied to SELINKS to direct the partitioning of code into client and server components. A Swift-partitioned SELINKS program could have more code running at the client, potentially improving the responsiveness of the application and reducing load at the server.

In designing SIF, Chong et al. have worked out several useful idioms for enforcing information flow policies in web applications. For example, to provide a degree of protection against attacks like script injection, SIF places restrictions on the use of cascading style sheets and dynamically generated JavaScript. (Section 8.4 describes measures in SELINKS to defend against similar threats.) One could implement these behaviors in an SELINKS policy module and the type system could ensure that developers adhere to a

programming discipline that has been found to be effective with SIF.

7.5.1   Label-based Database Security

To understand the benefits of our approach with SELINKS, we consider some alternative approaches to securing a document-management application.

Database-side enforcement.   Some DBMSs aim to enforce a fine-grained policies directly, with little or no application assistance. For example, Oracle 10g [97] has native support for schemas in which each row includes a security label that protects access to that row. In this case, the label model and the enforcement policy are provided directly by the DBMS. As a result, the application code does not need to be trusted to perform the security checks correctly since the DBMS will perform them transparently. Application programmers need only focus on the functional requirements; i.e., they can write queries like (using LINKS syntax):

```
for(var row ← doc_table_handle)
  where (row.text ∼/.∗{keyword}.∗/)
    [row]
}
```

Native support for authorization checks in the DBMS can be optimized.

There are two downsides to a database-only enforcement model. The first problem is the lack of customizability. Each DBMS has different security mechanisms, and these may not easily map to application concerns. For instance, Oracle's row-level security is geared primarily to a hierarchical model of security labels, in which security labels are represented by integers that denote privilege levels. A user with privilege at level l_1 may access a row labeled l_2 assuming $l_1 \geq l_2$. While useful, this native support is not

sufficient to implement the label model we described above. For one, a typical encoding of access control lists in a hierarchical model requires a lattice model of security [42], rather than the total-order approach used in Oracle. Encoding principal sets in a hierarchical model is also not robust with respect to dynamic policy changes [125]. Furthermore, Oracle 10g is atypical—most DBMSs provide a far more impoverished security model. For instance, PostgreSQL [103], SQLServer [119], and MySQL [90] all provide roughly the same security model, based on discretionary role-based access control [95]. Object privileges are coarse-grained (read, write, execute etc.) and apply at the level of tables, columns, views, or stored procedures. By contrast, SELINKS labels can be defined using LINKS' rich datatype specification facility, labels can be associated with data at varying granularity (table, row, or even within a row), and these labels can be given user-defined semantics via the enforcement policy.

The second problem is that database-only enforcement does not solve the *end-to-end* security problem—while we may be confident that no data moves from the database to the server without proper access, the DBMS cannot ensure the server does not (inadvertently or maliciously) release the data inappropriately, e.g., by writing it to a publicly-visible web log. By contrast, SELINKS ensures that sensitive data, whether accessed via a database query or a server action, is always mediated by a call to the enforcement policy. This provides a level of trustworthiness similar to application-transparent enforcement within the DBMS, but with greater scope. Indeed, it opens up the possibility for enforcing policies that combine information available in the database and the server.

Server-side enforcement. Another common approach is to enforce fine-grained security

policies primarily in the server. This is the approach taken in the web application frameworks, like J2EE and ASP.NET. In J2EE [56], *Entity Enterprise Java Beans (EJBs)* are used to represent database rows at the server where row data is made available via user-defined methods. For our example we could define a method `findByKeyword` to search a document's text. Access to this method (and other operations) is controlled using the Java Authentication and Authorization Service (JAAS) to invoke user-defined functions under relevant circumstances. ASP.NET is similar to J2EE except it integrates more cleanly with authentication services provided by the Windows operating system [5]. Other lightweight approaches to web programming, like PHP [100] or Ruby On Rails [110], take a more ad hoc approach to security—a set of best practices is recommended to protect applications from common vulnerabilities like code injection attacks. All these approaches are extremely flexible. As with SELINKS, the developer can customize the label model and its semantics. Because policies are enforced at the server, they can consider server and database context, providing broader scope.

The main drawback of the server-side approach is the performance hit that comes with moving data from the database to the server, potentially unnecessarily. As illustration of this, Cecchet et al [24] report that J2EE implementations based on entity beans can be up to an order of magnitude slower than those that do not. That said, Spacco and Pugh [106] report that for the same application much of the performance can be restored with some additional design and tuning, but this can be a frustrating and brittle process. The other problem with server-side enforcement is trustworthiness: the application programmer is responsible for correctly invoking security policy functions manually, so that mistakes can lead to security vulnerabilities.

Hybrid enforcement. SELINKS essentially represents a kind of hybrid enforcement strategy: it presents a server-side programming model but compiles server functions to UDFs to allow them to run on the database and thus optimize performance. This same basic strategy could also be encoded "by hand." One could define a custom notion of security label (e.g., as a certain format of strings), and then write a series of user-defined functions akin to the SELINKS enforcement policy for interpreting these strings. The application writer would then be responsible for calling these functions during database accesses to enforce security. To avoid changing the application, a popular alternative is to have the DBMS perform UDF calls transparently when accessing the database via a *view* [97]. For example, we could define a view of our document table as containing only the docid and text fields; when querying these fields, calls to UDFs would be made by the DBMS transparently to filter results according to the hidden doclab field.

This by-hand approach has three main drawbacks, compared to what is provided by SELINKS. First, database-resident functions are painfully low-level, operating on application object *encodings* rather than, as in SELINKS, the objects themselves. Second, different DBMSs have different UDF languages, and thus a manual approach requires possibly many implementations; by contrast the SELINKS compiler can be used to target many possible UDF languages. Finally, if application programmers must construct queries with the appropriate calls to security enforcement functions there is the danger that coding errors could result in a policy being circumvented. While using views reduces the likelihood of this problem, there are still parts of the application that manage the policy, e.g., by updating the doclab portions of the objects, and these bits of code are subject to mistakes. The SELINKS type checker ensures that operations on sensitive

data (whether in queries like our keyword search example or in operations such as server logging functions) respect the security policy.

Finally, an important benefit of our approach is that it enables an application to make design choices that are pertinent to security, and have them reliably enforced in the server using the abstractions that are available there, if necessary. For example, in the case of SEWIKI, the policy label of a parent node indirectly and uniformly restricts access to all its children; labels that appear at child nodes may add to this restriction. To implement this tree-based semantics literally would require recursive policy checks. Doing this in the database would be both cumbersome and inefficient because SQL is not particularly well suited to handling recursive relations. Instead, we use code running in the web server to enforce the invariant that a node's label always reflects the policies of its ancestors' labels. Thus, even in situations where end-to-end tracking of information flow is not essential (as in our access control policy here), the flexibility afforded by server-side security enforcement in SELINKS is crucial both to reasonable efficiency and ease of use.

As a final note, a form of hybrid policy enforcement has also been recently proposed outside the context of web applications. SEPostgreSQL is an extension of PostgreSQL that aims to achieve end-to-end security through integration with the SELinux secure operating system [80]. SEPostgreSQL allows SELinux policy metadata to be associated with tables, columns, and rows in the database. Access to protected objects in SEPostgreSQL is mediated by the SELinux operating system's underlying reference monitor. This opens the possibility of enforcing a uniform policy throughout the operating system and database. However, it is unclear if the ideas behind SEPostgreSQL translate well to other DBMSs. In contrast, a key benefit of our work is portability. We rely only on

widely-used features of PostgreSQL (user-defined types and functions) which are also available in most other mainstream DBMSs. The assurance using FABLE types to avoid security bypasses is also unique to our approach.

7.6   Other Technical Machinery

Our technique of separating the enforcement policy from the rest of the program (in FABLE) is based on Grossman et al's *colored brackets* [58]. They use these brackets to model type abstraction, whereas we use them to ensure that the privilege of unlabeling and relabeling terms is not mistakenly granted to application code. As a result, we do not need to specially designate application code that may arise within policy terms, keeping things a bit simpler. We plan to investigate the use of different colored brackets to distinguish different enforcement policies, following Grossman et al.'s support for multiple agents.

8. Looking Ahead

This dissertation was motivated by a long-term vision of a modular, composable, and formally verifiable approach to the enforcement of security policies. In its idealized form, we conceive of a framework flexible enough to capture the idiosyncrasies of enforcement mechanisms used by real software implementations, yet precise enough to admit formal verification of high-level security goals. The enforcement mechanisms would be modular so that, once verified, they could be reused with a variety of applications with high assurance. For applications that needed to address a range of concerns, policies would be enforceable in combination. For example, some critical components of an application could be protected by strong, highly restrictive policies, while other parts could be secured by permissive, low-overhead policies. Source-level programmers would be able use this framework to ensure that new software was secure by construction. Legacy applications could also be retrofitted with security policies customized to match specific deployment scenarios.

While this vision remains a distant (perhaps even unattainable) goal, the work in this dissertation has made significant strides towards its fulfillment. We have shown, at least in theory, that a language-based framework can be expressive enough to verify the enforcement of many interesting security policies. We have also shown that the kernel of our approach is practical, and that programming with simple user-defined policies in

SELINKS is possible today.

But, there is much left to do. In this chapter we acknowledge some limitations of our work and identify a number of directions in which our work might be advanced.

8.1 Assessment of Limitations

This dissertation aimed to defend the thesis that language-based enforcement of user-defined policies can be both expressive and practical. By developing encodings of many kinds security policies, we have demonstrated that FLAIR is more expressive than any previously known security-typed language. Additionally, by proving that programs enjoy useful extensional properties as a consequence of type correctness, we have shown that our approach can provide a level of assurance that is competitive with more traditional, specialized security-typed languages. By developing SELINKS and using it to construct realistic applications, we have also showed that our approach can be applied in practice. Nevertheless, our work suffers from a number of limitations.

First, we are unable to precisely characterize the class of security properties enforceable in FLAIR. Many of the security policies we have explored here aim to establish *safety properties* for programs; i.e., they proscribe certain "bad events" from occurring during a program execution [75]. The security automata policies of Chapter 3 are a canonical example of such safety-oriented policies. A succession of results about the expressiveness of security automata have attempted to characterize the precise class of safety properties that they are able to enforce [113, 59, 13, 79]. By virtue of our ability to enforce automata-based policies, these prior results establish a useful lower bound on the

237

class of safety properties that can be enforced in FLAIR. But, this lower bound is not very precise. We have also demonstrated that FLAIR is powerful enough to enforce properties like noninterference, which are not safety properties. Noninterference-like properties have been variously categorized as *2-safety* properties [126] and, more recently, as *hyperproperties* [33]. But, only a small subset of hyperproperties are within reach of our enforcement techniques. For example, hyperproperties include *liveness properties*—i.e., properties that ensure that a program eventually does "something good". We have made no attempt to enforce liveness properties.

A more practical concern is that the generality of policy enforcement in FLAIR/SE-LINKS comes at a price. We require programmers to insert explicit calls to enforcement policy functions, rather than inferring the placement of the calls automatically. While the annotation burden is small for simple policies like access control, for more complex policies like information flow, as programs grow larger, the number of policy checks that must be inserted quickly becomes unacceptable. Specialized systems like Jif or FlowCaml do not suffer from this problem. Automatic insertion of policy checks is essential if SE-LINKS is to compete with these systems in the enforcement of information flow policies.

Another concern is the usability of our advanced typing constructs in a source language. Our experience with SELINKS is limited to FABLE-style purely functional policies. We have found that using dependent types to express a security labeling is fairly lightweight. However, the combination of dependent and affine types that we propose in FLAIR is much more burdensome for the programmer. As such, we conjecture that FLAIR may be more useful as the common intermediate language for a variety of systems that enforce special-purpose policies.

Finally, our theory of policy composition is fairly primitive. For example, although SEWIKI, our main example program, enforces a combined access control and provenance policy, our proofs of correctness apply only to each of these policies in isolation. We do propose some syntactic techniques to ensure that policies compose well, but this applies only to relatively simple modes of composition. Despite this weakness, our work is the first to provide a platform on which further work on modular proofs of correctness for composite policies may be explored.

8.2   Automated Enforcement of Policies

While SELINKS makes it easy to to enforce simple policies, the difficulty of enforcing more complex policies is often unacceptable. This has been one of the main factors in limiting our example applications to access control and simple provenance—information flow policies are too hard to enforce for large programs. The main thrust of the work proposed in this section is to make it easier to reason about and enforce some of the more complex policies.

The difficulty of enforcing a policy in FLAIR/SELINKS is due to three main concerns. We consider each of these and, in turn, propose ways in which these concerns may be addressed.

The first source of difficulty is that we require application programs to include explicit calls to enforcement policy functions. For security policies like information flow, complete mediation demands that enforcement policies mediate *all* operations on sensitive data; e.g., function calls must be performed indirectly by calling policy functions like

239

apply. So, we propose transforming programs to insert policy checks automatically.

Second, even for programs that include all the required enforcement policy calls, type checking relies on annotations that make explicit the connection between security labels and data. So, we propose using novel forms of type inference, suitable for use with dependent types.

Finally, even if a program can be shown to be type correct with respect to an enforcement policy, the security of the system relies crucially on a proof that the policy encoding correctly establishes some high-level goals. To reduce this burden, our proposal is to leverage automated theorem provers or proof assistants to mechanize much of the reasoning about policy correctness.

8.2.1   Transforming Programs to Insert Policy Checks

Currently, in order to enforce a policy, we expect a programmer to provide type annotations that protect sensitive data with their security labels. Programs that manipulate labeled data are required to include calls to policy functions—these calls must also be inserted manually into the program text. Instead of requiring the programmer to insert checks manually, we would prefer, given a manual security labeling, to insert the appropriate policy checks automatically. This section outlines a line of research that aims to achieve this goal. We begin with an abstract statement of the problem.

Given a program e and an enforcement policy P, where e is not type-correct with respect to P, compute a program e' (if one exists), where e is related to e' by the relation R, such that e' is type-correct with respect to P.

An enforcement policy P in FABLE (or, equivalently, a signature in FLAIR) is a purely declarative specification of the mechanism by which a policy is to be enforced. In order to automatically insert policy checks into a program, we could require the policy designer to additionally provide an algorithmic specification of P. Essentially, we could augment P with a set of rewriting rules that describe a program transformation. The general form of a rewriting rule could be "rewrite p_1 as p_2", where p_1 and p_2 are program *patterns*. The intention is to apply a collection of these rules to the source program e, rewriting sub-expressions of e that match the pattern p_1 according to the pattern p_2.

An example rule is shown below:

$$\text{rewrite} \quad (e_1 : (\alpha \rightarrow \beta)\{l\} \;\; e_2 : \alpha)$$
$$\text{as} \quad (\mathsf{apply} \;\; e_1 \;\; e_2)$$

In this case, the pattern $(e_1 : (\alpha \rightarrow \beta)\{l\} \;\; e_2 : \alpha)$ is matched by any sub-expression of e that is an application of a function e_1 to some argument e_2. The type annotations that appear in the pattern restrict e_1 to be typeable as a function with type $\alpha \rightarrow \beta$, labeled with the label l. Since labeled functions are not directly applicable, we need to wrap the application in a call to the apply enforcement policy function. The pattern p_2 does exactly this: it calls the apply function, passing in e_1 and e_2 as arguments.

This approach has a number of benefits. First, we aim to guarantee that the target program e' is type-correct with respect to the policy. This means that verifying the security of e' only requires reasoning about the declarative policy P, and not the (potentially complicated) rewriting rules, i.e., despite the added complexity of the program transformation algorithm, the trusted computing base is unchanged. Furthermore, if successful, this work would not only ease the construction of new secure programs, but would also

open the possibility of retrofitting existing programs with policy checks. However, making this idea work in practice will require addressing a number of concerns.

First, it may be possible to apply multiple rules to a single sub-expression. This could be either because we are attempting to enforce multiple policies simultaneously, or due to inherent ambiguities or non-determinism in the way in which a single policy is to be enforced. Thus, we would require some mechanism by which conflicts among the rules are to be resolved.

A related issue is the termination of the rewriting process. Can program fragments be re-written multiple times, and if so, can we provide any assurances that the rewriting process converges ultimately? Or, perhaps, non-termination (or, pragmatically, simply a long-running rewriting process) can be treated as a failure mode. If so, what are the ways in which this failure can be explained? That is, is the non-termination due to an ambiguity in the rewriting rules, or is it due to a badly formed source program? What about explaining other failure modes? For instance, in the simple example above, if the sub-program contains an application $e_1\ e_2$, where the type of e_2 does not match the type of the formal parameter of e_1, then, it seems reasonable that the rewriting process should fail. Is there a way in which these so-called "reasonable" failure modes can be characterized?

Finally, the abstract statement of the problem intentionally left the notion of correctness unspecified—this is potentially the most challenging issue to address. What are reasonable ways of constraining the behavior of the rewritten program e' so that it accurately reflects the intended semantics of the source program e, i.e., what is an appropriate definition for the relation R? Clearly, we would like to ensure that the programmer-specified labeling relationships in e are left unchanged in e'. But, we might also like to go

further and ensure that e' preserves the semantics of e in some non-trivial way, e.g., that the runtime behaviors of e (under a suitable operational semantics) and e' are identical, except for possibly failed policy checks in e'.

There are a number of related works that might provide suitable answers to each of these questions. First, work on aspect-oriented programming [72] is based on similar ideas. An aspect consists of a *point-cut*—a pattern that defines a set of program points of interest—and some associated *advice*, which defines an action to be performed at the points of interest. However, our notion of rewriting departs from aspects in two respects. First, we seek to transform a type-incorrect program by inserting "advice", in the form of the appropriate policy checks. Aspects have traditionally been used to alter the runtime behavior of type-correct programs, rather than to fix type-incorrect programs. Additionally, since policy checks in our framework can often be erased entirely, rewriting may not alter the runtime behavior of a transformed program at all. Nevertheless, it seems likely that the close connection to aspect-oriented programming can be used profitably; e.g., notions of correctness associated with aspects (like harmlessness [39]) may also be applicable in our setting.

Other ideas that may also be useful include work on blame assignment for use with software contracts [47] or hybrid programming languages [138]. It might be useful to show that if it is impossible to transform a program, or if a policy check in a trans-formed program fails at runtime, that the blame for the failure resides with the application program and is not some undesired artifact of the policy or rewriting rules. Finally, in this context, recent work on explaining failures of program analyses may also be rele-vant [133].

8.2.2 Inferring and Propagating Label Annotations

As a means of documenting security assumptions, security label type annotations appear to be far from optimal. Type annotations are buried within and are interspersed throughout the program text, causing the high-level intent of the connection between policy and data to be obscured. We would prefer to have a way of specifying a mapping between protected data and their policies separate from the program. From such a high-level specification, we would like to automatically derive the labeling annotations needed to type check a program.

Our vision for this element of proposed work is to complement the enforcement policy with a notion of a *labeling policy*, a high-level specification of the security constraints on the data sources and sinks in the program. Ideally, this arrangement would further simplify reasoning about the security of an application—given that the type checker can ensure that an application is consistent with its policies, a security analyst can ignore the program text altogether and focus only on the enforcement and labeling policies.

A labeling policy might include constraints of the following flavors. For instance, it might state that all resources accessible on the file system via a certain path are to be considered high-security. Or, for example, that the label of every network packet can be found at a specific offset from the start of the packet. Or, even that the label of a database element can be retrieved by following a succession of foreign-key/primary-key associations between multiple relations in a database. The labeling policy might also place constraints on data sinks like network ports or terminals, limiting the types of data that can be sent out on them.

244

Given a labeling policy, we would have to infer labels for objects manipulated by the program. For example, if the program opens a file by passing a string constructed by the concatenation of various constants, we would like to be able to discover the security label for that file by examining the labeling policy. Of course, this would require precise reasoning about the string constants in the program; but, it may be possible to leverage the power of dependent types in SELINKS to reason in this manner (as is done, say, in a language like Cayenne [6]). Alternatively, we could automatically give network packets the type of a dependent record, with the security label stored at the appropriate offset. We would have to develop similar methods to ensure that database queries always retrieved the appropriate labels along with the protected data.

Associating labels with the data sources and sinks is only one half of the problem. Given such an association, we would also like to use type inference to propagate label/data dependences throughout a program. As discussed in Chapter 7, this form of inference for security-type systems has been explored in context of FlowCaml. However, the static label model of FlowCaml is a simplifying assumption that is not applicable to our setting. Extending type inference to support dynamic labels would be a useful contribution in its own right—such a procedure could also simplify programming in Jif, for example. But, the generality of dependent types in SELINKS poses an additional challenge. Recent proposals like *liquid types* [109] may provide the basis of an inference mechanism for SELINKS.

8.2.3   Semi-Automated Proofs of Policy Correctness

A key benefit of our approach to policy enforcement is the ability to prove that programs satisfy end-to-end properties as a result of complete mediation. As such, conducting proofs of these properties is an integral part of our conception of the process of policy enforcement. That is, policy enforcement is not complete until all the policy code has been verified to correctly establish a desired security property of a program. However, at present, our proofs that a policy correctly enforces some high-level security property are entirely manual. Whereas our proposals for program rewriting and type inference sought to ease the process of constructing application programs, in this section we focus on simplifying the task of the policy designer and analyst.

In Chapter 7, we observed that our user-defined policies are a form of type-system extension. In this context, Millstein et al's work on semantic type qualifiers is relevant [28]—these are custom type qualifiers that a programmer can use to indicate data invariants to be enforced by the type system. Much like our policies, the specification of these qualifier extensions have to be proved to correctly describe the high-level invariants that they are intended to establish. Marino et al. have proposed using have proposed using proof assistants to partially automate these proofs of correctness [82].

Adapting this proposal to our setting is an interesting direction of future work. Whereas Marino et al. attempt to prove relatively simple syntactic properties, proving semantic properties (like noninterference) of our policy encodings would require a substantially larger effort. However, as discussed in Chapter 2, we have noticed that although non-trivial, our proofs of correctness are simplified considerably by the type-soundness

results of the underlying calculi, whether FABLE or FLAIR. Based on this experience, we conjecture that given a mechanized formalization of the metatheory of, say, FLAIR, a policy analyst could rely on several key lemmas in soundness proof to discharge a proof of the desired security property.

At the time of this writing, we have formalized a significant subset of the soundness proof of the λAIR calculus in the Coq proof assistant [17]. But, we have yet to apply this formalization to proofs of security properties.

8.3 Enhancements to Support Large-scale Policies

In the future, we would like to extend our evaluation of SELINKS by attempting to enforce large policies that address diverse security concerns. In this section, we consider how SELINKS might be scaled up to bring this goal within reach.

8.3.1 Interfacing with Trust Management Frameworks

Trust management frameworks were introduced by Blaze et al. as a means of specifying the authorization requirements of large distributed systems [19]. One basic design goal of trust management is to separate the specification of the policy from the means of its enforcement. Another goal is to formalize policy languages to the extent that a precise semantics can be given to a policy specification—both, to enable the construction of interpreters that can answer policy queries, as well as to make policies amenable to formal analysis so as to check compliance with high-level security goals [77].

Our approach to policy enforcement has hinged on the premise that reasoning end

to end about the security of a system depends crucially on a precise specification of the enforcement mechanism that ties a high-level policy to a program. In FABLE, the glue that ties a high-level policy to the program is the enforcement policy. We believe that the notion of an enforcement policy is particularly applicable in the context of trust management, in that it bridges the gap between specification and mechanism intentionally kept separate by trust management systems. By using an enforcement policy, our work admits proofs of end-to-end operational properties of programs. This complements prior work on policy analysis in trust management, which aims to guarantee that policy specifications themselves are consistent with high-level system objectives. Given a trust management policy, we could write enforcement policy glue code to ensure that application programs always include the appropriate calls to the policy interpreter. If successful, not only could we prove end-to-end properties of program executions, we could also check that these low-level security properties are consistent with the high-level objectives deduced from an analysis of the policy alone. Of course, this effort would begin with a study of existing policies formalized in trust management systems. For example, health-care policies formalized in Cassandra may be one starting point [14].

However, scaling SELINKS to be able to interface with trust management languages poses a number of interesting problems. For example, finding a reliable and transparent way to tie resources manipulated by the program to the policy elements that govern the usage of those resources. Throughout this dissertation, we have achieved this by associating a security label with each sensitive resource. However, interfacing with a trust management policy would be greatly simplified (and, indeed, enhanced) if the research proposed in Section 8.2.2 is successful. That is, given a specification of resources in a

248

language like SPKI/SDSI, we could automatically reconstruct the type-level labelings of objects in the program and propagate these throughout the program.

Interpreters for trust management languages are often stateful. For example, Becker and Nanz [15] describe the semantics of trust management policy language using a variant of Datalog extended with state modification effects. We conjecture that the techniques we have developed with FLAIR will be useful in this context. Extending SELINKS with FLAIR, and enhancing FLAIR with some of the ideas of the previous section to make it more usable at the source level, will also be an interesting line of research.

There are will also be some interesting engineering issues. Trust management was designed with distributed systems in mind. SELINKS also targets distributed systems, but is currently limited to the three-tier topology. Extending SELINKS to handle more general topologies of distribution would take a substantial effort. This would include a more sophisticated model of trust in the various nodes/tiers of a system. Our current model trusts the server and the database implicitly, but places no trust whatsoever in the client. An extension to more general topologies would requiring refining this model so that finer degrees of trust can be placed in each node.

8.3.2 Administrative Models for Policy Updates

Most existing security-typed languages assume that a program's security policy does not change once the program begins its execution. This is an unrealistic assumption for realistic long-running programs. For operating systems, network servers, and database systems, the privileges of principals are likely to change. New principals may enter the

system, while existing principals may leave or change duties.

On the other hand, it would be unwise to simply allow the policy to change at arbitrary program points. For example, if the program is unaware of a revocation in the security lattice it could allow a principal to view data illegally. More subtly, a combination of policy changes could violate separation of duty, inadvertently allowing flows permitted by neither the old nor the new policy.

In prior work, we proposed a security-typed language RX that permits security policies to change during program execution [125]. We equipped RX with an administrative model based on the RT role-based trust management framework [76]. In effect, elements of the policy define roles with *designated owners* who are responsible for administering the role's contents. Thus, only when the program is acting in a way trusted by that owner may the role be changed. We defined a type system to enforce this administrative model and, additionally, to ensure that updates do not cause undesirable information flows.

We propose adapting the ideas of RX for use in SELINKS. Since RX policies clearly rely on mutable state, once again, the ideas we developed for the enforcement of stateful policies in Chapter 3 are relevant. By including support of λAIR-style tracking of policy state in SELINKS, it should become possible to enforce RX's administrative models for policy updates. However, the flexibility of λAIR will allow us to easily explore other administrative models as well; e.g., a recent variant of RX proposes a more flexible administrative model [8]. Additionally, we would be able to investigate questions regarding the suitability of administrative models for policies other than information flow. For example, what is a suitable administrative model for information release policies like AIR? How would those models be combined with the administration of information flow

policies? This effort would also mesh well with a broader initiative that aims to integrate SELINKS with trust management.

8.3.3   Administrative Models for Policy Composition

In Chapter 2, we pointed out that our security theorems apply primarily in situations where only a single policy is in effect within a program. However, in practice, multiple policies may be used in conjunction and we would like to reason that interactions between the policies do not result in violations of the intended security properties.

In its simplest form, policies can be composed in a way such that different security policies govern different parts of the same application. For example, we may wish to track information flows on some data and just enforce access control on other data—and ensure that the two kinds of data never mix. The policy composition criteria that we defined in Chapter 2 (and Appendix A) apply to exactly this case. We show that by adhering to this simple form of compositionality, one can reason about the security of the entire system by considering each policy in isolation.

However, more interesting compositions of policies are also natural. For instance, a policy might state that data governed by *Alice*'s access control policy is subject to a lattice-based information flow policy once it is released to *Bob*. While the enforcement of *Alice*'s access control policy and the lattice-based policy may have been proved correct in isolation, it is not immediately clear that the composed policy does not violate the invariants of its components, much less that it meets some desirable composed semantics.

In the future, we could devise models to control how policies are allowed to be

composed. One might attempt to wrap enforcement policy code within a module, where the module interfaces, defined by the administrator of a policy, would specify all the invariants that must be preserved for a policy to be composed with other policies. A particularly simple example would be to limit a policy to be composed only with other policies that were administered by a trusted principal. However, more complex forms of composition may be possible as well. For example, policies could be combined using boolean formulas. We could also borrow ideas like inheritance and overloading from object-oriented languages (our design of AIR already hints at some of these directions), or have support for ML-style functors, to have better support for managing large policies.

8.4   Defending Against Emerging Threats to Web Security

In this dissertation, we have mainly considered threats due to insider attacks. However, several recent trends, particularly with respect to the world wide web, have made it possible for outsiders to subvert application-level security controls by mounting abstraction violating attacks. The line of research proposed in this section aims to address these web-based threats, either by extending SELINKS or by using SELINKS in conjunction with other kinds of defenses.

A classic example of a web-based threat is a cross-site scripting (XSS) attack. This can occur when a web page contains script content from a third party, such as an advertiser or other users. This script executes in the web browser with the same level of privilege as scripts that originated from the server and can steal steal private information from the client's web browser and possibly co-opt the client's web browser into attacking other

252

web sites. XSS attacks were identified as the most common security vulnerability in 2007 [86].

Without specific defenses, an application like SEWIKI is also susceptible to XSS attacks. Figure 8.1 illustrates a possible attack. The upper frame depicts a wiki document stored at the server, that contains a top-secret (at the left) and a secret component (at the right). When this document is requested by a client with clearance to view only the secret component, SEWIKI prunes the top-secret component of the document tree and serves only the secret component to the client. However, the malicious client inserts a script into the secret component. This script, designed to run in the web browser of a user with top-secret clearance, fetches the top-secret component of the document and redirects the web browser to the attacker's web site, `evilsite.com`, passing in the top-secret data as part of the query string. The lower frame shows the altered document stored at the server along with the attack script. At some point, a top-secret user requests the document and SEWIKI serves the entire document to the user (without pruning the top-secret part). The attack script then runs in the top-secret user's web browser and forwards the top-secret data to `evilsite.com`.

In designing SELINKS, we were careful to incorporate the trust assumptions of each tier in our model. However, our model only goes so far—attacks like XSS, or more recently, cross-site request forgery [12], fall outside the scope of our abstractions. Clearly, a comprehensive approach to web security demands that we pay attention to these very real threats.

In prior work, we have proposed addressing XSS attacks by relying on coopera-tion between the server and client [68]. In our approach (called BEEP), users (like the

The figure shows two stages of a diagram. The first stage (top) shows a document tree on a server with annotations. The annotations read:

"A document with a top-secret component and a secret component"

```
<script>
  var ts = doc.getElement("topsecret");
  doc.location="http://evilsite.com/?data="+ts
</script>
```

"A malicious user with only secret clearance"

The second stage (bottom) shows the attack progression with labels "server" and "client", "saves changes", "evilsite.com", and:

"The victim is a user with top-secret clearance"

Figure 8.1: A cross-site scripting attack on SEWIKI

top-secret user in our example) can run specially modified browsers that are trusted to enforce a policy that is included in each web page by the server. In this case, the trusted web browser can enforce policies that prevent potentially malicious scripts in the secret component of the document from running, or from reloading the browser etc. Thus, like SELINKS, BEEP relies on coordination across the tiers of an application (in this case, the client and server) to reliably enforce a security policy. (In fact, our implementation of SELINKS includes BEEP policies in every web page it serves. Thus, SEWIKI users running BEEP-enabled browsers are protected from the attack described here.)

We conjecture that as Web 2.0 applications with very rich client side features continue to gain prominence, enriching a browser's JavaScript runtime environment with the ability to enforce complex policies will become increasingly important. Java applets, which were once the main vehicle of interactive content on web pages, have been supplanted by AJAX-enabled JavaScript. Following the example of Java, where expressive security policies ranging from sand-boxing to stack inspection were found to be necessary, one might expect that it will be useful to enforce non-trivial JavaScript security policies within the browser.

As with BEEP and SELINKS, we conjecture that approaching these problems from an application-wide cross-tier perspective is likely to pay dividends. For example, servers can specify policies to be enforced at the client; or, clients can request content that match their security needs. Providing each application component with the ability to attest to its security claims and verify claims of other untrusted components will be challenging. However, recent adaptations of information flow policies to a cryptographic setting may help provide some of the answers [52, 131].

While BEEP protects clients from malicious code that might be served with a web page, a server also needs to be protected from a malicious client. For instance, web applications have complex control-flow properties that govern a client's workflow through the application. Violations of this workflow can cause the web application to enter an inconsistent state. One approach to solving this problem is to use a FABLE-style security automaton policy to ensure that each client request is consistent with the current state in the workflow of an application. We have experimented with this enforcement technique in SEWINESTORE, our e-commerce application. However, other techniques may also be applicable. For instance, ideas from system-call monitoring [66], originally developed to ensure that an operating system's integrity is not compromised when an application is attacked, can also be applied to web applications. A server side monitor could intercept all client requests and ensure that they conform to some specification of the client's expected behavior. Additionally, one might be able to adapt ideas from kernel-based control-flow integrity monitors to suit web applications [99]. By analyzing the source of a web application, one could automatically extract a model of the client's behavior which could then be enforced by a server-side monitor, on a per-session basis. Such an approach could be particularly effective for a multi-tier language like SELINKS, where the entire application's source could be analyzed at once to extract a precise model of its expected control-flow behavior.

8.5 Concluding Remarks

This chapter has acknowledged a number of limitations to the work described in this dissertation. In seeking to address these limitations, we have also described a number of directions in which our work might be advanced.

9. Conclusions

This dissertation has made a number of contributions in support of the contention that the enforcement of expressive user-defined security policies can be wide-ranging, reliable, and practical. Our evidence includes the following main elements:

- A succession of programming-language calculi—FABLE, λAIR, and FLAIR—which we have shown to be expressive enough to verify the enforcement of access control, provenance, information flow, downgrading, and automata-based policies, for both functional and effectful programs.

- The statement and proof of several useful end-to-end properties for programs that enforce each kind of security policy, demonstrating that our approach retains a key benefit of traditional security-typed languages, while exceeding prior approaches by being applicable to a broader class of security policies.

- An implementation of security typing in SELINKS, and the subsequent use of SE-LINKS in the construction of two realistic web applications, corroborating our claim of practicality.

In conducting the work described in this dissertation we have gained a number of insights. Our work arose from the observation (entirely obvious in hindsight) that to reason about the correct enforcement of a security policy in software, one has to make

the enforcement mechanism precise. A lot of the prior work on security policy design (e.g., the work on trust management described in Section 8.3.1) starts from the premise that policy specification and mechanism need to be separated. While this is useful for evaluating the high-level security objectives by reasoning about the policy in isolation, it does not allow reasoning about programs that enforce a policy, because the details of enforcement matter. For example, our proof of non-observability relies on a precise definition of how membership checks on access control lists are performed. Variations on the forms of checks can have significant consequences on the kinds of properties that can be proved. For example, in Chapter 2 we showed alternative encodings of access control in which an authorization check was performed using capabilities instead of interposing a check at each request. We noted that the capability approach can more easily be used support idioms like the delegation of access rights, but it may not be very robust against time-of-check/time-of-use bugs. By developing the concept of an *enforcement policy*, we were able to show that all the relevant details of the enforcement mechanism can be factored into a small set of verifiable functions.

We also find our basic approach (as exemplified by FABLE) attractive because it brings the benefits of security typing to common programming tasks. For example, the most common form of access control policy is extremely simple—a successful authorization check releases the protected data without any further constraints. However, despite its simplicity, access control is frequently implemented incorrectly; e.g., Security Focus regularly reports vulnerabilities where access control checks are bypassed due to a software error [114]. To prevent these errors, we developed a simple encoding of access control in FABLE. Happily, we found that programming with this encoding in SELINKS was also

259

easy. Nevertheless, the little assistance that the programmer provides in the form of label annotations was enough to prove a useful end-to-end property—i.e., non-observability.

Giving the programmer the freedom to chose the format of security labels is also extremely useful. We have argued that the specific choice of label model can have a profound impact on the kinds of security properties that can be enforced—e.g., role-based label models can be better for controlling policies that change at runtime [125]. But, on a still more practical level, this flexibility allows SELINKS to easily interface with existing systems that already use specific formats of security policies. Furthermore, allowing labels to be formed from arbitrary data (like our use of time-stamps in the provenance labels of SEWIKI), lets the programmer use types to help manage tasks that would not traditionally be within the purview of a security type system.

Dependent types have recently become a fairly popular approach to program verification. But, rather than unleash the full power of dependent types we have taken a lightweight, pragmatic approach that we hope hits a sweet spot. We use dependent types to express a security labeling and for giving precise types to evidence, but not to do full program verification (as in Epigram [2], Coq [17], Cayenne [6], etc.). Our experience with SELINKS indicates that this kind of dependent typing is relatively easy to use. Additionally, rather than focusing on static enforcement of policies (which is usually impossible, because most interesting policies are dynamic) we permit runtime checks to be used to discharge typing obligations. By using a kind of intensional type analysis we can ensure that all the necessary runtime checks are present. This means that programmer can quickly develop secure applications by inserting runtime checks wherever necessary to convince the type system of complete mediation. But, as an application matures, one

could write down more expressive types to get stronger static guarantees and remove some runtime checks, or at least move checks outside of certain interface boundaries.

Despite being so lightweight, we found that the combination of dependent types with a little bit of verifiable enforcement policy code can be extremely expressive. Even in its simplest form (FABLE), we were able to show encodings of many interesting purely functional (or flow-insensitive) policies. But, in order to encode flow-sensitive analyses, or to account for side effects, we needed more. By adding affine types (another relatively standard, off-the-shelf construct) to the mix were able to overcome this restriction. It appears as though our particular combination of a small amount of trusted code, dependent types, and affine types may be of fairly broad interest. Although we set out to design a framework for enforcing security policies, as we discussed in Section 7.2, it appears as though a language resembling FLAIR could be the basis of a more general-purpose type system that supports flow-sensitive user-defined extensions.

In conclusion, by developing FLAIR, this dissertation has contributed a new, more flexible approach to language-based security. To our knowledge, no prior framework has been able to enforce such a wide range of policies with an equally high level of assurance.

A. Proofs of Theorems Related to FABLE

A.1 Soundness of FABLE

Definition 5 (Well-formed environment). Γ *is well-formed if and only if*

> $(i.)$ *All names bound in Γ are distinct*
>
> $(ii.)$ $\Gamma = \Gamma_1, x : t, \Gamma_2$ $\Rightarrow$ $FV(t) \subseteq dom(\Gamma_1)$
>
> $(iii.)$ $e_1 \succ e_2 \in \Gamma$ $\Rightarrow$ $\Gamma \vdash_c e_1 : lab \,\wedge\, \Gamma \vdash_c e_2 : lab$

Lemma 6 (Canonical forms). *For Γ well-formed, all of the following are true.*

1. $\Gamma \vdash_c v_c : (\forall \alpha.t) \quad \Rightarrow \quad v_c = \Lambda \alpha.e$

2. $\Gamma \vdash_c v_c : (x{:}t_1) \rightarrow t_2 \quad \Rightarrow \quad v_c = \lambda x{:}t.e$

3. $\Gamma \vdash_{app} v_{app} : t\{l\} \quad \Rightarrow \quad v_{app} = (\!(\{l\}v')\!)$

4. $\Gamma \vdash_{pol} v_{pol} : t\{l\} \quad \Rightarrow \quad v_{pol} = \{l\}v'$

Proof. Straightforward from induction on the structure of the typing derivation. Observe that in *app*-context, terms such as $(\!(\Lambda \alpha.e)\!)$ and $(\!(\lambda x{:}t.e)\!)$ are not values. $\square$

Theorem 7 (Progress). *Given (A1) $\cdot \vdash_c e : t$. Then either $\exists e'.e \overset{c}{\rightsquigarrow} e'$ or $\exists v.e = v$.*

Proof. By induction on the structure of (A1).

Case (T-INT): n is a value.

Case (T-VAR): Inapplicable, since by assumption, $dom(\Gamma) = \emptyset$ and e is a closed term.

Case (T-FIX): e takes a step via (E-FIX).

Case (T-ABS), (T-TAB): e is a value.

Case (T-TAP): We either have $e = e\,[t]$ or $v_c[t]$. In the first case, we use the induction hypothesis and apply (E-CTX) to show that $e\,[t] \overset{c}{\rightsquigarrow} e'\,[t]$. In the second case, by the second premise of (T-TAP) we have (A1.2) $\Gamma \vdash_c v_c : \forall \alpha.t$. By canonical forms Proposition 6 on (A1.2), we get that v_c must be of the form $\wedge \alpha.e$ or $(\!(\wedge \alpha.e)\!)$ (since the types of both $\langle t \rangle u$ and $(\!(\langle t \rangle u)\!)$ are of the form $t'\{e\}$) and (E-TAP) is applicable.

Case (T-APP): If e is either $e_1 e_2$ or $v_c e_2$, then, by applying the induction hypothesis to the third and fourth premises respectively, we get our result using (E-CTX). If e is $v_c v'_c$ then, if $c = pol$, by canonical forms on third premise of (T-APP), $v_c = \lambda x{:}t.e$ and (E-APP) is applicable.

Case (T-LAB): $C(\overrightarrow{v_c}, e, \overrightarrow{e})$ is an evaluation context of the form $E_c \cdot e$. So, via the induction hypothesis $e \overset{c}{\rightsquigarrow} e'$ and by (E-CTX) the goal is established.

Case (T-HIDE), (T-SHOW): Application of the induction hypothesis to the first premise.

Case (T-MATCH): If e is **match** e' **with** ... then we just use the induction hypothesis on the fifth premise of (T-MATCH) and we have our result via (E-CTX). If, however, e is **match** v_c **with** $x_i.p_i \Rightarrow e_i$, then we must show that reduction via (E-MATCH) is applicable. To establish this, note that the third premise of (T-MATCH) requires $p_n = x_{def}$. Thus, it suffices to show that $v_c \succ x_{def} : \sigma$ for all label-typed values v_c. But, all *lab*-typed values are of the form $C(\overrightarrow{u})$ where each sub-term is also a *lab*-typed value. So, matching via (U-VAR) must succeed.

Case (T-UNLAB): In this case, $c = pol$. If $e = \{\circ\}e'$ then by using the induction hypothesis on the second premise we have our result via (E-CTX). If $e = \{\circ\}v_{pol}$, by the premise we have that $\Gamma \vdash_{pol} v_{pol} : t\{e\}$. Thus, from Lemma 6, v_{pol} must be of the form $\{e''\}v'_{pol}$ and reduction can proceed using (E-UNLAB).

Case (T-RELAB): In this case, $c = pol$. If $e = \{e''\}e'$ then by using the induction hypothesis on the second premise we have our result via (E-CTX). If $e = \{e''\}v_{pol}$, then, by definition, e is a value.

Case (T-POL): If we have $e = (\!|e|\!)$ we can use (E-BRAC) with the induction hypothesis in the premise. If $c = pol$ we can reduce by (E-NEST). Otherwise, if $e = (\!|v_{pol}|\!)$, then v_{pol} must be one of the following:

Sub-case $v = n$: In which case, a reduction via (E-BINT) is possible.

Sub-case $v = C(\overrightarrow{u})$: In which case, a reduction via (E-BLAB) is possible.

Sub-case $v = \lambda x{:}t.e$: In which case, $(\!|v|\!)$ is reducible using (E-BABS) to a value.

Sub-case $v = \Lambda\alpha.e$: In which case, a reduction via (E-BTAB) is possible.

Sub-case $v = \{e\}u$: In which case, $(\!|v|\!)$ is an application value.

Case (T-CONV): Straightforward from the induction hypothesis applied to the first premise.

$\square$

Proposition 8 (Well-formed sub-derivations). *If Γ is well-formed, and (A1) $\Gamma \vdash_c e : t$ contains a premise of the form $\Gamma' \vdash_c e' : t'$ then Γ' is well-formed and $\Gamma' \vdash_c t'$. Similarly, if (A1) contains a sub-derivation of the form $\Gamma' \vdash_c t'$ then Γ' is well-formed.*

Proposition 9 (Sub-coloring of derivations). *If, for well-formed Γ, $\Gamma \vdash_{app} e : t$, then $\Gamma \vdash_{pol} e : t$.*

Proposition 10 (Weakening). *Given $\Gamma \vdash_c e : t$ and Γ, Γ' well-formed. Then, $\Gamma, \Gamma' \vdash_c e : t$.*

Lemma 11 (Substitution). *Given $\Gamma_1, x{:}t_x, \Gamma_2$ well-formed and*

(A1) $\Gamma_1, x{:}t_x, \Gamma_2 \vdash_c e : t$

(A2) $\Gamma_1 \vdash_c v : t_x$

Then, for $\sigma = x \mapsto v$,

$$\Gamma_1, \sigma(\Gamma_2) \vdash_c \sigma e : \sigma t$$

Proof. Proved by mutual induction along with Lemma 12 on the structure of assumption (A1). We assume a standard definition of capture-avoiding substitution in $\sigma(e)$ and $\sigma(t)$. Throughout, we are free to assume $\sigma\Gamma_1 = \Gamma_1$, since $x \notin \text{dom}(\Gamma)$

Case (T-INT): Trivial.

Case (T-VAR): Here we have two sub-cases, depending on whether or not $x \in \text{dom}(\sigma)$.

Sub-case (a): (A1) is of the form $\Gamma_1, x : t_x, \Gamma_2 \vdash_c y : t_2$; $y \neq x$ and thus, $\sigma(y) = y$.
    We have two further sub-cases:

Sub-case (a.i): $y{:}t_2 \in \Gamma_2$. In this case, $FV(t_2) \cap \text{dom}(\sigma) \neq \emptyset$; thus our conclusion is of the form $\Gamma_1, \sigma(\Gamma_2) \vdash_c y : \sigma(t_2)$

Sub-case (a.ii): $y{:}t_2 \in \Gamma_1$. From our initial remark, we know that $\sigma\Gamma_1 = \Gamma_1$; thus, $\sigma t_2 = t_2$. Our conclusion has the required form $\Gamma_1, \sigma(\Gamma_2) \vdash_c y : \sigma(t_2)$.

Sub-case (b): (A1) is of the form $\Gamma_1, x{:}t_x, \Gamma_2 \vdash_c x : t_x$. But, $\sigma(x) = v$ and, so, from (A2), $\Gamma_1 \vdash_c \sigma(x) : t_x$ is trivial. Furthermore, $\sigma(t_x) = t_x$, since $\Gamma_1 \vdash t_x$ and, by Proposition 8, $x \notin \text{dom}(\Gamma_1)$. Finally, we have our conclusion from weakening, i.e., Proposition 10.

Case (T-FIX): From α-renaming, we have $f \notin \text{dom}(\sigma)$. Thus, $\sigma(\textbf{fix } f.v) = \textbf{fix } f.\sigma(v)$. Now, we can use the induction hypothesis on the second premise to derive $\Gamma_1, \sigma(\Gamma_2) \vdash_c \sigma(v) : \sigma(t)$. The first premise follows from the mutual induction hypothesis of Lemma 12.

Case (T-TAB): The first premise of (A1) gives us $\Gamma_1, x{:}t_x, \Gamma_2, \alpha \vdash_c e : t$. Now, since $\alpha \notin \text{dom}(\sigma)$, we can use the induction hypothesis to get $\Gamma_1, \sigma(\Gamma_2), \alpha \vdash_c \sigma(e) : \sigma(t)$. The conclusion follows immediately.

Case (T-TAP): We use the mutual induction hypothesis of Lemma 12 to establish that $\Gamma_1, \sigma\Gamma_2 \vdash \sigma t$. Now, we use the induction hypothesis on the second premise, and in the conclusion we have the type $(\alpha \mapsto \sigma(t))\sigma t'$. Since $\alpha \notin \text{dom}(\sigma)$ we can rewrite this type as $\sigma((\alpha \mapsto t)t')$, as required.

Case (T-ABS): Our goal is to show, via (T-ABS), $\Gamma_1, \sigma\Gamma_2 \vdash_c \lambda y : \sigma(t_y).\sigma(e) : \sigma(t)$, since by α-conversion, $\text{dom}(\sigma)$ cannot mention y.
    From inversion of (A1), we can obtain from the first premise

$$\Gamma_1, x{:}t_x, \Gamma_2, y{:}t_y \vdash_c e : t$$

From the induction hypothesis applied to this judgment we can obtain

$$(\text{T1}) \qquad \Gamma_1, \sigma(\Gamma_2), y{:}\sigma(t_y) \vdash_c \sigma(e) : \sigma(t)$$

Thus, to reach our goal, we use this last judgment (T1) in the second premise of (T-ABS). The first premise follows from the mutual induction hypothesis.

Case (T-APP): The induction hypothesis applied to the first premise gives

$$\Gamma_1, \sigma(\Gamma_2) \vdash_c \sigma(e_1) : (x{:}\sigma(t_1)) \rightarrow \sigma(t_2)$$

and to the second premise gives

$$\Gamma_1, \sigma(\Gamma_2) \vdash_c \sigma(e_2) : \sigma(t_1)$$

Thus, in the conclusion, we get $(x \mapsto \sigma(t_1))\sigma(t_2)$. Again, as with (T-TAP), via α-conversion, we can ensure $x \notin \text{dom}(\sigma)$ and we can rewrite this type as required to $\sigma([x \mapsto t_1]t_2)$.

Case (T-LAB): We use the induction hypothesis on each of the n-premises, obtaining $\Gamma_1, \sigma\Gamma_2 \vdash_c \sigma(e_i) : lab$ for the ith premise. For the conclusion, we note that $\sigma(C(\overrightarrow{e})) = C(\sigma(\overrightarrow{e}))$ and obtain $\Gamma_1, \sigma\Gamma_2 \vdash_c \sigma(C(\overrightarrow{e})) : \sigma(lab \sim C(\overrightarrow{e}))$, the desired result.

Case (T-HIDE), (T-SHOW): Straightforward use of induction hypothesis on the premise.

Case (T-MATCH): Premise 1 follows from the induction hypothesis. The second premise applied to $\sigma(t)$ follows from mutual induction. Premises 3 and 4 are trivial, since v is a closed term and $\text{dom}(\sigma)$ does not include any of $\overrightarrow{x_i}$. For the fifth premise for each p_i we use the induction hypothesis again and similarly for each e_i. The sixth and seventh premises are also established by the induction hypothesis.

Case (T-UNLAB), (T-RELAB): Induction hypothesis on the first premise.

Case (T-POL): We have $\Gamma_1, x{:}t_x, \Gamma_2 \vdash_c (\!|e|\!) : t$ with $\Gamma_1, x{:}t_x, \Gamma_2 \vdash_{pol} e : t$ in the premise. Now, if (A2) is $\Gamma_1 \vdash_{pol} v : t_x$, then we can use the induction hypothesis to establish $\Gamma_1, \sigma\Gamma_2 \vdash_{pol} \sigma(e) : \sigma(t)$ and conclude with (T-POL). However, if (A4) is $\Gamma_1 \vdash_{app} v : t_1$, then we must first use Proposition 9 before proceeding as before.

Case (T-CONV): Applying the induction hypothesis to the first premise, we obtain $\Gamma_1, \sigma\Gamma_2 \vdash_c \sigma e : \sigma t$. We proceed by induction on the structure of the second premise of (A1) $\Gamma \vdash t \cong t'$, to show that $\sigma\Gamma \vdash \sigma t \cong \sigma t'$.

Sub-case (TE-ID): Trivial.

Sub-case (TE-SYM), (TE-CTX): Induction hypothesis.

Sub-case (TE-REFINE): $e \succ e' \in \Gamma \Rightarrow \sigma e \succ \sigma e' \in \sigma\Gamma$.

Sub-case (TE-REDUCE): By construction, we have that $\forall \sigma'. \sigma' e \overset{c}{\leadsto} \sigma' e'$. $\qquad\square$

Lemma 12 (Substitution for type well-formedness judgment). *Given well-formed $\Gamma_1, x{:}t_x, \Gamma_2$. If the following conditions are true:*

(A1) $\Gamma_1, x{:}t_x, \Gamma_2 \vdash t$

(A4) $\Gamma_1 \vdash_c v : t_x$

Then, for $\sigma = x \mapsto v$,

$$\Gamma_1, \sigma\Gamma_2 \vdash \sigma(t)$$

Proof. By mutual induction with Lemma 11 on the structure of assumption (A1).

Case (K-INT): Trivial.

Case (K-TVAR): $\sigma = (x \mapsto v)$; Thus, $\alpha \in \Gamma_1, \sigma\Gamma_2$.

Case (K-LAB): Trivial.

Case (K-SLAB): We use the mutual induction hypothesis to show $\Gamma_1, \sigma\Gamma_2 \vdash_{pol} \sigma e : lab$. If (A2) is of the form $\Gamma_1 \vdash_{app} v : t_x$, then we first use Proposition 9 before proceeding.

Case (K-LABT): Induction hypothesis on the first premise, and then similar to (K-SLAB) on the second premise.

Case (K-FUN): Induction hypothesis on each premise, using Proposition 8 to satisfy the well-formedness requirements.

Case (K-ALL): Induction hypothesis. □

Corollary 13 (Contraction of assumptions). *For well-formed Γ_1 and $\Gamma_1, e_1 \succ e_2$, if all of the following are true*

(A1) $\Gamma_1, \overrightarrow{x} : lab, e_1 \succ e_2, \Gamma_2 \vdash_c e : t$

(A2) $e_1 \succ e_2 : \sigma$

(A3) $dom(\sigma) = \overrightarrow{x}$

Then, $\Gamma_1, \sigma\Gamma_2 \vdash_c \sigma e : \sigma' t$, where $\sigma' \in \{\emptyset, \sigma\}$.

Proof. If (A1) does not contain a sub-derivation with an application of (T-CONV) using (TE-REFINE), then the assumption $e_1 \succ e_2$ is redundant and we conclude with $\sigma' = \emptyset$.

If (A1) does contain an application of (TE-REFINE) then, we can use the substitution lemma to establish our result. By Lemma 11, we have that (T1) $\Gamma_1, \sigma e_1 \succ \sigma e_2, \sigma\Gamma_2 \vdash_c \sigma e : \sigma t$. However, by construction of pattern matching and by assumption (A2), we have $\sigma e_1 = \sigma e_2$. Thus, every relevant application of (TE-REFINE) in (T1) that concludes with $T \cdot \sigma(e_1) \cong T \cdot \sigma(e_2)$ can be replaced by an application of (TE-ID) in the result $\Gamma_1, \sigma\Gamma_2 \vdash_c \sigma e : \sigma t$. □

Lemma 14 (Type substitution). *Given well-formed $\Gamma_1, \alpha, \Gamma_2$ well-formed. If all of the following conditions are true:*

(A1) $\Gamma_1, \alpha, \Gamma_2 \vdash_c e : t$

(A2) $\Gamma_1 \vdash t'$

(A3) $\sigma = \alpha \mapsto t'$

Then,

$$\Gamma_1, \sigma\Gamma_2 \vdash_c \sigma e : \sigma t$$

or

$$\Gamma_1, \sigma\Gamma_2 \vdash_c \sigma t$$

Proof. By mutual induction, with the proposition $\Gamma_1, \alpha, \Gamma_2 \vdash t \implies \Gamma_1, \sigma\Gamma_2 \vdash_c \sigma t$, on the structure of (A1).

Case (T-INT): Trivial.

Case (T-VAR): If $x{:}t \in \Gamma_1$, then by well-formedness, α is not free in t and $\sigma(t) = t$. If $x{:}t \in \Gamma_2$, then $x : \sigma(t) \in \sigma(\Gamma_2)$ and we have the conclusion using (T-VAR).

266

Case (T-FIX): The second premise is of the form

$$\Gamma_1, \alpha, \Gamma_2, f{:}t \vdash_c v : t$$

Applying the induction hypothesis to this premise, we have

$$\Gamma_1, \sigma(\Gamma_2), f : \sigma(t) \vdash_c \sigma(v) : \sigma(t)$$

Which suffices for the conclusion.

Case (T-TAB): We have

$$\frac{\Gamma_1, \alpha, \Gamma_2, \beta \vdash_c e : t}{\Gamma_1, \alpha, \Gamma_2 \vdash_c \backslash\!\backslash\beta.e : \forall\beta.t}$$

with $\alpha \neq \beta$ from α-renaming. Thus, by the induction hypothesis we have

$$\Gamma_1, \sigma(\Gamma_2), \beta \vdash_c \sigma(e) : \sigma(t)$$

Case (T-TAP):

$$\frac{\Gamma_1, \alpha, \Gamma_2 \vdash t \qquad \Gamma_1, \alpha, \Gamma_2 \vdash_c e : \forall\beta.t'}{\Gamma_1, \alpha, \Gamma_2 \vdash_c e\,[t] : (\beta \mapsto t)t'}$$

By the mutual induction hypothesis, we have

$$\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(t)$$

and,

$$\Gamma_1, \sigma(\Gamma_2) \vdash_c \sigma(e) : \forall\beta.\sigma(t')$$

Together, this is sufficient to establish the goal.

Case (T-ABS):

Using the induction hypothesis (with the right side of the disjunct) on the first premise of (A1) we have

$$\Gamma_1, \sigma(\Gamma_2) \vdash \sigma(t_1)$$

and using IH on the second premise, we have

$$\Gamma_1, \sigma(\Gamma_2) \vdash_c \sigma(e) : \sigma(t_2)$$

Which is sufficient to conclude

$$\Gamma_1, \sigma(\Gamma_2) \vdash_c \backslash x : \sigma(t_1).\sigma(e) : \sigma((x{:}t_1) \to t_2)$$

Case (T-APP): Induction hypothesis to each premise.

Case (T-LAB), (T-HIDE), (T-SHOW): Induction hypothesis.

Case (T-MATCH): Induction hypothesis to each premise.

Case (T-UNLAB), (T-RELAB): Induction hypothesis.

Case (T-POL): Induction hypothesis. (Note, the statement of the Lemma does not impose any restrictions on the color of the derivation.)

Case (T-CONV): Induction hypothesis on the first premise. We must establish

$$\Gamma_1, \alpha, \Gamma_2 \vdash t \cong t' \Rightarrow \Gamma_1, \sigma(\Gamma_2) \vdash \sigma(t) \cong \sigma(t')$$

For (TE-REDUCE), we can establish
$\Gamma_1, \sigma(\Gamma_2) \vdash_c \sigma'(\sigma(e)) : lab$ by the induction hypothesis and $\sigma'(\sigma(e)) \stackrel{c}{\leadsto}^* \sigma'(\sigma(e'))$ since, by erasure, type substitution does not affect reduction.

For (TE-REFINE), we proceed similarly to (T-VAR) by cases on whether $e \succ e'$ appears in Γ_1 or Γ_2.

We now proceed to the cases of $\Gamma_1, \alpha, \Gamma_2 \vdash t$, the mutual induction proposition.

Case (K-INT): Trivial.

Case (K-TVAR): By assumption (A2) if $\sigma(\alpha) = t$; otherwise $\beta \in \Gamma_1, \sigma(\Gamma_2)$.

Case (K-LAB): Trivial.

Case (K-SLAB): Mutual induction hypothesis.

Case (K-LABT): Mutual induction hypothesis for the second premise.

Case (K-FUN): Induction hypothesis.

Case (K-UNIV): Induction hypothesis, with α-renaming as needed.

$\square$

Proposition 15 (Inversion of type abstractions). *Given Γ well formed, and $\Gamma \vdash_c \wedge\alpha.e : \forall\alpha.t_e$. Then, $\Gamma, \alpha \vdash_c e : t_e$*

Proposition 16 (Inversion of abstractions). *Given Γ well formed, and $\Gamma \vdash_c \lambda x{:}t.e : (x{:}t) \rightarrow t_e$. Then, $\Gamma, x{:}t \vdash_c e : t_e$.*

Theorem 17 (Preservation). *Given*

$$(A1) \quad \cdot \vdash_c e : t \qquad and \qquad (A2) \; e \stackrel{c}{\leadsto} e'$$

Then, $\Gamma \vdash_c e' : t$.

Proof. By induction on the structure of the derivation (A1).

Case (T-INT): n is a value.

Case (T-VAR): x is not a closed term.

Case (T-FIX): Inversion of (A2) gives an application of (E-FIX). By applying the substitution Lemma 11 to the second premise of (T-FIX), using (A1) as an assumption, and taking $\sigma = (f \mapsto \textbf{fix } f.v)$. From the first premise of (T-FIX) we have that $f \notin FV(t)$. Thus, $\sigma(t) = t$ and we have the desired result.

Case (T-TAB): $\wedge\alpha.e$ is a value.

Case (T-TAP): We have $e = e_1[t]$. Inversion of (A2) gives either an application of (E-CTX) and $e' = e_1'[t]$. Then, by the induction hypothesis, we are done. If $e = v[t]$, then (A2) is an application of (E-TAP) with conclusion $(\alpha \mapsto t)t'$ and $v = \wedge\alpha.$e'. From

Proposition 15 we get $\Gamma, \alpha \vdash_c e' : t'$ to which we can apply the type substitution lemma, Lemma 14 for the conclusion.

Case (T-ABS): $\lambda x{:}t.e$ is a value.

Case (T-APP): We have $e = e_1\, e_2$. By inversion of (A2) we get either (E-CTX) if either e_i is an expression, and we can conclude with the induction hypothesis. Otherwise, both are values and we have an application of (E-APP), with $e_1 = \lambda x{:}t.e$ with type $(x{:}t) \rightarrow t'$. From Proposition 16, we have $\Gamma, x{:}t \vdash_c e : t'$. From the substitution lemma, Lemma 11, we can establish $\Gamma \vdash_c (x \mapsto e_2)e : (x \mapsto e_2)t'$, as required.

Case (T-LAB), (T-HIDE): Immediate, from the induction hypothesis.

Case (T-SHOW): Using the induction hypothesis in the premise, it is easy to establish (T1) $\Gamma \vdash_c e' : lab \sim e'$, but we are required to conclude with the type $lab \sim e$. However, by (A2) we have $e \overset{c}{\rightsquigarrow} e'$, and $FV(e) = \emptyset$. Thus, to establish the result, we conclude with (T-CONV), with (T-1) in the first premise, and (TE-SYM), followed by (TE-REDUCE) and (A2) for the second premise.

Case (T-MATCH): By inversion, we get (A2) an instance of (E-CTX) or (E-MATCH). The former is straightforward from the induction hypothesis. In the latter case, the third premise of (A2) gives us (A2.1) $v \succ p_j : \sigma_j$ and from the first premise of (A1) we get (A1.1) $\Gamma \vdash_c v : lab$; by (T-LAB) all sub-terms of v must also be values of type lab. Thus, for each $x \in FV(p_j)$, $\Gamma \vdash_c \sigma_j(x) : lab$ and from the last premise of (A1) we have (T1) $\Gamma, \overrightarrow{x}{:}lab, v \succ p_j \vdash_c e_j : t$. From the contraction of assumptions, Corollary 13, taking $\Gamma_2 = \cdot$, we get $\Gamma \vdash_c \sigma_j(e_j) : \sigma_j(t_j)$.

 Finally, by the second premise of (A1) $\Gamma \vdash_c t$ and noting that $\overrightarrow{x_j} \notin \mathrm{dom}(\Gamma)$, and since $\mathrm{dom}(\sigma_j) = \overrightarrow{x_j}$, we have that $\sigma_j(t) = t$. Thus, from (T2), we have $\Gamma \vdash_c \sigma_j(e_j) : t$, which is our goal.

Case (T-UNLAB): Induction hypothesis.

Case (T-RELAB): Induction hypothesis on the first premise. The second premise is unchanged since the label expression e' is not reduced at runtime.

Case (T-POL): $e = (\!| e' |\!)$. Inversion of (A2) gives one of several cases

Sub-case (E-POL): e' is not a value and we apply the induction hypothesis to the premise of (A1) and the premise of (E-POL), which is sufficient to apply (T-POL) for the result.

Sub-case (E-BLAB): (A1) is (T-POL) with (T-LAB), (T-HIDE) or (T-SHOW) in the premise. In each case, we must show that $\Gamma \vdash_{pol} C(\overrightarrow{u}) : lab \Rightarrow \Gamma \vdash_{app} C(\overrightarrow{u}) : lab$, which is straightforward by induction since each sub-term is a pre-value of the form $C'(\overrightarrow{u'})$, with the base case being $\Gamma \vdash_{app} C : lab$.

Sub-case (E-BABS): We have

$$(A1)\frac{\Gamma \vdash_{pol} \lambda x : t_1.e : (x{:}t_1) \rightarrow t_2}{\Gamma \vdash_c (\!| \lambda x : t_1.e |\!) : (x{:}t_1) \rightarrow t_2}$$

To conclude, we must show

```
policy login(user:string, pw:string) =
            let token = match checkpw user pw with
                            USER(k) => USER(k)
                            _ => FAILED in
                (token, {token}0)

let member(u:lab, a:lab) =
            match a with
                ACL(u, i) => TRUE
                ACL(j, tl) => member u tl
                _ => FALSE

policy access<k,α>(u:lab ~ USER(k), cap:int{u}, acl:lab, data:α{acl}) =
            match member u acl with
                TRUE => {○}data
                _ => halt #access denied
```

Figure A.1: Enforcing a simple access control policy

$$(G) \frac{\Gamma, x{:}t_1 \vdash_c (\!|e|\!) : t_2}{\Gamma \vdash_c \lambda x{:}t_1.(\!|e|\!) : (x{:}t_1) \to t_2}$$

From the inversion lemma, Proposition 16, applied to the first premise of (A1), we get (A1.1.1) $\Gamma, x{:}t_1 \vdash_{pol} e : t_2$. For the conclusion, for the first premise of (G) we apply (T-POL) with (A1.1.1) in the premise.

Sub-case (E-BTAB): Similar to the previous case, using Proposition 15.

Sub-case (E-BINT): Apply (T-INT).

Sub-case (E-NEST): (A1) is of the form $\Gamma \vdash_{pol} (\!|e|\!) : t$ with (A1.1) $\Gamma \vdash_{pol} e : t$ in the premise. (A1.1) is exactly the desired conclusion.

Case (T-CONV): Induction hypothesis on the first premise; second premise is unchanged. □

A.2 Correctness of the Access Control Policy

Figure A.1 reproduces the access control policy from Section 2.1.1. In this section we prove the correctness of this policy with respect to the non-observability condition.

Definition 18 (Similarity up to l). *Expressions e and e', identified up to α-renaming, are similar up to label l according to the relation $e \sim_l e'$, defined in Figure A.2.*

Definition 19 (Bisimulation). *Expressions e_1 and e_2 are bisimilar at label l, written $e_1 \approx_l e_2$, if, and only if, $e_1 \sim_l e_2$ and for $\{i,j\} = \{1,2\}$, $e_i \overset{c}{\rightsquigarrow} e_i' \Rightarrow e_j \overset{c}{\rightsquigarrow} e_j'$ and $e_1' \approx_l e_2'$.*

$$e \sim_l e \qquad \{l\}e \sim_l \{l\}e' \qquad \frac{e \sim_l e'}{\{l'\}e \sim_l \{l'\}e'} \qquad \frac{e \sim_l e'}{\lambda x{:}t.e \sim_l \lambda x{:}t.e'}$$

$$\frac{i = 1,2 \quad e_i \sim_l e_i'}{e_1 e_2 \sim_l e_1' e_2'} \qquad \frac{v \sim_l v'}{\mathbf{fix}\ f{:}t.v \sim_l \mathbf{fix}\ f{:}t.v'} \qquad \frac{e \sim_l e'}{\bigwedge\alpha.e \sim_l \bigwedge\alpha.e'}$$

$$\frac{e \sim_l e' \quad t \sim_l t'}{e[t] \sim_l e[t']} \qquad \frac{\forall i.e_i \sim_l e_i'}{C(\overrightarrow{e}) \sim_l C(\overrightarrow{e'})} \qquad \frac{e \sim_l e' \quad e_i \sim_l f_i \quad p_i \sim_l q_i}{\mathbf{match}\ e\ \mathbf{with}\ p_i \to e_i \sim_l \mathbf{match}\ e'\ \mathbf{with}\ q_i \to f_i}$$

$$\frac{e \sim_l e'}{(\!|e|\!) \sim_l (\!|e'|\!)} \qquad \frac{e \sim_l e'}{T \cdot e \sim_l T \cdot e'} \qquad \frac{\mathrm{dom}(\sigma_1) = \mathrm{dom}(\sigma_2) \quad \forall x.\sigma_1(x) \sim_l \sigma_2(x)}{\sigma_1 \sim_l \sigma_2}$$

Figure A.2: Similarity of expressions under the access control policy

Lemma 20 (Similarity under substitution). *Given substitutions $\sigma_1 \sim_l \sigma_2$, then $\sigma_1(e) \sim_l \sigma_2(e)$.*

Proof. By construction of $e \sim_l e'$ and definition of substitution. $\qquad\square$

Theorem 21 (Non-observability). *Given two label constants acl and user, and for*

(A1) $i = 1,2,\ \cdot \vdash_{app} v_i : t\{acl\}$

(A2) A $(\!|\cdot|\!)$*-free expression e such* $a : t_a, m : t_m, cap : \mathbf{unit}\{\mathsf{user}\}, x : t\{\mathsf{acl}\} \vdash_{app} e : t_e$

(A3) *Type-respecting substitutions*

$$\sigma_i = (a \mapsto \mathsf{access}, m \mapsto \mathsf{member}, cap \mapsto (\!|\{\mathsf{user}\}()|\!), x \mapsto v_i)$$

(A4) $\mathsf{member\ user\ acl} \overset{c_*}{\leadsto} \mathsf{False}$

Then, $\sigma_1(e) \sim_{\mathsf{acl}} \sigma_2(e)$; and

$$\sigma_1(e) \overset{c}{\leadsto} \sigma_1(e') \iff \sigma_2(e) \overset{c}{\leadsto} \sigma_2(e') \wedge \sigma_1(e') \approx_{\mathsf{acl}} \sigma_2(e')$$

Proof. By induction on the structure of the typing derivation (A2).

Case (T-UNIT): Trivial, since $() \sim_{\mathsf{acl}} ()$ and $()$ is a value.

Case (T-VAR): The interesting case is when (A1) is of the form $\Gamma \vdash_{app} x : t\{acl\}$. In this case, we have $\sigma_1(x) = v_1$ and $\sigma_2(x) = v_2$. By inverting assumption (A2), we have $v_i = \{acl\}v_i'$, which, by definition, gives us $v_1 \sim_{\mathsf{acl}} v_2$. Since v_i is irreducible, we get $v_1 \approx_{\mathsf{acl}} v_2$. In any other case, we get $\sigma_1(x) = \sigma_2(x)$.

Case (T-FIX): We have

$$\mathbf{fix}\ f{:}\sigma_1(t).\sigma_1(v) \overset{c}{\leadsto} (f \mapsto \mathbf{fix}\ f{:}\sigma_1(t).\sigma_1(v))\sigma_1(v) = \sigma_1(f \mapsto \mathbf{fix}\ f{:}t.v)v$$

and

$$\textbf{fix } f{:}\sigma_2(t).\sigma_2(v) \overset{c}{\leadsto} (f \mapsto \textbf{fix } f{:}\sigma_2 u(t).\sigma_2(v))\sigma_2(v) = \sigma_2(f \mapsto \textbf{fix } f{:}t.v)v$$

and since $\sigma_i(f \mapsto \textbf{fix } f{:}t.v)v$ is a value, we have

$$\sigma_1(e) \sim_{\mathsf{acl}} \sigma_2(e) \Rightarrow \sigma_1(e) \approx_{\mathsf{acl}} \sigma_2(e)$$

Case T-TAB: We have, from (A1)

$$\sigma_1(\Lambda\alpha.e) = v_1' \sim_{\mathsf{acl}} v_2' = \sigma_2(\Lambda\alpha.e)$$

, and since both are values bisimilarity is as in (T-FIX).

Case (T-TAP): We have $\Gamma \vdash_c e[t]$ and $\sigma_i(/\alpha.e[t]) \overset{c}{\leadsto} \sigma_i((\alpha \mapsto t)e)$ We have $\sigma_1((\alpha \mapsto t)e) \sim_{\mathsf{acl}} \sigma_2((\alpha \mapsto t)e)$ from Lemma 20. But, from the type-substitution lemma (Lemma 14) we have

$$a{:}t_a, m{:}t_m, cap{:}\textbf{unit}\{\textsf{user}\}, x{:}t\{\textsf{acl}\} \vdash_{app} (\alpha \mapsto t)e : t_e$$

Thus, for bisimilarity, we use the induction hypothesis applied to this last judgment.

Case (T-ABS): Similar to (T-TAB).

Case (T-APP): The interesting case is when $\sigma_i(e) = v_i v_i'$. In other cases, we use the induction hypothesis and (E-CTX).

Sub-case ($v_i = \lambda x{:}t.e$) and e is $([\cdot])$-free:

From Proposition 16, we have (A2.1.1) $\Gamma, x{:}t \vdash_{app} e : t'$. To conclude, we must show,

$$(\sigma_1, (x \mapsto \sigma_1(v')))e \approx_{\mathsf{acl}} (\sigma_2, (x \mapsto \sigma_2(v')))e$$

But, by hypothesis, we have $\sigma_1 \sim_{\mathsf{acl}} \sigma_2$ and by the induction hypothesis we have $\sigma_1(v') \sim_{\mathsf{acl}} \sigma_2(v')$. Thus, we have

$$(\sigma_1, (x \mapsto \sigma_1(v')))e \sim_{\mathsf{acl}} (\sigma_2, (x \mapsto \sigma_2(v')))e$$

immediately, from Lemma 20. To conclude with bisimilarity, we use

$$(\sigma_i, (x \mapsto \sigma_i(v')))e = \sigma_i((x \mapsto v')e)$$

and we use (A2.1.1) and the substitution lemma, Lemma 11 to establish

$$a{:}t_a, m{:}t_m, cap{:}\textbf{unit}\{\textsf{user}\}, x{:}t\{\textsf{acl}\} \vdash_{app} (x \mapsto v')e : (x \mapsto v')t_e$$

Finally, we apply the induction hypothesis to this last judgment to get bisimilarity.

Sub-case ($v_i = \lambda x{:}t.([e])$) and e is $([\cdot])$-free:

In this case, we are unable to use the induction hypothesis to establish bisimilarity since,

$$\frac{(A2.1.1)\Gamma, x : t \vdash_{pol} e : t'}{\cdots}$$
$$\frac{}{\Gamma \vdash_{app} (\lambda x{:}t.([e]))v'}$$

and the hypothesis only applies to *app*-context judgments. However, our assumption was that e is $([])$-free. Thus, since we are assuming that $\sigma_i(e)$ is of the form $\lambda x{:}t.([e'])v'$, $([e'])$ must result from an application of access : t_a, where $(a \mapsto \mathsf{access}) \in \sigma_i$. We proceed by cases on the syntactic form of $\lambda x{:}t.([e'])v'$

Sub-case $(\lambda u : lab \sim \mathsf{user}.([\lambda cap{:}t.e]))$: After a step of reduction, we get $(u \mapsto v'_i)([\lambda cap{:}t.e])$ and since access is a closed term $\sigma_i([\lambda cap{:}t.e]) = ([\lambda cap{:}t.e])$. Thus, by Lemma 20, we have

$$(u \mapsto v'_1)([\lambda cap{:}t.e]) \sim_{\mathsf{acl}} (u \mapsto v'_2)([\lambda cap{:}t.e])$$

To establish bisimilarity, each side reduces in one step to a value using (E-BABS)

$$\lambda cap{:}t.([(u \mapsto v'_1)e]) \sim_{\mathsf{acl}} \lambda cap{:}t.([(u \mapsto v'_2)e])$$

Sub-case $(\lambda cap{:}unit\{u\}.([e]))$: We must have that $u = user$ since, we have an application

$$e = (\lambda x : unit\{u\}.([e]))\sigma_i(v')$$

and, by assumption v' is $([])$-free. Thus, we have an application in which $e_i = \lambda x : unit\{user\}.([e])([\{user\}()])$ reduces, as in the previous subcase, to similar values.

Sub-case $(\lambda acl : lab.([e]))$: Similar to previous sub-case.

Sub-case $(\lambda x : t\{l\}.([e]))$: We must have $l = acl$, since we have an application

$$e = (\lambda x : t\{l\}.\sigma_i(([e]))\sigma_i(v')$$

and, by assumption v' is $([])$-free. Thus, we have an application in which $e_i = \lambda x : t\{acl\}.([e])v_i$ which reduces in one step to

```
match member user acl with
TRUE => {○}data
_ => halt
```

But, by assumption we have member user acl $\overset{c_*}{\leadsto}$ *FALSE*. Thus, in both cases the program halts, maintaining bisimilarity.

Case (T-LAB): If we have $\Gamma \vdash_{app} C(\overrightarrow{u}, e', \overrightarrow{e}) : lab\ C(\overrightarrow{u}, e', \overrightarrow{e})$, then we can reduce the ith sub-term using (E-CTX) and establish the result using the induction hypothesis. Otherwise, we have $\sigma_1(e) = C(\overrightarrow{u}) = \sigma_2(e)$, because, if $x \in FV(e) \Rightarrow v_i$ is a sub-term of $\sigma(e)$ which is *lab*-typed. This is impossible since by (A1) v_i has a labeled type and unlabelings are not permitted in *app*-context, and e is $([])$-free.

Case (T-HIDE), (T-SHOW): Induction hypothesis.

Case (T-MATCH): The interesting case is when the matched expression is in fact a value v; in all other cases we conclude using (E-CTX) and the induction hypothesis. However, by the first premise, we have $\Gamma \vdash \sigma_i(v) : lab$ and $\sigma_1(v) \sim_{\mathsf{acl}} \sigma_2(v)$, which at type *lab*

typename *Prov* α= (l:*lab* {Auditors} $*$ α\{\{$\circ$\}l\})

policy flatten$<\alpha>$ (x:*Prov* (*Prov* α)) =
    let l,inner = x **in**
    let m,a = inner **in**
    let lm = Union(\{$\circ$\}l, \{$\circ$\}m) **in**
      (\{Auditors\}lm, \{lm\}a)

policy apply$<\alpha,\beta>$ (lf:*Prov* ($\alpha \longrightarrow \beta$), mx:*Prov* α) =
    let l,f = lf **in**
    let m,x = mx **in**
    let y = (\{$\circ$\}f) (\{$\circ$\}x) **in**
    let lm = Union(\{$\circ$\}l, \{$\circ$\}m) **in**
      (\{Auditors\}lm, \{lm\}y)

Figure A.3: Enforcing a dynamic provenance-tracking policy

implies $\sigma_1(v) \equiv \sigma_2(v)$. Thus, both $\sigma_1(e)$ and $\sigma_2(e)$ reduce to the same pattern branch and we conclude with the induction hypothesis.

Case (T-UNLAB), (T-RELAB): Inapplicable, since by assumption (A2) is in *app-context*.

Case (T-POL): Inapplicable, since by assumption e is $(\!|)$-free.

Case (T-CONV): Induction hypothesis. $\square$

A.3   Dynamic Provenance Tracking

Figure A.3 reproduces the provenance policy from Section 2.2.2. Figures A.4 and A.5 define a logical relation parameterized by a label l in order to relate terms with similar provenance.

Lemma 22 (Substitution for type-shape relation). *Given a well-formed Γ such that*

(A1) $\Gamma \vdash t$

(A2) $\sigma_1 \approx_p \sigma_2 : \Gamma, \Gamma$

Then, $\sigma_1 t \approx \sigma_2 t$

Proof. Straightforward induction on the structure of (A1)—a substitution of free variables in t does not change the shape of t. $\square$

Lemma 23 (Substitution for logical relation). *Given a well-formed Γ such that*

(A1) $\Gamma \vdash_{app} e : t$

(A2) e is $(\!|\cdot|\!)$-*free*

274

$$[\![e]\!] \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Interpretation of labels as sets}$$

$$[\![C]\!] \stackrel{\text{def}}{=} \{C\} \qquad\qquad [\![\mathsf{Union}(l_1,l_2)]\!] \stackrel{\text{def}}{=} [\![l_1]\!] \cup [\![l_2]\!]$$

$$t \leq t' \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{Prefixing relation on types}$$

$$t \leq t \qquad\qquad\qquad \frac{t \leq t'}{t \leq t'\{e\}}$$

$$e \approx_p e' : t,t' \qquad\qquad e \text{ and } e' \text{ are related at types } t \text{ and } t', \text{ parameterized by a provenance color } p$$

$$\frac{\begin{array}{c} i \in \{1,2\} \quad \cdot \vdash_c v_i : t_i \quad t_i'\{e_i\} \leq t_i \quad t \leq t_i \quad e_i \stackrel{pol_*}{\rightsquigarrow} v_i^{lab} \\ p \in [\![v_1^{lab}]\!] \cap [\![v_2^{lab}]\!] \qquad \vee \qquad \mathsf{Auditors} \in [\![v_1^{lab}]\!] \cap [\![v_2^{lab}]\!] \end{array}}{v_1 \approx_p v_2 : t_1,t_2} \text{ (R-EQUIVC)}$$

$$n \approx_p n : \mathbf{int},\mathbf{int} \text{ (R-INT)} \qquad\qquad \frac{i \in \{1,2\} \qquad \cdot \vdash_c e_i : t_i \\ e_i \stackrel{c_*}{\rightsquigarrow} v_i \quad \Rightarrow \quad v_1 \approx_p v_2 : t_1,t_2}{e_1 \approx_p e_2 : t_1,t_2} \text{ (R-EXPR)}$$

$$\frac{\begin{array}{c} \cdot \vdash_c v : (x{:}t_1) \rightarrow t_2 \qquad\qquad \cdot \vdash_c v' : (x{:}t_1') \rightarrow t_2' \\ \forall v_1,v_1'. \ v_1 \approx_p v_1' : t_1,t_1' \quad \Rightarrow \quad v v_1 \approx_p v' v_1' : (x \mapsto v_1)t_2, (x \mapsto v_1')t_2' \end{array}}{v \approx_p v' : (x{:}t_1) \rightarrow t_2, (x{:}t_1') \rightarrow t_2'} \text{ (R-ABS)}$$

$$\frac{\begin{array}{c} i \in \{1,2\} \qquad\qquad\qquad\qquad \cdot \vdash_c v_i : \forall \alpha.t_i \\ \forall t_1',t_2'. \ t_1' \approx t_2' \quad \Rightarrow \quad v_1[t_1'] \approx_p v_2[t_2'] : (\alpha \mapsto t_1')t_1, (\alpha \mapsto t_2')t_2 \end{array}}{v_1 \approx_p v_2 : \forall \alpha.t_1, \forall \alpha.t_2} \text{ (R-UNIV)}$$

$$\frac{i \in \{1,2\} \qquad \cdot \vdash_c C(\overrightarrow{e^i}) : lab \qquad \forall j. e_j^1 \approx_p e_j^2 : lab,lab}{C(\overrightarrow{e^1}) \approx_p C(\overrightarrow{e^2}) : lab,lab} \text{ (R-LAB)}$$

$$\frac{e_1 \approx_p e_2 : lab,lab}{e_1 \approx_p e_2 : lab \sim e_1, lab \sim e_2} \text{ (R-LAB2)} \qquad \frac{e_1 \approx_p e_2 : t_1,t_2 \quad i \in \{1,2\} \quad \cdot \vdash t_i \cong t_i'}{e_1 \approx_p e_2 : t_1',t_2'} \text{ (R-CONV)}$$

$$\frac{i \in \{1,2\} \quad \cdot \vdash_c (\![v_i]\!) : t_i \quad v_1 \approx_p v_2 : t_1,t_2}{(\![v_1]\!) \approx_p (\![v_2]\!) : t_1,t_2} \text{ (R-BRAC)} \qquad \frac{v \approx_p v' : t,t'}{\{e\}v \approx_p \{e'\}v' : t\{e\}, t'\{e'\}} \text{ (R-RELAB)}$$

$$\frac{\begin{array}{c} \mathrm{dom}(\sigma) = \mathrm{dom}(\sigma') \quad \forall \alpha \in \mathrm{dom}(\sigma).\sigma(\alpha) \approx \sigma'(\alpha) \\ \forall x \in \mathrm{dom}(\sigma).\sigma(x) \approx_p \sigma'(x) : \Gamma(x), \Gamma'(x) \end{array}}{\sigma \approx_p \sigma' : \Gamma, \Gamma'} \text{ (R-SUBST)}$$

Figure A.4: A logical relation for dynamic provenance tracking (Part 1)

$$\frac{\cdot \vdash t}{t \approx t} \text{ (RT-ID)} \qquad \frac{i \in \{1,2\} \quad \cdot \vdash lab \sim e_i}{lab \sim e_1 \approx lab \sim e_2} \text{ (RT-LAB)}$$

$$\frac{i \in \{1,2\} \quad \cdot \vdash t_i\{e_i\} \quad t_1 \approx t_2}{t_1\{e_1\} \approx t_2\{e_2\}} \text{ (RT-LABELED)}$$

$$\frac{\begin{array}{ccc} i \in \{1,2\} & \cdot \vdash (x{:}t_i) \to t_i' & t_1 \approx t_2 \\ \forall v_1, v_2.\cdot \vdash_c v_i : t_i & \Rightarrow & (x \mapsto v_1)t_1' \approx (x \mapsto v_2)t_2' \end{array}}{(x{:}t_1) \to t_1' \approx (x{:}t_2) \to t_2'} \text{ (RT-FUN)}$$

$$\frac{\begin{array}{ccc} i \in \{1,2\} & & \cdot \vdash \forall \alpha.t_i \\ \forall t, t'.t \approx t' & \Rightarrow & (\alpha \mapsto t)t_1 \approx (\alpha \mapsto t')t_2 \end{array}}{\forall \alpha.t_1 \approx \forall \alpha.t_2} \text{ (RT-UNIV)}$$

Figure A.5: A logical relation for dynamic provenance tracking (Part 2)

(A3) $\sigma_1 \approx_p \sigma_2 : \Gamma, \Gamma$

(A4) $e_1 \succ e_2 \notin \Gamma$

Then, $\sigma_1 e \approx_p \sigma_2 e : \sigma_1 t, \sigma_2 t$

Proof. By induction on the structure of (A1).

Case (T-INT): Trivial, with (R-INT) and $\sigma_i(t) = t$.

Case (T-VAR): By assumption (A3) $\sigma_1 \approx_p \sigma_2 : \Gamma, \Gamma$ by (R-SUBST) with $\sigma_1(x) \approx_p \sigma_2(x) : \Gamma(x), \Gamma(x)$ in the premise, where $\Gamma(x) = t = \sigma_i(t)$.

Case (T-FIX): Fix can be handled using a standard labeled reduction as in Mitchell [85], section 8.3.4. Note that according to (R-EXPR), we are only concerned with terminating computations.

Case (T-TAB):

$$\frac{\Gamma, \alpha \vdash e : t}{\Gamma \vdash_{app} \bigwedge \alpha.e : \forall \alpha.t} \text{ (T-TAB)}$$

We use the induction hypothesis on the premise to show that

$$\sigma_1' e \approx_p \sigma_2' e : \sigma_1' t, \sigma_2' t$$

where $\sigma_i' = \sigma_i, \alpha \mapsto t_i'$ and $t_1' \approx t_2'$. Thus, by (A3) we have $\sigma_1' \approx_p \sigma_2' : (\Gamma, \alpha), (\Gamma, \alpha)$. This suffices to establish the conclusion via (R-UNIV).

Case (T-TAP):

$$\frac{\Gamma \vdash t \quad \Gamma \vdash_{app} e : \forall \alpha.t'}{\Gamma \vdash_{app} e\,[t] : t''} \text{ (T-TAP)}$$

From the second premise we can use the induction hypothesis to establish that

$$\sigma_1(e) \approx_p \sigma_2(e) : \sigma_1(\forall \alpha.t), \sigma_2(\forall \alpha.t)$$

via (R-UNIV). I.e.

$$\forall t_1, t_2.t_1 \approx t_2 \implies \sigma_1(e)[t_1] \approx_p \sigma_2(e)[t_2] : (\sigma_1, \alpha \mapsto t_1)t', (\sigma_2, \alpha \mapsto t_2)t'$$

But, from the first premise (T-TAP) and from Lemma 22 we have that $\sigma_1(t) \approx \sigma_2(t)$. Thus, we can conclude

$$\sigma_1(e)[\sigma_1 t] \approx_p \sigma_2(e)[\sigma_2 t] : \sigma_1 t'', \sigma_2 t''$$

as required.

Case (T-ABS):

$$\frac{\Gamma \vdash t_1 \qquad \Gamma, x : t_1 \vdash_c e : t_2}{\Gamma \vdash_{app} \backslash x : t_1.e : x : t_1 \to t_2} \text{ (T-ABS)}$$

Our goal is to establish $\sigma_1(\lambda x{:}t_1.e) \approx_p \sigma_2(\lambda x{:}t_1.e) : \sigma_1((x{:}t_1) \to t_2), \sigma_2((x{:}t_1) \to t_2)$ via (R-ABS).

To use (R-ABS), we must first show for $i \in \{1,2\}$,

(T1) $\cdot \vdash_{app} \sigma_i(\lambda x{:}t_1.e) : \sigma_i((x{:}t_1) \to t_2)$

But this follows directly from the substitution lemma, Lemma 11.
Next, we must show

(R3) $\begin{array}{l} \forall v_1, v_2.v_1 \approx_p v_2 : \sigma_1 t_1, \sigma_2 t_1 \implies \\ \sigma_1(\lambda x{:}t_1.e)v_1 \approx_p \sigma_2(\lambda x{:}t_1.e)v_2 : (\sigma_1, x \mapsto v_1)t_2, (\sigma_2, x \mapsto v_2)t_2 \end{array}$

But, from the induction hypothesis applied to (T1) we have $\sigma_1(\lambda x{:}t_1.e) \approx_p \sigma_2(\lambda x{:}t_1.e)$, via (R-ABS). (R3) follows from the premise of (R-ABS).

Case (T-APP):

$$\frac{\Gamma \vdash_c e_1 : (x{:}t_1) \to t_2 \qquad \Gamma \vdash_c e_2 : t_1}{\Gamma \vdash_c e_1 e_2 : t_2'} \text{ (T-APP)}$$

Our goal is to show, via (R-EXPR),

$$\sigma_1(e_1 e_2) \approx_p \sigma_2(e_1 e_2) : \sigma_1 t_2', \sigma_2 t_2'$$

By the induction hypothesis applied to the premises of (A1) we have

(R1) $\sigma_1(e_1) \approx_p \sigma_2(e_1) : \sigma_1((x{:}t_1) \to t_2), \sigma_2((x{:}t_1) \to t_2)$

(R2) $\sigma_1(e_2) \approx_p \sigma_2(e_2) : \sigma_1 t_1, \sigma_2 t_1$

If e_1 is not a value, then by inversion (R1) is an instance of (R-EXPR) and we have

$$\sigma_1 e_1 \overset{c_*}{\leadsto} v_1 \wedge \sigma_2 e_1 \overset{c_*}{\leadsto} v_2 \Rightarrow v_1 \approx_p v_2 : \sigma_1((x{:}t_1) \to t_2), \sigma_2((x{:}t_1) \to t_2)$$

If e_1 is a value we have the conclusion of the previous implication directly. Inverting this relation, an instance of (R-ABS), we can derive

(R1.1) $\forall v, v', v \approx_p v' : \sigma_1 t_1, \sigma_2 t_1 \Rightarrow (\sigma_1(e_1)v \overset{c}{\leadsto}{}^* v_1 \wedge \sigma_2(e_1)v' \overset{c}{\leadsto}{}^* v_2) \Rightarrow v_1 \approx_p v_2 : (\sigma_1, x \mapsto v)t_2, (\sigma_2, x \mapsto v')t_2$

Similarly, inverting (R2) we can conclude

(R2.1) $\sigma_1(e_2) \overset{c_*}{\leadsto} v_1 \wedge \sigma_2(e_2) \overset{c_*}{\leadsto} v_2 \Rightarrow v_1 \approx_p v_2 : \sigma_1 t_1, \sigma_2 t_1$

If either, $\sigma_i e_1$ or $\sigma_i e_2$ diverge then we can establish the result trivially using (R-EXPR) since the guard in the implication of the last premise is false.

If neither diverges, we use (R1.1) instantiating v and v' to v_1 and v_2, respectively, from (R2.1).

Case (T-LAB), (T-HIDE), (T-SHOW): Induction hypothesis on the premise and concluding with either (R-LAB2) or reusing the premise of (R-LAB2) to conclude with (R-LAB).

Case (T-MATCH):

$$\frac{\begin{array}{cccc} \Gamma \vdash_c e : lab & \Gamma \vdash t & p_n = x & \text{where } x \notin \text{dom}(\Gamma) \\ \overrightarrow{x_i} = FV(p_i) \setminus \text{dom}(\Gamma) & \Gamma, \overrightarrow{x_i} : lab \vdash_c p_i : lab & \Gamma, \overrightarrow{x_i} : lab, e \succ p_i \vdash_c e_i : t \end{array}}{\Gamma \vdash_c \mathbf{match}\ e\ \mathbf{with}\ p_1 \Rightarrow e_1 \dots p_n \Rightarrow e_n : t} \text{(T-MATCH)}$$

Applying the IH to the first premise we get that

- (R1) $\sigma_1(e) \approx_p \sigma_2(e) : lab, lab$

However, by inverting (R1), we find that it must be an instance of (R-EXPR) or (R-LAB). We are only concerned with the case in which both $\sigma_1(e)$ and $\sigma_2(e)$ reduce to a values v_1 and v_2 respectively, since otherwise $\sigma_1 \mathbf{match}... \approx_p \sigma_2 \mathbf{match}... : t, t$ vacuously, using (R-EXPR) and the substitution lemma. In this case (where they both converge), we can consider the last premise of (R-EXPR) to be an instance of (R-LAB). However, $v_1 \approx_p v_2 : lab$, via (R-LAB) only if $v_1 = v_2 = v$.

Thus, if $\mathbf{match}\ v_1... \overset{c}{\leadsto} \sigma^* \sigma_1(e_j)$, iff $\mathbf{match}\ v_2... \overset{c}{\leadsto} \sigma^* \sigma_2(e_j)$, where σ^* is the pattern-matching substitution.

But, from (A1) we have

$$\Gamma, \overrightarrow{x_i} : lab, v \succ p_j \vdash_c e_j : t$$

By Corollary 13 we have $\Gamma, \overrightarrow{x_i} : lab \vdash_c \sigma_j(e_j) : t$. Furthermore, since $v_1 = v_2$, we have

$$\sigma^* \approx_p \sigma^* : \overrightarrow{x_i} : lab, \overrightarrow{x_i} : lab$$

278

using (R-LAB) repeatedly, as necessary. Therefore,

$$\sigma_1, \sigma^* \approx_p \sigma_2, \sigma^* : \Gamma, \overrightarrow{x_i} : lab, \Gamma, \overrightarrow{x_i} : lab$$

.

Finally, using the induction hypothesis on e_j, σ_1, σ^*, σ_2, σ^* we have our conclusion.

Case (T-UNLAB), (T-RELAB): Impossible; the statement only applies to $\Gamma \vdash_{app} e : t$.

Case (T-POL): Impossible; the statement only applies to $(\!|\cdot|\!)$-free terms.

Case (T-CONV):

$$\frac{\Gamma \vdash_c e : t \qquad \Gamma \vdash t \cong t'}{\Gamma \vdash_c e : t'} \quad \text{(T-CONV)}$$

By the induction hypothesis applied to

$$\Gamma \vdash_{app} e : t$$

we have

$$\sigma_1(e) \approx_p \sigma_2(e) : \sigma_1 t, \sigma_2 t$$

By the second premise of (A1), we have that $t \cong t'$. But, by definition of (TE-REDUCE), $t \cong t'$ iff $\forall \sigma. \sigma(t) \overset{c}{\leadsto}{}^* \sigma(t')$. Thus, we establish the conclusion using (R-CONV).

$\square$

Theorem 24 (Dependency correctness). *Given all of the following:*

(A1) a $(\!|\cdot|\!)$-free expression e such that $a : t_a, f : t_f, x : Prov\ t \vdash_{app} e : t'$,

(A2) a type-respecting substitution $\sigma = (a \mapsto \mathsf{apply}, f \mapsto \mathsf{flatten})$.

(A3) $\vdash_{app} v_i : Prov\ t$ for $i = 1, 2$ and $v_1 \approx_p v_2 : Prov\ t, Prov\ t$

(A4) for $i \in \{1, 2\}$, $\sigma_i = \sigma, x \mapsto v_i$

Then, $(\sigma_1(e) \overset{app_*}{\leadsto} v_1' \quad \wedge \quad \sigma_2(e) \overset{app_*}{\leadsto} v_2') \Rightarrow v_1' \approx_p v_2' : \sigma_1 t', \sigma_2 t'$.

Proof. It suffices to show that $\mathsf{apply} \approx_p \mathsf{apply} : t_a, t_a$ and $\mathsf{flatten} \approx_p \mathsf{flatten} : t_f, t_f$. Then, using (A3) to establish $\sigma_1 \approx_p \sigma_2 : a : t_a, f : t_f, x : Prov\ t$ we can use Lemma 23 for the conclusion.

Case (APPLY): The interesting case of $\mathsf{apply} \approx_p \mathsf{apply} : t_a$ amounts to showing

$$\mathsf{apply}[t_1, t_2] v_1 \approx_p \mathsf{apply}[t_1, t_2] v_2 : t_a, t_a$$

where (A5) $v_1 \approx_p v_2 : Prov\ t_1 \to t_2, Prov\ t_1 \to t_2$. We write (A5) using the more convenient notation of dependent tuples.

(A5)
$$\frac{(\!|\{Auditors\}l|\!) \approx_p (\!|\{Auditors\}l'|\!) : lab\,\{Auditors\}, lab\,\{Auditors\} \qquad (\!|\{l\}f|\!) \approx_p (\!|\{l'\}g|\!) : (t_1 \to t_2)\{l\}, (t_1 \to t_2)\{l'\}}{((\!|\{Auditors\}l|\!), (\!|\{l\}f|\!)) \approx_p ((\!|\{Auditors\}l'|\!), (\!|\{l'\}g|\!)) : Prov\ (t_1 \to t_2), Prov\ (t_1 \to t_2)}$$

The first premise is an instance of (R-BRAC) followed by (R-EQUIVC) since we have a labeling with Auditors. We now proceed by cases on the structure of the second premise of (A5), (A5.2).

Inverting (A5.2), we find that it must be an instance of (R-EQUIVC) or (R-BRAC).

Sub-case (R-EQUIVC): In this case, we have from the last premise, that $c \in [\![l_1]\!] \cap [\![l_2]\!]$. Since $\mathsf{apply}[t_1,t_2]v_1$ and $\mathsf{apply}[t_1,t_2]v_2$ are expressions with type $Prov\ t_1 \rightarrow Prov\ t_2$, we must establish the relation using (R-ABS). However, from inspection of apply, we find that whenever $\mathsf{apply}[t_1,t_2]v_i x$ terminates, it does so with a value of the form $(l_i, v_i^*) : Prov\ t_2$, i.e. $v_i^* : t_2\{l_i\}$. Since we have already established that $c \in [\![l_1]\!] \cap [\![l_2]\!]$, we can establish that $(l_1, v_1^*) \approx_p (l_2, v_2^*) : Prov\ t_2, Prov\ t_2$ using the same form as (A5).

Sub-case (R-BRAC):

$$\frac{\cdot \vdash_{app} ([\{l\}f]) : t \qquad \cdot \vdash_{app} ([\{l'\}g]) : t' \qquad \{l\}f \approx_p \{l'\}g : t, t'}{([\{l\}f]) \approx_p ([\{l'\}g]) : t, t'} \text{ (R-BRAC)}$$

Inverting the third premise, we find that it must be an instance of (R-RELAB), with (R-ABS) in the premise. That is, we have

$$(A6) \qquad \forall v_1', v_2'.\, v_1' \approx_p v_2' : t_1, t_1 \Rightarrow f v_1' \approx_p g v_2' : (x \mapsto v_1')t_2, (x \mapsto v_2')t_2$$

To establish the conclusion

$$\mathsf{apply}[t_1,t_2]v_1 v_1' \approx_p \mathsf{apply}[t_1,t_2]v_2 v_2' : t_2', t_2''$$

we notice that if both expressions do not diverge, the left and right side reduce to

$$(l_1, ([\{Union(l_1, l_1')\}f v_1'])) \qquad , \qquad (l_2, ([\{Union(l_2, l_2')\}g v_2']))$$

respectively. To establish the goal, we use reuse (A5) with (R-BRAC), (R-RELAB) and the conclusion of (A6) for the second premise.

Case (FLATTEN): Following an argument similar to apply and concluding with (R-EQUIVC), since Auditor-labeled terms are always considered equivalent in the relation. $\qquad \square$

A.4 Correctness of the Static Information-flow Policy

Figure A.6 reproduces the policy of Figure 2.9. To assist with the proof, we have annotated each relabeling operator with a unique index corresponding to its location in the source program.

Figure A.7 gives the semantics of FABLE^2, an extension of FABLE in the spirit of Core-ML^2 [104], Pottier and Simonet's technique for representing multiple program execution within the syntax of a single program. Appendix C elaborates upon the proof

```
policy lub(x:lab, y:lab) = match x,y with
    _, HIGH | HIGH, _ => HIGH
    | _, _ => LOW
policy join<α,l,m> (x:α{l}{m}) = ({lub l m}_1 {○}_2 {○}_3 x)
policy sub<α,l> (x:α{l}, m:lab) = ({lub l m}_4 {○}_5 x)
policy apply<α,β,l,m> (f:(α ⟶ β){l}, x:α) = {l}_6 ((({○}_7 f) x)
policy default<α> (l:lab, x:α) = {l}_8 x


let client (f:(int{HIGH} ⟶ int{HIGH}){LOW}, x:int{LOW}) =
  let x = (sub [int] x HIGH) in
    join [int] (apply [int{HIGH}][int{HIGH}] f x)
```

Figure A.6: Enforcing a static information flow policy

shown here, generalizing it to the FLAIR language, and applying it to the enforcement of static information flow in the presence of side effects.

Lemma 25 (Subject reduction for FABLE^2). *Given well-formed $\Gamma = \overrightarrow{x} : \overrightarrow{t}$, where $\overrightarrow{t}$ are the types of the policy π in Figure A.6, and a $(\!|)$-free FABLE^2 program e such that (A1) $\Gamma \vdash_c e : t$, and (A2) $e \overset{c}{\leadsto} e'$. Then, $\Gamma \vdash_c e' : t$.*

Proof. By induction on the structure of the typing derivation (A1). Most cases are identical to the proof of Theorem 17. We have to only pay special attention to the new lifting rules, (E-JOIN), (E-SUB), and (E-BAPP).

Case (T-RELAB): Inverting the reduction relation (A2), we now get (E-JOIN), (E-SUB), and (E-BAPP) in addition to all the previous cases.

Sub-case (E-JOIN): On the LHS we have $\{lub\, l\, m\}\{○\}\{○\}\{\!|v_1 \parallel v_2 |\!\}$, where the type of v_i is $t\overrightarrow{\{e\}}\{l\}\{m\}$. From the third premise of (T-BRACKET), we have that lub $\overrightarrow{e}$ $l\, m \overset{c}{\leadsto}^*$ High. This suffices to show on the RHS that lub $\overrightarrow{e}$ (lub l m) $\overset{c}{\leadsto}^*$ High, since lub is associative.

Sub-case (E-SUB): On the LHS we have $\{lub\, l\, m\}\{○\}\{\!|v_1 \parallel v_2|\!\}$, where the type of v_i is $t\overrightarrow{\{e\}}\{l\}$. From the third premise of (T-BRACKET), we have that lub $\overrightarrow{e}$ $l \overset{c}{\leadsto}^*$ High. This suffices to show on the RHS that lub $\overrightarrow{e}$ (lub l m) $\overset{c}{\leadsto}^*$ High, since lub is monotonic.

Sub-case (E-BAPP):

Case (E-BAPP1): The left-side of the reduction is typed using :

$$\frac{\dfrac{\Gamma \vdash_c \{○\}_7\{\!|v_f \parallel v_f'|\!\} : (x{:}t') \to t \ \ (A1.1.1) \qquad \dots}{\Gamma \vdash_c (\{○\}_7\{\!|v_f \parallel v_f'|\!\}) \, v_x : t \ \ (A1.1)}}{\Gamma \vdash_c \{l\}_6(\{○\}_7\{\!|v_f \parallel v_f'|\!\})v_x : t\{l\}}$$

where the type of v_f and v_f' is $(t \to t')\{l\}$. From the third premise of (A1.1.1), we can conclude that $l = $ High.

To type the RHS we use (T-BRACKET). We must show both the left and right sides of the bracket have the same type $t\{l\}$ and that the label l is High. To show that

281

$$
\begin{array}{llll}
e & ::= & \dots \mid \{\!\{e_1 \parallel e_2\}\!\} & \text{Bracketed expressions represent multiple executions} \\
v_c & ::= & \dots \mid \{\!\{v_c \parallel v_c\}\!\} \mid \{\circ\}\{\!\{v_c \parallel v_c\}\!\} & \text{Extensions to values} \\
E_c & ::= & \dots \mid \{\!\{\bullet \parallel e\}\!\} \mid \{\!\{e \parallel \bullet\}\!\} & \text{Evaluation contexts}
\end{array}
$$

$\Gamma \vdash_c e : t$ Additional type rules

$$
\frac{
\begin{array}{ccc}
t = t'\overrightarrow{\{e_l\}} & t' \neq t''\{e\} & \text{lub } \overrightarrow{e_l} \overset{c_*}{\rightsquigarrow} \text{HIGH} \\
\Gamma \vdash_c e_1 : t & \Gamma \vdash_c e_2 : t & \forall i.e_i \neq \{\!\{e_i' \parallel e_i''\}\!\}
\end{array}
}{
\Gamma \vdash_c \{\!\{e_1 \parallel e_2\}\!\} : t
} \quad \text{(T-BRACKET)}
$$

$e \overset{c}{\rightsquigarrow} e'$ Additional reduction rules

$$
\frac{i \in \{1,2\}}{\lfloor \{\!\{v_1 \parallel v_2\}\!\} \rfloor_i \equiv v_i} \text{ (PROJ-1)} \qquad
\frac{i \in \{1,2\}}{\lfloor \{e\}\{\!\{v_1 \parallel v_2\}\!\} \rfloor_i \equiv \{e\}v_i} \text{ (PROJ-2)}
$$

$$
\frac{i \in \{1,2\} \qquad v \notin \{\{\!\{v_1 \parallel v_2\}\!\}, \{e\}\{\!\{v_1 \parallel v_2\}\!\}\}}{\lfloor v \rfloor_i \equiv v} \text{ (PROJ-3)}
$$

$$
\{e\}_1\{\circ\}_2\{\circ\}_3\{\!\{v_1 \parallel v_2\}\!\} \overset{c}{\rightsquigarrow} \{\!\{\{e\}_1\{\circ\}_2\{\circ\}_3 v_1 \parallel \{e\}_1\{\circ\}_2\{\circ\}_3 v_2\}\!\} \text{ (E-JOIN)}
$$

$$
\{e\}_4\{\circ\}_5\{\!\{v_1 \parallel v_2\}\!\} \overset{c}{\rightsquigarrow} \{\!\{\{e\}_4\{\circ\}_5 v_1 \parallel \{e\}_4\{\circ\}_5 v_2\}\!\} \text{ (E-SUB)}
$$

$$
\{e\}_6((\{\circ\}_7\{\!\{v_f \parallel v_f'\}\!\})v_x) \overset{c}{\rightsquigarrow} \{\!\{\{e\}_6((\{\circ\}_7 v_f)\lfloor v_x \rfloor_1) \parallel \{e\}_6((\{\circ\}_7 v_f')\lfloor v_x \rfloor_2)\}\!\} \text{ (E-BAPP)}
$$

Figure A.7: Semantics of FABLE[2]

$(\{\circ\}v_f)\lfloor v_x\rfloor_1$ and $(\{\circ\}v_f')\lfloor v_x\rfloor_2$ have the same type, observe that both v_f and v_f' have the same type (from the fourth and fifth premises of (A1.1.1)). Similarly, both $\lfloor v_x\rfloor_1$ and $\lfloor v_x\rfloor_2$ have the same type. So, in the conclusion, we can construct an application of (T-APP) with the same type as in (A1.1) on each side of the bracket. Finally, the leading relabeling operation with $l = \mathsf{High}$ ensures that the RHS has a type of the form $t'\{High\}$, which satisfies the crucial third premise of (T-BRACKET). $\qquad\square$

Theorem 26 (Noninterference). *Given* $\overrightarrow{p} : \overrightarrow{t}, x : t\{HIGH\} \vdash_c e : t'\{LOW\}$, *where e is* ⟦⟧*-free and t' is not a labeled type; and, for $i = 1, 2$, $\cdot \vdash_c v_i : t\{HIGH\}$. Then, for type-respecting substitutions $\sigma_i = (\overrightarrow{p} \mapsto \pi, x \mapsto v_i)$, where π is the policy of Figure A.6,* $\sigma_1(e) \overset{c_*}{\leadsto} v_1' \wedge \sigma_2(e) \overset{c_*}{\leadsto} v_2' \Rightarrow v_1' = v_2'.$

Proof. Straightforward from the substitution lemma, Lemma 11, Lemma 25, and from construction, $\Gamma \vdash_c v : t$ where t not guarded at HIGH, implies $\lfloor v\rfloor_1 = \lfloor v\rfloor_2$. $\qquad\square$

A.5   Completeness of the Static Information-flow Policy

In this section, we show that the information flow policy of Figure A.6 is complete with respect to to the purely functional fragment of Pottier and Simonet's Core-ML [104]. Figure A.8 reproduces the syntax and the static semantics of a minimal functional fragment of Core-ML.

Definition 27 (Non-degeneracy of Core-ML typing). *A Core-ML type $(t_1 \to t_2)^l$ is non-degenerate if, and only if, $l \triangleleft t_2$ and both t_1 and t_2 are non-degenerate; unit is non-degenerate. A typing derivation $\mathscr{D} = \Gamma \vdash_{ML} e : t$ is non-degenerate if, and only if, for every sub-derivation $\mathscr{D}'$ with conclusion t', t' is non-degenerate.*

The non-degeneracy condition above assures that all function-typed expressions e are given types that permit the application of e. Note the third premise of (ML-APP) that requires the type of the function to be non-degenerate. So, while in programs such as $(\lambda x.0)e_1$, e_1 may be given a degenerate type since it is never applied. It is straightforward to transform a typing derivation for such programs into a non-degenerate derivation.

Figure A.9 shows a translation from Core-ML typing derivations $\mathscr{D}$ to FABLE programs e.

Theorem 28 (Completeness of static information flow). *Given e such that, $\mathscr{D} = \Gamma \vdash_{ML} e : t$ is non-degenerate; then $\overrightarrow{y} : \overrightarrow{t_\pi}, \llbracket\Gamma\rrbracket \vdash_c \llbracket\mathscr{D}\rrbracket : \llbracket t\rrbracket$, where $\overrightarrow{t_\pi}$ are the types of π, the policy of Figure A.6.*

Proof. By induction on the structure of the translation $\llbracket\mathscr{D}\rrbracket$.

Case (X-U): Trivial.

Case (X-V): Trivial.

Case (X-ABS): By the induction hypothesis we have

$$\overrightarrow{y} : \overrightarrow{t_\pi}, \llbracket\Gamma, f : (t_1 \to t_2)^l, x : t_1\rrbracket \vdash_c \llbracket\mathscr{D}\rrbracket : \llbracket t_2\rrbracket$$

$$
\begin{array}{rll}
\mathsf{e} & ::= & ()\mid x\mid \mathit{fixf}.\lambda x.\mathsf{e}\mid \mathsf{e_1e_2} \quad \textit{expressions}\\
\mathsf{t} & ::= & \mathsf{unit}\mid (\mathsf{t_1}\to \mathsf{t_2})^l \qquad\quad\; \textit{types}
\end{array}
$$

$$(\ominus\to\oplus)^{\oplus}\;\text{(Subtyping)}\qquad \textbf{Guards}\qquad l\lhd \mathsf{unit}\qquad \dfrac{\mathscr{L}\vdash l\sqsubseteq l'}{l\lhd(\mathsf{t}\to \mathsf{t'})^{l'}}$$

$$\Gamma\vdash_{ML}():\mathsf{unit}\ \text{(ML-UNIT)}\qquad\qquad \Gamma\vdash_{ML}x:\Gamma(x)\ \text{(ML-VAR)}$$

$$\dfrac{\Gamma[x:\mathsf{t}][f:(\mathsf{t}\to \mathsf{t'})^l]\vdash_{ML}\mathsf{e}:(\mathsf{t}\to \mathsf{t'})^l}{\Gamma\vdash_{ML}\mathit{fixf}.\lambda x.\mathsf{e}:(\mathsf{t}\to \mathsf{t'})^l}\ \text{(ML-ABS)}$$

$$\dfrac{\Gamma\vdash_{ML}\mathsf{e_1}:(\mathsf{t}\to \mathsf{t'})^l\quad \Gamma\vdash_{ML}\mathsf{e_2}:\mathsf{t}\quad l\lhd \mathsf{t'}}{\Gamma\vdash_{ML}\mathsf{e_1e_2}:\mathsf{t'}}\ \text{(ML-APP)}$$

$$\dfrac{\Gamma\vdash_{ML}\mathsf{e}:\mathsf{t'}\quad \mathsf{t'}\le \mathsf{t}}{\Gamma\vdash_{ML}\mathsf{e}:\mathsf{t}}\ \text{(ML-SUB)}\qquad t\le t\ \text{(SUB-ID)}\qquad \dfrac{t_1'\le t_1\quad t_2\le t_2'\quad l\sqsubseteq l'}{(t_1\to t_2)^l\le (t_1'\to t_2')^{l'}}\ \text{(SUB-FN)}$$

Figure A.8: Core-ML syntax and typing (Functional fragment)

To establish the conclusion we use, (T-FIX) with (T-POL) and (T-RELAB) followed by (T-ABS), with the induction hypothesis and Lemma 9 (sub-coloring of derivations) applicable in the premises of (T-ABS).

Case (X-APP1, X-APP2): By the induction hypothesis we have both

$$
\begin{array}{ll}
i. & \overrightarrow{y}{:}\overrightarrow{t_\pi},[\![\Gamma]\!]\vdash [\![\mathscr{D}_1]\!]:[\![(t_1\to t_2)^l]\!],\quad \text{and,}\\
ii. & \overrightarrow{y}{:}\overrightarrow{t_\pi},[\![\Gamma]\!]\vdash [\![\mathscr{D}_2]\!]:[\![t_1]\!]
\end{array}
$$

The type of apply is $\forall\alpha,\beta.\textbf{phantom}\ l.(f{:}(\alpha\to\beta)\{l\})\to(x{:}\alpha)\to\beta\{l\}$
Thus, we have the type of apply $\ldots[\![\mathscr{D}_1]\!][\![\mathscr{D}_2]\!]$ to be $[\![t_2^l]\!]$. In the case of (X-APP2) this is specifically $()\{[\![l]\!]^{lub}\}$ which is an acceptable translation of the Core-ML type unit. In the case of (X-APP1), this type is specifically $[\![t]\!]\{[\![l']\!]^{lub}\}\{[\![l]\!]^{lub}\}$ which is not yet an acceptable translation of t_2. Thus, we apply join $\ldots$ which has type $\forall\alpha.\textbf{phantom}\ l,m.x{:}\alpha\{l\}\{m\}\to \alpha\{lublm\}$ to obtain $[\![t]\!]\{\mathsf{lub}[\![l']\!]^{lub}[\![l]\!]^{lub}\}$. To conclude, we use the final premise of (ML-APP) which asserts that $l\lhd t_2$, which requires $l\sqsubseteq l'$. Thus, $\mathsf{lub}[\![l']\!]^{lub}[\![l]\!]^{lub}=[\![l']\!]$.

Case (CASE X-SUB): We proceed by induction on the structure of the subtyping derivation using the induction hypothesis to establish that $\overrightarrow{y}{:}\overrightarrow{t_\pi},[\![\Gamma]\!]\vdash [\![\mathscr{D}]\!]:[\![t]\!]$. That is, we wish to establish that given

$$
\begin{array}{ll}
\overrightarrow{y}{:}\overrightarrow{t_\pi},[\![\Gamma]\!]\vdash e:[\![t]\!],\qquad \text{then}\\
\overrightarrow{y}{:}\overrightarrow{t_\pi},[\![\Gamma]\!]\vdash [\![t\le t']\!]^{(e)}:[\![t']\!]
\end{array}
$$

The (SUB-ID) case is trivial. We examine first the type of e' in (SUB-FN). By assumption we have that the type of e is $(t_1\to t_2)\{e_l\}$. We have that $x:t_1'$ by ascription in the lambda

$\llbracket \mathsf{t} \rrbracket, \llbracket \Gamma \rrbracket$ Translation of Core-ML types and environment

$$\llbracket \mathsf{unit} \rrbracket \equiv \mathbf{unit} \qquad \frac{e_l \in \{\mathsf{LOW}, \mathsf{HIGH}\}}{\llbracket \mathsf{unit} \rrbracket \equiv \mathbf{unit}\{e_l\}}$$

$$\llbracket (\mathsf{t}_1 \to \mathsf{t}_2)^l \rrbracket \equiv (\llbracket \mathsf{t}_1 \rrbracket \to \llbracket \mathsf{t}_2 \rrbracket)\{\llbracket l \rrbracket^{lab}\} \qquad \llbracket x:\mathsf{t}, \Gamma \rrbracket^{env} \equiv x : \llbracket \mathsf{t} \rrbracket, \llbracket \Gamma \rrbracket^{env}$$

$\llbracket l \rrbracket^{lab}$ Translation of Core-ML labels to FABLE terms

$$\llbracket L \rrbracket^{lab} \equiv \mathsf{LOW} \qquad \llbracket H \rrbracket^{lab} \equiv \mathsf{HIGH}$$

$\llbracket \mathscr{D} \rrbracket$ Translation of derivations $\mathscr{D}$ to FABLE expressions

$$\llbracket \Gamma \vdash_{ML} () : \mathsf{unit} \rrbracket \equiv () \ (\text{X-U}) \qquad \llbracket \Gamma \vdash_{ML} x : \Gamma(x) \rrbracket \equiv x \ (\text{X-V})$$

$$\llbracket \frac{\mathscr{D}}{\Gamma \vdash_{ML} fix f.\lambda x.e : (\mathsf{t}_1 \to \mathsf{t}_2)^l} \rrbracket \equiv \mathbf{fix}\ f.(\{\llbracket l \rrbracket^{lab}\}\lambda x{:}\llbracket \mathsf{t}_1 \rrbracket.\llbracket \mathscr{D} \rrbracket)\ (\text{X-ABS})$$

$$\llbracket \frac{\mathscr{D}_1 = \Gamma \vdash_{ML} e_1 : (\mathsf{t}_1 \to \mathsf{t}_2)^l \qquad \mathscr{D}_2 \qquad l \lhd \mathsf{t}_2}{\Gamma \vdash_{ML} e_1 e_2 : \mathsf{t}_2} \rrbracket [(\mathsf{t}_2 = \mathsf{t}')] \equiv$$
$$\mathsf{join}\ [\mathsf{t}_2]\ (\mathsf{apply}\ [\llbracket \mathsf{t}_1 \rrbracket][\llbracket \mathsf{t}_2 \rrbracket] \llbracket \mathscr{D}_1 \rrbracket \llbracket \mathscr{D}_2 \rrbracket)\ (\text{X-APP1})$$

$$\llbracket \frac{\mathscr{D}_1 = \Gamma \vdash_{ML} e_1 : (\mathsf{t}_1 \to \mathsf{t}_2)^l \qquad \mathscr{D}_2 \qquad l \lhd \mathsf{t}_2}{\Gamma \vdash_{ML} e_1 e_2 : \mathsf{t}_2} \rrbracket [(\mathsf{t}_2 = \mathsf{unit})] \equiv$$
$$\mathsf{apply}\ [\llbracket \mathsf{t}_1 \rrbracket][\llbracket \mathsf{t}_2 \rrbracket] \llbracket \mathscr{D}_1 \rrbracket \llbracket \mathscr{D}_2 \rrbracket\ (\text{X-APP2})$$

$$\llbracket \frac{\mathscr{D} \qquad \mathsf{t} \leq \mathsf{t}'}{\Gamma \vdash_{ML} e : \mathsf{t}'} \rrbracket \equiv \llbracket \mathsf{t} \leq \mathsf{t}' \rrbracket^{(\llbracket \mathscr{D} \rrbracket)}\ (\text{X-SUB})$$

$\llbracket \mathsf{t} \leq \mathsf{t}' \rrbracket^{(e)} \equiv e'$ Subtyping a $\llbracket \mathsf{t} \rrbracket$-typed FABLE expression, e

$$\llbracket \mathsf{t} \leq \mathsf{t} \rrbracket^{(e)} \equiv e\ (\text{SUB-ID})$$

$$\llbracket \frac{\mathscr{D}_1 = \mathsf{t}'_1 \leq \mathsf{t}_1 \qquad \mathscr{D}_2 = \mathsf{t}_2 \leq \mathsf{t}'_2 \qquad l \sqsubseteq l'}{(\mathsf{t}_1 \to \mathsf{t}_2)^l \leq (\mathsf{t}'_1 \to \mathsf{t}'_2)^{l'}} \rrbracket^{(e)} \equiv \mathsf{default}\ [t'_1 \to t'_2]\ e_{l'}\ (\lambda x{:}t'_1.\llbracket \mathscr{D}_2 \rrbracket^{(e')})\ (\text{SUB-FN})$$

$$\text{where} \qquad \begin{aligned} &t_1 = \llbracket \mathsf{t}_1 \rrbracket, \quad t'_1 = \llbracket \mathsf{t}'_1 \rrbracket \\ &t_2 = \mathsf{t}^{l_2}, e_{l_2} = \llbracket l_2 \rrbracket^{lab} \\ &t_2 = \llbracket \mathsf{t}_2 \rrbracket = t\{e_{l_2}\}, \quad t'_2 = \llbracket \mathsf{t}'_2 \rrbracket \\ &e_l = \llbracket l \rrbracket^{lab}, e_{l'} = \llbracket l' \rrbracket^{lab}, \qquad \text{and,} \\ &e' = \mathsf{join}[t]\,(\mathsf{apply}\ [t_1]\ [t_2]\ e\ \llbracket \mathscr{D}_1 \rrbracket^{(x)}) \end{aligned}$$

Figure A.9: Translation from a Core-ML derivation $\mathscr{D}$ to FABLE

binding. Thus, by the inductive hypothesis we have that $\overrightarrow{y:t_\pi}, [\![\Gamma]\!], x : t_1' \vdash [\![\mathcal{D}_1]\!]^x : t_1$. Now, using the type for apply given in (Case X-APP1), we conclude that apply$\dots [\![\mathcal{D}_1]\!]^x$ has type $t_2\{e_l\}$. After the application of join we conclude that e' has type $t\{$lub e_{l_2} $e_l\}$. However, from the non-degeneracy assumption, we have $l \lhd t_2$ or $l_2 \sqsubseteq l$; thus, the type of e' is $t\{e_{l_2}\} = [\![t_2]\!]$. To type $\backslash x : t_1'.[\![\mathcal{D}_2]\!]^{(e')}$ we use the induction hypothesis to establish that $[\![\mathcal{D}_2]\!]^{(e')}$ has type t_2', to arrive at the type $t_1' \rightarrow t_2'$ using (T-ABS). Finally, the type of default is $\forall\alpha.(l{:}lab) \rightarrow \alpha \rightarrow \alpha\{l\}$, which is sufficient to establish the type of $(t_1' \rightarrow t_2')\{e_{l'}\} \equiv [\![(t_1' \rightarrow t_2')^{l'}]\!]$ for the translation, which is our goal. $\qquad\square$

B. Proofs of Theorems Related to λAIR

B.1 Soundness of λAIR

Definition 29 (Well-formed environment). *$\Gamma;A$ is well-formed if and only if*

$$
\begin{aligned}
&(i.) &&\textit{All names bound in } \Gamma \textit{ are distinct}\\
&(ii.) &&\Gamma = \Gamma_1, x:t, \Gamma_2 \;\Rightarrow\; FV(t) \subseteq dom(\Gamma_1)\\
&(iii.) &&A = A_1, x, A_2 \;\Rightarrow\; \Gamma = \Gamma_1, x:t, \Gamma_2
\end{aligned}
$$

Definition 30 (Type consistency of a signature). *A signature S and its model M are type consistent if and only if for each $\mathscr{D} \rightsquigarrow e \in \overrightarrow{E}$, where $B : \overrightarrow{E} \in M$, we have for $\Gamma = \alpha_1::N, \ldots, \alpha_n::N$:*

1. *$\Gamma; \cdot \vdash \llbracket B \rrbracket^{\mathscr{D}} : t; \varepsilon \iff \Gamma; \cdot \vdash e : t; \varepsilon$*

2. *For every $B \in dom(S)$, $B : \overrightarrow{E} \in M$.*

3. *For every $(T :: K) \in S$, $\vdash K$ ok where*

$$
\frac{}{\vdash k \ ok} \qquad \frac{\vdash K \ ok}{\vdash k \to K \ ok} \qquad \frac{\cdot \vdash t :: k \quad \vdash K \ ok}{\vdash t \to K \ ok}
$$

4. *For every $(B : t) \in S$, $\cdot \vdash t :: k$.*

5. *$\Gamma; \cdot \vdash_\varphi \llbracket B \rrbracket^{\mathscr{D}} : t; \varepsilon \wedge \varepsilon \neq \cdot \;\Rightarrow\; (\Gamma \vdash t :: \text{A} \vee \Gamma; \cdot \vdash_\varphi \llbracket B \rrbracket^{\mathscr{D}} : t; \cdot)$*

Theorem 31 (Progress). *Given $\Gamma = \alpha_1::N, \ldots, \alpha_n::N$ such that (A1) $\Gamma; \cdot \vdash_{\text{term}} e : t; \varepsilon$; and (A2) given an interpretation M such that M and S are type consistent, then either $\exists e'.M \vdash e \xrightarrow{l} e'$ or $\exists v.e = v$.*

Proof. By induction on the structure of (A1).

Case (T-BD): If e is a data constructor D, then it is a value and we are done. Otherwise $e = B$, and from Definition 30, we can satisfy the premise of (E-DELTA) and take a step to $\llbracket B \rrbracket$.

Case (T-X), (T-XA): By assumption, Γ only contains type names α_i; i.e., e is a closed term. So, these cases are impossible.

Case (T-NEW): If in new e, e is an expression, then by the induction hypothesis on the first premise we have $M \vdash e \xrightarrow{l} e'$. Then, by the syntactic form of the evaluation contexts $\mathscr{E}$ and by the congruence rule (E-CTX) we have the result. On the other hand, if e is a value v then new v is also a value.

Case (T-TAB): $\Lambda \alpha::k.e$ is a value.

Equations, models, and certificates

equation	E	$::=$	$\mathscr{D} \rightsquigarrow e$	model	M	$::=$	$B : \overrightarrow{E} \mid M, M$
eqn. domain	$\mathscr{D}$	$::=$	$v \mid t \mid \mathscr{D}, \mathscr{D} \mid \cdot$	certificates	e	$::=$	$\dots \mid [\![B]\!]^{\mathscr{D}}$

$\Gamma; A \vdash_{\varphi} e : t; \varepsilon$ $\qquad\qquad$ A φ-level expression e has type t and uses names ε

$$\frac{S \in \Gamma \quad e \in \{B, D\} \quad S(e) = t}{\Gamma; \cdot \vdash_{\varphi} e : t; \cdot} \text{ (T-BD)} \qquad \frac{\Gamma \vdash \Gamma(x) :: U}{\Gamma; \cdot \vdash_{\varphi} x : \Gamma(x); \cdot} \text{ (T-X)} \qquad \frac{}{\Gamma; x \vdash_{\varphi} x : \Gamma(x); \cdot} \text{ (T-XA)}$$

$$\frac{}{\Gamma; \cdot \vdash_{\text{type}} x : \Gamma(x); \cdot} \text{ (T-X-type)} \qquad \frac{\Gamma; A \vdash_{\text{type}} e : t; \varepsilon_1 \uplus \varepsilon}{\Gamma; A \vdash_{\text{type}} e : t; \varepsilon} \text{ (T-NC-type)} \qquad \frac{\Gamma \vdash t :: k \quad k \neq N}{\Gamma; \cdot \vdash_{\varphi} \bot : t; \cdot} \text{ (T-BOT)}$$

$$\frac{\Gamma; A \vdash_{\varphi} e : t; \varepsilon \quad \Gamma \vdash t :: U \quad \Gamma(\alpha) = N}{\Gamma; A \vdash_{\varphi} \text{new } e : \text{¡}t^{\alpha}; \alpha \uplus \varepsilon} \text{ (T-NEW)} \qquad \frac{\Gamma; A \vdash_{\varphi} e : t^{\alpha}; \varepsilon}{\Gamma; A \vdash_{\varphi} e : t^{\circ}; \varepsilon} \text{ (T-DROP)} \qquad \frac{\Gamma; A \vdash_{\varphi} e : t; \varepsilon \quad \varepsilon' \subseteq \text{dom}(\Gamma)}{\Gamma; A, A' \vdash_{\varphi} e : t; \varepsilon \uplus \varepsilon'} \text{ (T-WKN)}$$

$$\frac{\Gamma, \alpha :: k; A \vdash_{\varphi} e : t; \varepsilon \uplus \varepsilon' \quad \alpha \notin \varepsilon \quad \varepsilon' \in \{\cdot, \alpha\} \quad q = p(A, \varepsilon)}{\Gamma; A \vdash_{\varphi} \Lambda \alpha :: k.e : q(\forall \alpha :: k \xrightarrow{\varepsilon'} t); \varepsilon} \text{ (T-TAB)} \qquad \frac{\Gamma; A \vdash_{\varphi} e : q(\forall \alpha :: k \xrightarrow{\varepsilon'} t'); \varepsilon \quad \Gamma \vdash t :: k}{\Gamma; A \vdash_{\varphi} e [t] : [\alpha \mapsto t]t'; \varepsilon \uplus ([\alpha \mapsto t]\varepsilon')} \text{ (T-TAP)}$$

$$\frac{\Gamma \vdash t_x :: k \quad q = p(A, \varepsilon) \quad \Gamma, x : t_x; A, a(x, k) \vdash_{\varphi} e : t_e; \varepsilon}{\Gamma; A \vdash_{\varphi} \lambda x{:}t_x.e : q((x{:}t_x) \to t_e); \varepsilon} \text{ (T-ABS)} \qquad \frac{\Gamma; A \vdash_{\varphi} e : q((x{:}t') \to t); \varepsilon_1 \quad \Gamma; A' \vdash_{\varphi} e' : t'; \varepsilon_2}{\Gamma; A, A' \vdash_{\varphi} e\, e' : [x \mapsto e']t; \varepsilon_1 \uplus \varepsilon_2} \text{ (T-APP)}$$

$$\frac{\Gamma; A \vdash_{\varphi} e : t_e; \varepsilon \quad \cdot; \cdot \vdash t_i :: k_i \quad \forall i.a(x_i, k_i) \in A'' \quad \overrightarrow{x : t}; A'' \vdash e_{pat} : t_e; \cdot \quad \Gamma, \overrightarrow{x : t}; A', A'' \vdash_{\varphi} e' : t; \varepsilon' \quad \Gamma; A' \vdash_{\varphi} e'' : t; \varepsilon''}{\Gamma; A, A' \vdash_{\varphi} \text{case } e \text{ of } \overrightarrow{x{:}t}.e_{pat} : e' \text{ else } e'' : t; \varepsilon \uplus (\varepsilon' \cup \varepsilon'')} \text{ (T-CASE)}$$

$$\frac{\Gamma; A \vdash_{\varphi} e : q(t' \Rightarrow t); \varepsilon \quad \Gamma; A' \vdash_{\varphi} e' : t'; \varepsilon'}{\Gamma; A, A' \vdash_{\varphi} e\,(e') : t; \varepsilon \uplus \varepsilon'} \text{ (T-CAP)} \qquad \frac{\Gamma; A \vdash_{\varphi} [\![B]\!]^{\mathscr{D}} v : t; \varepsilon}{\Gamma; A \vdash_{\varphi} [\![B]\!]^{\mathscr{D}, v} : t; \varepsilon} \text{ (T-B1)}$$

$$\frac{\Gamma; A \vdash_{\varphi} [\![B]\!]^{\mathscr{D}} [t] : t'; \varepsilon}{\Gamma; A \vdash_{\varphi} [\![B]\!]^{\mathscr{D}, t} : t'; \varepsilon} \text{ (T-B2)} \qquad \frac{\Gamma; A \vdash_{\varphi} B : t; \varepsilon}{\Gamma; A \vdash_{\varphi} [\![B]\!] : t; \varepsilon} \text{ (T-B3)} \qquad \frac{\Gamma; A \vdash_{\varphi} e : t; \varepsilon \quad t \cong t'}{\Gamma; A \vdash_{\varphi} e : t'; \varepsilon} \text{ (T-CONV)}$$

where
$$a(x, \text{A}) = x \quad a(x, \text{U}) = \cdot$$
$$p(A, \varepsilon) = \text{¡} \quad p(\cdot, \cdot) = \cdot$$

Figure B.1: Static semantics of λAIR (Typing judgment)

$t \cong t'$ Congruence of types t and t' under reduction of type-level expressions.

Type contexts T ::= $\bullet \mid x{:}\bullet \to t \mid x{:}t \xrightarrow{\varepsilon} \bullet \mid \forall \alpha{::}k \to \bullet \mid \bullet \Rightarrow t \mid t \Rightarrow \bullet \mid q\,\bullet \mid \bullet\, t \mid t\, \bullet \mid \bullet^{\eta}$

$$t \cong t \quad \text{(TE-ID)} \qquad\qquad \frac{t \cong t'}{t' \cong t} \ \text{(TE-SYM)} \qquad\qquad \frac{t \cong t'}{T \cdot t \cong T \cdot t'} \ \text{(TE-CTX)}$$

$$\frac{\cdot;\cdot \vdash_{\mathsf{type}} e : t, \varepsilon \qquad M \vdash e \xrightarrow{l} e' \qquad \cdot;\cdot \vdash_{\mathsf{type}} e' : t, \varepsilon}{T \cdot e \cong T \cdot e'} \ \text{(TE-RED)}$$

$\Gamma \vdash t :: K$ A type t has kind K in environment Γ

$$\frac{\Gamma(\alpha) = k}{\Gamma \vdash \alpha :: k} \ \text{(K-A)} \qquad \frac{\Gamma \vdash t :: \mathtt{A} \quad \Gamma(\eta) = \mathtt{N} \ \vee\ \eta = \circ}{\Gamma \vdash t^{\eta} :: \mathtt{A}} \ \text{(K-N)} \qquad \frac{S(T) = K}{\Gamma \vdash T :: K} \ \text{(K-TC)}$$

$$\frac{\Gamma \vdash t :: \mathtt{U}}{\Gamma \vdash \mathop{\text{\textwidth}}\nolimits_{\!\!\!\text{¡}} t :: \mathtt{A}} \ \text{(K-AFN)} \qquad \frac{\Gamma \vdash t :: k \quad \Gamma, x : t \vdash t' :: k'}{\Gamma \vdash (x{:}t) \to t' :: \mathtt{U}} \ \text{(K-FUN)} \qquad \frac{\begin{array}{c}\Gamma \vdash t :: t' \to K \\ \Gamma; \cdot \vdash_{\mathsf{type}} e : t'; \cdot\end{array}}{\Gamma \vdash t\, e :: K} \ \text{(K-DEP)}$$

$$\frac{\Gamma' = \Gamma, \alpha{::}k \quad \Gamma' \vdash t :: k \quad \alpha' \in \varepsilon \Rightarrow \Gamma'(\alpha') = \mathtt{N}}{\Gamma \vdash \forall \alpha{::}k \xrightarrow{\varepsilon} t :: \mathtt{U}} \ \text{(K-UNIV)}$$

$$\frac{\Gamma \vdash t :: k \to K \quad \Gamma \vdash t' :: k}{\Gamma \vdash t\, t' :: K} \ \text{(K-TAP)} \qquad \frac{\Gamma \vdash t_1 :: \mathtt{U} \quad \Gamma \vdash t_2 :: \mathtt{U} \quad t_2 \in \{t \Rightarrow t', T\}}{\Gamma \vdash t_1 \Rightarrow t_2 :: \mathtt{U}} \ \text{(K-CON)}$$

Figure B.2: Static semantics of λ_{AIR} (Type equivalence and kinding judgment)

Values and evalutation contexts

$$\text{values} \quad v \quad ::= \quad D \mid \llbracket B \rrbracket^{\mathscr{D}} \mid \lambda x{:}t.e \mid \Lambda\alpha{::}k.e \mid v\,(v') \mid \text{new } v$$
$$\text{eval ctxt} \quad \mathscr{E} \quad ::= \quad \bullet \mid \bullet\, e \mid v\,\bullet \mid \bullet\,[t] \mid \bullet\,(e) \mid v\,(\bullet) \mid \text{case } \bullet \text{ of} \dots \mid \text{new } \bullet$$

$M \vdash e \xrightarrow{l} e'$ An expression e reduces to e' recording l in the trace.

$$\frac{M \vdash e \xrightarrow{l} e' \quad e' \neq \bot}{M \vdash \mathscr{E} \cdot e \xrightarrow{l} \mathscr{E} \cdot e'} \text{ (E-CTX)} \qquad \frac{M \vdash e \xrightarrow{l} \bot}{M \vdash \mathscr{E} \cdot e \xrightarrow{l} \bot} \text{ (E-BOT)} \qquad \frac{}{M \vdash \bot \longrightarrow \bot} \text{ (E-INF)}$$

$$\frac{e' = (x \mapsto v)\, e}{M \vdash \lambda x{:}t.e\; v \longrightarrow e'} \text{ (E-APP)} \qquad \frac{e' = (\alpha \mapsto t)e}{M \vdash \Lambda\alpha{::}k.e\; [t] \longrightarrow e'} \text{ (E-TAP)}$$

$$\frac{\textbf{if } (v \succ e_{pat} : \sigma) \textbf{ then } e = \sigma(e') \textbf{ else } e = e''}{M \vdash \text{case } v \text{ of } \overrightarrow{x{:}t}.e_{pat} : e' \text{ else } e'' \longrightarrow e} \text{ (E-CASE)} \qquad \frac{B : \overrightarrow{E} \in M}{M \vdash B \longrightarrow \llbracket B \rrbracket} \text{ (E-DELTA)}$$

$$\frac{\mathscr{D}, v \rightsquigarrow e \in \overrightarrow{E} \quad l = B : \mathscr{D}, v}{M, B : \overrightarrow{E}, M' \vdash \llbracket B \rrbracket^{\mathscr{D}}\; v \xrightarrow{l} e} \text{ (E-B1)} \qquad \frac{\mathscr{D}, v \rightsquigarrow e \notin \overrightarrow{E}}{M, B : \overrightarrow{E}, M' \vdash \llbracket B \rrbracket^{\mathscr{D}}\; v \longrightarrow \llbracket B \rrbracket^{\mathscr{D}, v}} \text{ (E-B2)}$$

$$\frac{\mathscr{D}, t \rightsquigarrow e \in \overrightarrow{E} \quad l = B : \mathscr{D}, t}{M, B : \overrightarrow{E}, M' \vdash \llbracket B \rrbracket^{\mathscr{D}}\; [t] \xrightarrow{l} e} \text{ (E-B3)} \qquad \frac{\mathscr{D}, t \rightsquigarrow e \notin \overrightarrow{E}}{M, B : \overrightarrow{E}, M' \vdash \llbracket B \rrbracket^{\mathscr{D}}\; [t] \longrightarrow \llbracket B \rrbracket^{\mathscr{D}, t}} \text{ (E-B4)}$$

$v \succ e_p : \sigma$ Pattern matching data constructors.

$$v \succ v : \cdot \text{ (U-ID)} \qquad v \succ x : x \mapsto v \text{ (U-VAR)} \qquad \frac{v \succ e :: \sigma \quad v' \succ \sigma\, e' : \sigma'}{v\,(v') \succ e\,(e') : \sigma, \sigma'} \text{ (U-CON)}$$

Figure B.3: Dynamic semantics of λAIR

Case (T-TAP):

$$\frac{\Gamma;A \vdash_\varphi e : q(\forall \alpha::k \xrightarrow{\varepsilon'} t'); \varepsilon \quad \Gamma \vdash t :: k}{\Gamma;A \vdash_\varphi e \ [t] : [\alpha \mapsto t]t'; \varepsilon \uplus ([\alpha \mapsto t]\varepsilon')} \text{ (T-TAP)}$$

If e is not a value, then by the induction hypothesis on the first premise, we have $M \vdash e \xrightarrow{l} e'$, and so, by (E-CTX) we have $M \vdash e \ [t] \xrightarrow{l} e' \ [t]$.

If e is a value, then by canonical forms of values of universally quantified types (applied to the first premise), we have two sub-cases:

Sub-case $e = \Lambda\alpha::k.e$: In this case, (E-TAP) is applicable and we step to $(\alpha \mapsto t)e$.

Sub-case $e = [\![B]\!]^{\mathscr{D}}$: In this case, either (E-B3) or (E-B4) is applicable. We first need to show that $B : \overrightarrow{E} \in M$. But, this follows from (A1) which requires $[\![B]\!]^V$ to be well-typed. Thus, we have that $B \in \text{dom}(S)$ and by the assumption of type-consistency of M and S, (A2), we have that $B : \overrightarrow{E} \in M$. The premises of (E-B3) and (E-B4) are mutually exclusive and total. So, a step using one or the other must be possible.

Case (T-BOT): Step using (E-INF).

Case (T-CAP):

$$\frac{\begin{array}{c} \Gamma;A \vdash_\varphi e : q(t' \Rightarrow t); \varepsilon \\ \Gamma;A' \vdash_\varphi e' : t'; \varepsilon' \end{array}}{\Gamma;A,A' \vdash_\varphi e \ (e') : t; \varepsilon \uplus \varepsilon'} \text{ (T-CAP)}$$

If e is not a value, by the induction hypothesis in the first premise we have, $M \vdash e_1 \ (e_2) \longrightarrow e_1' \ (e_2)$, using (E-CTX). Similarly, if e_2 is not a value. If both are values, then $v_1 \ (v_2)$ is a value too.

Case (T-CASE): We have case e of $\overrightarrow{x:t}.e_{pat} : e'$ else e''. If e is not a value, then by the penultimate form of $\mathscr{E}$ and the congruence (E-CTX) we can take a step. Otherwise, if e is a value, then we can always take a step using (E-CASE), since its premise is a tautology.

Case (T-ABS): $\lambda x{:}t.e$ is a value.

Case (T-APP):

$$\frac{\Gamma;A \vdash_\varphi e : q((x{:}t') \to t); \varepsilon_1 \quad \Gamma;A' \vdash_\varphi e' : t'; \varepsilon_2}{\Gamma;A,A' \vdash_\varphi e \ e' : [x \mapsto e']t; \varepsilon_1 \uplus \varepsilon_2} \text{ (T-APP)}$$

If either e_1 or e_2 are not values, then we can reduce using the (E-CTX) congruence rule. Otherwise, by canonical forms of function-typed values (applied to the first premise), we get that $e = \lambda x{:}t.e$ or $e = [\![B]\!]^{\mathscr{D}}$. In both cases the reasoning proceeds similarly to the (T-TAP) case, except using (E-APP) and (E-B1) or (E-B2).

Case (T-B1), (T-B2), (T-B3): $[\![B]\!]^{\mathscr{D}}$ is a value.

Case (T-DROP), (T-WKN), (T-CONV): Induction hypothesis applied to the hypothesis.

Case (T-X-type), (T-NC-type): Inapplicable.

$\square$

Proposition 32 (Well-formedness of environments). *If $\Gamma;A$ is well-formed, and (A1) $\Gamma;A \vdash e : t;\varepsilon$ contains a premise of the form $\Gamma';A' \vdash e' : t';\varepsilon'$ then $\Gamma';A'$ is well-formed and $\exists k.\Gamma' \vdash t' :: k$. Similarly, if (A1) contains a premise of the form $\Gamma'' \vdash t'' :: K$, then Γ'' is well-formed and $\vdash K$ ok.*

Proposition 33 (Weakening). *If $\Gamma;A$ is well-formed, and $\Gamma;A \vdash e : t;\varepsilon$. Then, for all Γ',A' such that $\Gamma,\Gamma';A,A'$ is well-formed, $\Gamma,\Gamma';A,A' \vdash e : t;\varepsilon$. Similarly, if $\Gamma \vdash t :: K$, then $\Gamma,\Gamma' \vdash t :: K$.*

Proposition 34 (Inversion of empty name constraints). *If $\Gamma;\cdot$ is well-formed, and $\Gamma;\cdot \vdash_\varphi v : t;\cdot$. Then, $\Gamma \vdash t::\mathtt{U}$.*

Proposition 35 (Inversion of non-empty name constraints). *If $\Gamma;\cdot$ is well-formed, and $\Gamma;\cdot \vdash_\varphi v : t;\varepsilon$ and $\varepsilon \neq \cdot$. Then, either $\Gamma \vdash t::\mathtt{A}$ or $\Gamma;\cdot \vdash_\varphi v : t;\cdot$.*

Proposition 36 (Uniqueness of kinding). *If $\Gamma \vdash t::k$ and $\Gamma \vdash t::k'$. Then $k = k'$.*

Lemma 37 (Substitution). *Given $\Gamma;\cdot$ well-formed, and $\Gamma';A$ such that:*

(A1) $\Gamma,\Gamma';A$ is well-formed.

(A2) $\Gamma,x:t_x,\Gamma';A_x,A$, well-formed, where $\Gamma \vdash t_x::k$, and $A_x = \cdot \vee A_x = a(x,k)$.

(A3) $\Gamma,x:t_x,\Gamma';A_x,A \vdash_\varphi e : t;\varepsilon$

(A4) $\Gamma;\cdot \vdash v:t_x;\varepsilon'$

(A5) $\varepsilon \uplus \varepsilon'$

Then, for $\sigma = x \mapsto v$,

$$\Gamma,\sigma(\Gamma');A \vdash \sigma(e) : \sigma(t);\varepsilon \uplus \varepsilon'$$

Proof. By mutual induction on the structure of (A3), together with Lemma 38.

Throughout, we will use $\sigma(\Gamma) = \Gamma$, since $x \notin \mathrm{dom}(\Gamma)$

Case (T-B): Base terms B are closed; so using (T-B) we can get

$$\Gamma,\sigma(\Gamma') \vdash \sigma(B) : \sigma(t);\cdot$$

To ensure that the name constraint is still $\varepsilon \uplus \varepsilon' = \varepsilon'$ we conclude with (T-AFN) that allows bound names ε' to be added at will.

Case (T-X):

$$\frac{\Gamma,x:t_x,\Gamma'(y) = t \quad \Gamma,x:t_x,\Gamma' \vdash t :: U}{\Gamma,x:t_x,\Gamma';\cdot \vdash_\varphi y : t;\cdot} \text{ (T-X)}$$

We consider two sub-cases, depending on whether or not $y = x$.

Sub-case $y \neq x$: Thus, $\sigma(y) = y$. We have two sub-cases depending on whether y appears in Γ or in Γ'.

Sub-case (i): $y : t \in \Gamma'$. In this case, $FV(t) \cap \mathrm{dom}(\sigma) \neq \emptyset$; thus our conclusion is of the form $\Gamma, \sigma(\Gamma'); \cdot \vdash y : \sigma(t); \cdot$, since we have $y : \sigma(t) \in \Gamma'$, to satisfy the first premise, and from Lemma 38, we have $\Gamma, \sigma(\Gamma') \vdash \sigma(t) :: U$ for the second premise. Finally, if $\varepsilon' \neq \cdot$, we conclude with (T-AFN) and introduce the additional name effects ε'.

Sub-case (ii): $y : t \in \Gamma$. From our initial remark, we know that $\sigma\Gamma = \Gamma$; thus, $\sigma(t) = t$. Our conclusion is of the form $\Gamma, \sigma(\Gamma'); \cdot \vdash y : t, \cdot$, with the first premise satisfied trivially. The second premise follows from the mutual induction hypothesis of Lemma 8 to establish that $\Gamma, \sigma\Gamma' \vdash \sigma t :: U$. Finally, as previously, we conclude with (T-AFN) for the effects, if necessary.

Sub-case $y = x$: Thus, $\sigma(y) = v$. From (A4) we have $\Gamma; \cdot \vdash v : t_x; \varepsilon'$ and, from weakening, we can establish $\Gamma, \sigma(\Gamma'); \cdot \vdash v : t_x; \varepsilon'$. Finally, since $x \notin \mathrm{dom}(\Gamma)$ we can conclude from Proposition 32 that $x \notin FV(t_x)$, and thus $\sigma(t_x) = t_x$.

Case (T-XA):

$$\frac{\Gamma, x : t_x, \Gamma'(y) = t}{\Gamma, x : t_x, \Gamma'; y \vdash_\varphi y : t; \cdot} \text{ (T-XA)}$$

Again, we proceed by cases on whether $x = y$.

Sub-case $y \neq x$: Identical to the same sub-case of (T-X).

Sub-case $y = x$: By weakening, we have $\Gamma, \sigma(\Gamma'); \cdot \vdash v : t; \varepsilon'$, which is sufficient since $A_x = y$ and $A = \cdot$.

Case (T-NEW):

$$\frac{\begin{array}{c}\Gamma, x : t_x, \Gamma'; A_x, A \vdash e : t; \varepsilon \\ \Gamma, x : t_x, \Gamma' \vdash t :: U \qquad \Gamma, x : t_x, \Gamma'(\alpha) = \mathbb{N}\end{array}}{\Gamma, x : t_x, \Gamma'; A_x, A \vdash \mathsf{new}\ e : {}_{\mathsf{i}}t^\alpha; \alpha \uplus \varepsilon} \text{ (T-NEW)}$$

By the induction hypothesis we have $\Gamma, \sigma(\Gamma'); A \vdash e : \sigma(t); \varepsilon \uplus \varepsilon'$, since, by assumption, ε' is disjoint from $\alpha \uplus \varepsilon$. From mutual induction with Lemma 38 we have $\Gamma, \sigma(\Gamma') \vdash \sigma(t) :: U$, and since $\alpha \notin \mathrm{dom}(\sigma)$ the third premise is also satisified. This suffices to establish the conclusion.

$$\Gamma, \sigma(\Gamma'); A \vdash \sigma(\mathsf{new}\ e) : \sigma(!t^\alpha); \alpha \uplus \varepsilon \uplus \varepsilon'$$

Case (T-ABS):

$$\frac{\begin{array}{c}\Gamma, x{:}t_x, \Gamma' \vdash t_y :: k \qquad q = p((A_x, A), \varepsilon) \\ \Gamma, x{:}t_x, \Gamma', y{:}t_y; A_x, A, a(y, k) \vdash e : t_e; \varepsilon\end{array}}{\Gamma, x{:}t_x, \Gamma'; A_x, A \vdash_\varphi \lambda y{:}t_y.e : q((y{:}t_y) \to t_e); \varepsilon} \text{ (T-ABS)}$$

Using Lemma 38 on the first premise we get

$$(P1') \quad \Gamma, \sigma(\Gamma') \vdash \sigma(t_y) :: k$$

We proceed on depending on whether or not ε' is empty.

Sub-case $\varepsilon' = \cdot$:

Using the induction hypothesis on the third premise we can get

$$(P3') \quad \Gamma, \sigma(\Gamma', y{:}t_y); A, a(y,k) \vdash \sigma(e) : \sigma(t_e); \varepsilon$$

From Proposition 34 and $\varepsilon' = \cdot$ we can establish $\Gamma \vdash t_x{::}\mathtt{U}$. From the uniqueness of kinding, Proposition 36, we can establish that $k = \mathtt{U}$ and thus $(A_x, A) = A$. Thus $q((A_x, A), \varepsilon \uplus \varepsilon') = q(A, \varepsilon)$ and we can conclude with the appropriate affinity qualifier on the function type.

Sub-case $\varepsilon' \neq \cdot$: We further divide this into sub-cases.

Sub-sub-case $x \in A_x$: Using the induction hypothesis on the third premise we can get

$$(P3') \quad \Gamma, \sigma(\Gamma', y{:}t_y); A, a(y,k) \vdash \sigma(e) : \sigma(t_e); \varepsilon \uplus \varepsilon'$$

Now, to get the right affinity qualifier on the result, we must show $p((A_x, A), \varepsilon) = p(A, \varepsilon \uplus \varepsilon')$. But, notice that $(Ax, A) \neq \cdot \wedge fx \uplus fx' \neq \cdot$. Thus, we have $p((Ax, A), \varepsilon) = p(A, \varepsilon \uplus \varepsilon') = \mathtt{¡}$

Sub-sub-case $x \notin A_x$: In this case, we apply Proposition 35 to get either (i) $\Gamma \vdash t_x{::}\mathtt{A}$ or (ii) $\Gamma; \cdot \vdash_\varphi v : t_x, \cdot$.

In case (i), since $k = \mathtt{A}$ and $x \notin Ax, A$, we must have $x \notin FV(e)$. Thus, $\sigma(e) = e$ and we can use the induction hypothesis to construct

$$(P3') \quad \Gamma, \sigma(\Gamma', y{:}t_y); A, a(y,k) \vdash e : \sigma(t_e); \varepsilon$$

Now, the affinity qualifier on the result is $p(A, \varepsilon) = p((A_x, A), \varepsilon)$, as required.

In case (ii), we can use $\Gamma; \cdot \vdash_\varphi v : t_x, \cdot$ to reduce to the first subcase with $\varepsilon' = \cdot$.

Case (T-APP): We have two possible ways of inverting (T-APP), depending on whether A_x is used in the first or the second premise. Let $A_x, A = A', A''$.

$$\frac{\begin{array}{c} \Gamma, x{:}t_x, \Gamma'; A' \vdash e : \mathtt{¡}((y{:}t') \rightarrow t); \varepsilon_1 \\ \Gamma, x{:}t_x, \Gamma'; A'' \vdash e' : t'; \varepsilon_2 \end{array}}{\Gamma, x{:}t_x, \Gamma'; A_x, A \vdash e\, e' : [y \mapsto e']t; \varepsilon_1 \uplus \varepsilon_2} \text{ (T-APP)}$$

We can apply the induction hypothesis to each of the two premises and obtain

$$\Gamma, \sigma(\Gamma'); A' \setminus A_x \vdash \sigma(e) : \mathtt{¡}(y{:}\sigma(t')) \rightarrow \sigma(t); \varepsilon_1 \uplus \varepsilon_1'$$

and

$$\Gamma, \sigma(\Gamma'); A'' \setminus A_x \vdash \sigma(e') : \sigma(t'); \varepsilon_2 \uplus \varepsilon_2'$$

Sub-case $\varepsilon' = \cdot$: In this case we have $\varepsilon_1' = \varepsilon_2'$ and the conclusion is straightforward.

Sub-case $\varepsilon' \neq \cdot$: Here, we proceed as in (T-ABS) and apply Proposition 35 to split into two subcases.

Sub-sub-case $\Gamma \vdash t_x{::}\mathtt{A}$: In this case, since x is in only either A' or A'' (not both), we can conclude that x is free in either e or in e' only (not both). Thus, either $\varepsilon_1' = \cdot \vee \varepsilon_2' = \cdot$ and $\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon_1' \uplus \varepsilon_2'$.

Sub-sub-case $\Gamma;\cdot \vdash v : t_x, \cdot$: In this case, we can reduce to the first subcase and contruct the premises appropriately to derive $\Gamma, \sigma(\Gamma') \vdash_{\varphi} \sigma(e\ e') : \sigma(t), \varepsilon_1 \uplus \varepsilon_2$. Finally, we can conclude with (T-AFN) to introduce ε' and establish the name constraint $\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon'$.

Case (T-TAB):

$$\frac{\begin{array}{c}\Gamma, x{:}t_x, \Gamma', \alpha{::}k; A_x, A \vdash e : t_e; \varepsilon_0 \uplus \varepsilon_1 \\ \alpha \notin \varepsilon_0 \qquad \varepsilon_1 \in \{\cdot, \alpha\} \qquad q = p((A_x, A), \varepsilon_0)\end{array}}{\Gamma, x{:}t_x, \Gamma'; A_x, A \vdash \Lambda\alpha{::}k.e : q(\forall\alpha{::}k \xrightarrow{\varepsilon_1} t_e); \varepsilon_0}$$

Applying the induction hypothesis to the first premise, we get

$$\Gamma, \sigma(\Gamma', \alpha{::}k); A \vdash \sigma(e) : \sigma(t_e); \varepsilon_0 \uplus \varepsilon_1 \uplus \varepsilon'$$

where we get ε' disjoint from ε_1 by α-renaming.

Since $\alpha \notin \text{dom}(\sigma)$ we can rewrite this as

$$\Gamma, \sigma(\Gamma'), \alpha{::}k; A \vdash \sigma(e) : \sigma(t_e); \varepsilon_0 \uplus \varepsilon' \uplus \varepsilon_1$$

This allows us to conclude with

$$\Gamma, \sigma(\Gamma'); A \vdash \sigma(e) : q'(\sigma(\forall\alpha{::}k \xrightarrow{\varepsilon_1} t_e)); \varepsilon_0 \uplus \varepsilon'$$

where $q' = p(A, \varepsilon_0 \uplus \varepsilon')$. In order to show that $q = q' = p((A_x, A), \varepsilon_0)$ we follow the same argument as in (T-ABS).

Case (T-TAP):

$$\frac{\Gamma, x{:}t_x, \Gamma'; A_x, A \vdash_{\varphi} e : q(\forall\alpha{::}k \xrightarrow{\varepsilon_1} t'); \varepsilon_0 \qquad \Gamma, x{:}t_x, \Gamma' \vdash t :: k}{\Gamma, x{:}t_x, \Gamma'; A_x, A \vdash_{\varphi} e\ [t] : [\alpha \mapsto t]t'; \varepsilon_0 \uplus ([\alpha \mapsto t]\varepsilon_1)} \text{(T-TAP)}$$

From the induction hypothesis on the first premise we get

$$\Gamma, \sigma(\Gamma'); A \vdash \sigma(e) : \forall\alpha{::}k \xrightarrow{\varepsilon_1} \sigma(t'); \varepsilon_0 \uplus \varepsilon'$$

For the second premise, from the mutual induction hypothesis of Lemma 38 we get

$$\Gamma, \sigma(\Gamma') \vdash \sigma(t) :: k$$

This is sufficient to establish the conclusion

$$\Gamma, \sigma(\Gamma'); A \vdash \sigma(e)\ [\sigma(t)] : [\alpha \mapsto \sigma(t)]\sigma(t'); \varepsilon_0 \uplus ([\alpha \mapsto t]\varepsilon_1) \uplus \varepsilon'$$

Case (T-CAP): Similar to (T-APP), we have two cases depending on whether A_x is used in the first or second premise. As previously, we establish that if $A_x = x$ then $\varepsilon' = \varepsilon'_1 \uplus \varepsilon'_2$ and in the conclusion we get $\varepsilon_1 \uplus \varepsilon_2 \uplus \varepsilon'_1 \uplus \varepsilon'_2$.

Case (T-CASE): Induction hypothesis on the first, third, fourth and fifth premise. For the second premise, use Lemma 38. As with (T-CAP) and (T-APP), we use affinity to ensure that if $t_x{::}A$, then x is free only in e or in e' and e''. If it is the latter, then since the

effects ε' and ε'' are allowed to overlap, we can establish the conclusion.

Case (T-AFN), (T-DROP), (T-B1), (T-B2), (T-B3): All follow from the induction hypothesis.

Case (T-CONV): Here, we must show that type equivalence is preserved under substitution. The only interesting case here is (TE-RED). Here, we restrict reduction to closed type-level expressions e and e'. Thus $\sigma(e) = e$ and $\sigma(e') = e'$ and so $M \vdash e \longrightarrow e' \iff M \vdash \sigma(e) \longrightarrow \sigma(e')$. By axiomatizing that the type of e is preserved under reduction to e', we can use the induction hypothesis on the first and third premises to establish the conclusion. $\qquad \square$

Lemma 38 (Substitution for kinding judgment). *Given well-formed* $\Gamma; \cdot$ *well-formed, and* Γ' *such that:*

(A1) Γ, Γ' *is well-formed.*

(A2) $\Gamma, x : t_x, \Gamma'$ *also well-formed.*

(A3) $\Gamma, x : t_x, \Gamma' \vdash t :: k$

(A4) $\Gamma; \cdot \vdash v : t_x$

Then, for $\sigma = x \mapsto v$,

$$\Gamma, \sigma(\Gamma') \vdash \sigma(t) :: k$$

Proof. By mutual induction on the structure of (A3), together with Lemma 37.

Case (K-A): $\alpha \in \Gamma, \Gamma'$ and $FV(k) = \emptyset$.

Case (K-N): Induction hypothesis on the first premise gives us $\Gamma, \sigma(\Gamma') \vdash \sigma(t) :: A$. The second premise is trivial since $\alpha \in \Gamma, \Gamma'$ and $\alpha \notin \mathrm{dom}(\sigma)$.

Case (K-TC): S is unchanged, and $FV(K) = \emptyset$.

Case (K-AFN): Induction hypothesis.

Case (K-ALL): Induction hypothesis gives us $\Gamma, \sigma\Gamma', \alpha::k \vdash \sigma(t) :: k'$.

Case (K-FUN): Induction hypothesis on the first premise gives us $\Gamma, \sigma(\Gamma') \vdash \sigma(t) :: k$, and on the second premise $\Gamma, \sigma(\Gamma', x{:}t) \vdash \sigma(t') :: k'$. Finally, for the third premise, $\varepsilon \cap \mathrm{dom}(\sigma) = \emptyset$.

Case (K-TAP): Induction hypothesis on each premise.

Case (K-DEP): This is the interesting case, where we must rely on mutual induction with Lemma 37 for the second premise to establish $\Gamma, \sigma(\Gamma'); \cdot \vdash_{\mathsf{type}} \sigma(e) : \sigma(t'); \varepsilon'$. We can conclude with (T-NC-type) and use the phase distinction to establish $\Gamma, \sigma(\Gamma'); \cdot \vdash_{\mathsf{type}} \sigma(e) : \sigma(t'); \cdot$ as required. For the first premise, we use the induction hypothesis to get $\Gamma, \sigma(\Gamma') \vdash t::t' \to K$, and from well-formedness of kinds we have that $FV(t') = \emptyset$. Thus, $\sigma(t') = t'$ and we have the conclusion $\Gamma, \sigma(\Gamma') \vdash t \, e :: K$.

Case (K-CON): Induction hypothesis on each premise.

$\qquad \square$

Lemma 39 (Type substitution). *Given well-formed Γ well-formed, and Γ' such that:*

(A1) $\Gamma, \alpha :: k, \Gamma'$ *well-formed.*

(A2) $\Gamma, \alpha :: k, \Gamma'; \cdot \vdash_\varphi e : t; \varepsilon$

(A3) $\Gamma \vdash t' :: k$

(A5) $\sigma = \alpha \mapsto t'$

Then,

$$\Gamma, \sigma(\Gamma') \vdash_\varphi \sigma e : \sigma t; \sigma \varepsilon$$

Proof. By mutual induction with the proposition $\Gamma, \alpha :: k, \Gamma' \vdash t :: K \Rightarrow \Gamma, \sigma(\Gamma') \vdash \sigma t :: k$, on the structure of (A2). Similar to the proof of Lemma 14. $\qquad \square$

Proposition 40 (Unification respects types). *If all of the following are true*

(A1) $\cdot; \cdot \vdash v : t$

(A2) $\overrightarrow{x : t}; A \vdash e_p : t$, *where* $\cdot; \cdot \vdash t_i :: A \Rightarrow x \in A$

(A3) $v \succ e_p : \sigma$

   Then, $dom(\sigma) = \overrightarrow{x}$, and $\cdot; \cdot \vdash \sigma(x_i) : t_i$.

Proposition 41 (Inversion of type abstractions). *Given $\Gamma; \cdot$ well formed, and $\Gamma; \cdot \vdash_\varphi \Lambda \alpha :: k e : \forall \alpha :: k \xrightarrow{\varepsilon_1} t_e; \varepsilon_0$. Then, $\Gamma, \alpha :: k; \cdot \vdash_\varphi e : t_e : \varepsilon_0 \uplus \varepsilon_1$.*

Proposition 42 (Inversion of abstractions). *Given $\Gamma; \cdot$ well formed, and $\Gamma; \cdot \vdash_\varphi \lambda x : t.e : (x : t) \rightarrow t_e; \varepsilon$. Then, for some k, $\Gamma, x : t; a(x, k) \vdash_\varphi e : t_e; \varepsilon$.*

Theorem 43 (Preservation). *Given $\Gamma = \alpha_1 :: N, \ldots, \alpha_n :: N$ and $(A1)\Gamma; \cdot \vdash_{\text{term}} e : t; \varepsilon$ and the interpretation M and S are type-consistent, then if $(A2)M \vdash e \xrightarrow{l} e'$, $\Gamma; \cdot \vdash_{\text{term}} e' : t; \varepsilon$.*

Proof. By induction on the structure of the derivation (A1).

Case (T-X), (T-XA), (T-X-type), (T-NC-type): Inapplicable.

Case (T-B): Inversion on (A2) gives (E-DELTA). Establish the conclusion by using (T-B3) with (A1) in the premise.

Case (T-BOT): Inversion on (A2) gives (E-INF). Conclusion is trivial from (A1).

Case (T-NEW): Inversion on (A2) gives (E-CTX) or (E-BOT). In the first case, the induction hypothesis applied to the first premise suffices. In the second case, (T-BOT) suffices.

Case (T-DROP): Apply (T-DROP) with the induction hypothesis in the premise.

Case (T-WKN): Apply (T-WKN) with the induction hypothesis in the premise.

Case (T-TAB), (T-ABS): In both cases, *e* is value.

Case (T-TAP): If e is not a value, inversion of (A2) gives (E-CTX) of (E-BOT) and apply the induction hypothesis or (T-BOT) to conclude.

If e is a value, the inversion of (A2) gives (E-TAP), (E-B3) or (E-B4).

Sub-case (E-TAP): From the inversion lemma, Proposition 41, applied to the first premise, we get

$$(A1.1) \qquad \Gamma, \alpha::k; \cdot \vdash_{\mathsf{term}} e : t_e; \varepsilon_0 \uplus \varepsilon_1$$

Applying the type substitution lemma, Lemma 39, to (A1.1) using the second premise of (A1) we get the desired result:

$$\Gamma; \cdot \vdash_{\mathsf{term}} (\alpha \mapsto t)e : (\alpha \mapsto t)t'; (\alpha \mapsto t)\varepsilon_0 \uplus \varepsilon_1$$

Sub-case (E-B3): The result follows from type consistency of M and S.

Sub-case (E-B4): Apply (T-B4) using (A1) in the premise.

Case (T-APP): If either e or e' are not values, inversion of (A2) gives (E-CTX) or (E-BOT) and we apply the induction hypothesis of (T-BOT) to conclude.

If both are values, then inversion of (A2) gives (E-APP), (E-B1) or (E-B2).

Sub-case (E-APP): From the inversion lemma, Proposition 42, applied to the first premise, we get

$$(A1.1) \qquad \Gamma, x::t'; a(x,k) \vdash_{\mathsf{term}} e : t; \varepsilon$$

Applying the substitution lemma, Lemma 37, to (A1.1) using the second premise of (A1) we get the desired result:

$$\Gamma; \cdot \vdash_{\mathsf{term}} (x \mapsto v)e : (x \mapsto v)t'; \varepsilon_0 \uplus \varepsilon_1$$

Sub-case (E-B1): The result follows from type consistency of M and S.

Sub-case (E-B2): Apply (T-B1) using (A1) in the premise.

Case (T-CAP): Inversion of (A1) gives (E-CTX) or (E-BOT). Result follows from induction hypothesis or (T-BOT).

Case (T-CASE): If the discriminant e is not a value, inversion of (A2) gives (E-CTX) or (E-BOT) which we handle as in the other cases.

If e is a value, then inversion of (A2) gives (E-CASE). There are two sub-cases, depending on which branch is taken.

Sub-case (else-branch): The final premise of (T-CASE) gives:

$$\Gamma; \cdot \vdash e'' : t; \varepsilon''$$

where $\varepsilon'' \subseteq \varepsilon$. However, we can always use (T-WKN) to introduce additional effects in the conclusion.

Sub-case (case-branch): From the premises of (A1) we have

$$\Gamma; \cdot \vdash_{\varphi} v : t_v; \varepsilon_v$$

298

and

$$\Gamma, \overrightarrow{x:t}; A'' \vdash_\varphi e' : t$$

From Lemma 40 we can show that in $v \succ e_p : \sigma$, $\forall x_i \in \text{dom}(\sigma).\Gamma; \cdot \vdash \sigma(x_i) : t_i$, then by repeated application of Lemma 37, we can conclude

$$\Gamma; \cdot \vdash_\varphi \sigma(e') : t; \varepsilon'$$

Finally, as in the else case, we can always expand the effects ε' using (T-AFN), if necessary.

Case (T-B1), (T-B2), (T-B3): All of these are values.

Case (T-CONV): Induction hypothesis on the premise. □

$$
\begin{array}{llll}
\text{policy} & \pi & ::= & (id, P, \overrightarrow{\sigma}, \overrightarrow{R_r}, \overrightarrow{R_t}) \\
\text{rule} & R & ::= & (id, x, d, \overrightarrow{\exists x{:}t.C}, e, A) \\
\text{judgment index} & \rho & ::= & r \mid t
\end{array}
$$

$\pi \models S$ Translation from a policy π to a signature S

$$
\frac{S = id{:}Id, B{:}Class, S_0 \quad S \models \overrightarrow{\sigma} : S_\sigma \quad S, S_\sigma \models_r \overrightarrow{R_r} : S_r \quad S, S_\sigma, S_r \models_t \overrightarrow{R_t} : S_t}{(id, P, \overrightarrow{\sigma}, \overrightarrow{R_r}, \overrightarrow{R_t}) \models S, S_\sigma, S_r, S_t}
\qquad
\frac{S \models_\rho R_0 : S_0 \quad S, S_0 \models_\rho \overrightarrow{R} : S'}{S \models_\rho R_0, \overrightarrow{R} : S_0, S'}
$$

$S \models \overrightarrow{\sigma} : S'$ Translation from states $\overrightarrow{\sigma}$ to signature S'

$$
\frac{S \models \sigma_0 : S_0 \quad S, S_0 \models \overrightarrow{\sigma} : S'}{S \models \sigma_0, \overrightarrow{\sigma} : S_0, S'}
\qquad
\frac{C \notin \mathrm{dom}(S) \quad t = Class \Rightarrow Instance}{S \models C : (C{:}t)}
$$

$$
\frac{i \in 1 \ldots n \quad \cdot \vdash_{S_0} t_i :: U \quad C \notin \mathrm{dom}(S)}{S \models C \text{ of } \overrightarrow{t} : (C{:}Class \Rightarrow t_1 \Rightarrow \ldots \Rightarrow t_n \Rightarrow Instance)}
$$

$S \models_\rho \overrightarrow{R} : S'$ Translation of a rule.

$$
\frac{\begin{array}{c}
\Gamma = src{::}N, dst{::}N, \alpha{::}U, s{:}\text{\textexclamdown}Instance^{src}, x{:}Protected\ \alpha\ src, d{:}\text{\textexclamdown}Instance^{dst}, d'{:}Class \\
src; dst; S; \Gamma \models_\rho \overrightarrow{\exists x{:}t.C}; e : t' \qquad id \notin \mathrm{dom}(S) \qquad B{:}Class \in S \\
t_r = \forall src{::}N, dst{::}N, \alpha{::}U. (s{:}\text{\textexclamdown}Instance^{src}) \rightarrow (ClassOf\ src\ B) \rightarrow (x{:}Protected\ \alpha\ src) \rightarrow \\
(d{:}\text{\textexclamdown}Instance^{dst}) \rightarrow (d'{:}Class) \rightarrow (ClassOf\ dst\ d') \rightarrow t'
\end{array}}{S \models_\rho (id, x, d, \overrightarrow{\exists x{:}t.C}, e, A) : (id{:}t_r)} \text{ (S-RULE)}
$$

Figure B.4: Translating an AIR policy to a λAIR signature (Part 1)

$$\frac{\Gamma \vdash t :: k \qquad src;dst;S;\Gamma,x{:}t \models C : t' \qquad src;dst;S;\Gamma,x{:}t \models_\rho \overrightarrow{\exists x{:}t.C};e : t''}{src;dst;S;\Gamma \models_\rho \exists x{:}t.C, \overrightarrow{\exists x{:}t.C};e : (x{:}t) \to t' \to t''} \text{ (TR-COND)}$$

$$\frac{\Gamma,s'{::}N;\cdot \vdash e : t;\varepsilon}{s;d;S;\Gamma \models_t \cdot;e : (¡Instance^s \times ¡Instance^d)} \text{ (T-BODY)}$$

$$\frac{\Gamma,s'{::}N;\cdot \vdash e : Protected\ \alpha\ s;\varepsilon}{s;d;S;\Gamma \models_r \cdot;e : (¡Instance^s \times ¡Instance^d \times Protected\ \alpha\ d)} \text{ (R-BODY)}$$

$$\frac{s;d;S;\Gamma \models A_1 : (e_1,Class) \qquad s;d;S;\Gamma \models A_2 : (e_2,Class)}{s;d;S;\Gamma \models A_1\ IsClass\ A_2 : IsClass\ e_1\ e_2} \text{ (C-CLS)}$$

$$\frac{s;d;S;\Gamma \models A_1 : (e_1,Prin) \qquad s;d;S;\Gamma \models A_2 : (e_2,Prin)}{s;d;S;\Gamma \models A_1\ ActsFor\ A_2 : ActsFor\ e_1\ e_2} \text{ (C-ACTS)}$$

$$\frac{s;d;S;\Gamma \models A_1 : (e_1,¡Instance) \qquad s;d;S;\Gamma \models A_2 : (e_2,Instance)}{s;d;S;\Gamma \models A_1\ InState\ A_2 : InState\ e_1\ e_2} \text{ (C-STATE)}$$

$$\frac{s;d;S;\Gamma \models A_1 : (e_1,Int) \qquad s;d;S;\Gamma \models A_2 : (e_2,Int)}{s;d;S;\Gamma \models A_1\ LessThan\ A_2 : LEQ\ e_1\ e_2} \text{ (C-LEQ)}$$

$$\frac{\Gamma;\cdot \vdash x : t;\cdot}{s;d;S;\Gamma \models x : (x,t)} \text{ (A-X)} \qquad \frac{\Gamma;\cdot \vdash B : t;\cdot}{s;d;S;\Gamma \models B : (B,t)} \text{ (A-B)}$$

$$\frac{\Gamma;\cdot \vdash x : ¡Instance^s;\cdot}{s;d;S;\Gamma \models Self : (x,¡Instance^s)} \text{ (A-SELF)}$$

$$\frac{s;d;S;\Gamma \models A : (e,Class)}{s;d;S;\Gamma \models Principal(A) : (principal\ e,Prin)} \text{ (A-PRIN)}$$

$$\frac{(B{:}Class) \in S \qquad s;d;S;\Gamma \models A_i : (e_i,t_i)}{s;d;S;\Gamma \models C(\overrightarrow{A}) : (C\ (B)\ (e_1)\ (\ldots)\ (e_n),Instance^\alpha)} \text{ (A-ST)}$$

$$\frac{s;d;S;\Gamma \models A : (e,¡Instance^{src}) \qquad (B : Class) \in S}{s;d;S;\Gamma \models Class(A) : (B,Class)} \text{ (A-SCLS)}$$

$$\frac{s;d;S;\Gamma \models A : (e,¡Instance^{dst}) \qquad (d : Class) \in \Gamma}{s;d;S;\Gamma \models Class(A) : (d,Class)} \text{ (A-DCLS)}$$

Figure B.5: Translating an AIR policy to a λAIR signature (Part 2)

$A;\pi \models T;A'$ An automaton in state A, accepts T and transitions to A'

$$\frac{A;\pi \models T;A'}{A;\pi \models \cdot,T;A'} \text{ (L-DOT)} \qquad \frac{A;\pi \models l;A' \quad A';\pi \models T;A''}{A;\pi \models l,T;A''} \text{ (L-TR)}$$

$$\frac{(B,\ldots) \notin \pi.\overrightarrow{\mathsf{R}}}{A;\pi \models (B:\mathscr{D});A} \text{ (L-NOTX)} \qquad \frac{A;\pi \models (B:\mathscr{D});A'}{A;\pi \models (B:t,\mathscr{D});A'} \text{ (L-SKIP-t)}$$

$$\frac{\begin{array}{c} src::N;\cdot \vdash v_{inst} : {}_{\mathclap{i}}Instance^{src} \\ src::N;\cdot \vdash v_{ev} : ClassOf\ src\ (\mathsf{Class}\ (\pi.id)\ (v_{prin})) \qquad v_{inst};A;\pi \overset{\mathscr{D}}{\Rightarrow} A';B \end{array}}{A;\pi \models (B:v_{inst},v_{ev},\mathscr{D});A'} \text{ (L-ENTER)}$$

$v_{src};A;\pi \overset{\mathscr{D}}{\Rightarrow} A';B$ Given trace event $\mathscr{D}$, A transitions to A' using rule B

$$\frac{\begin{array}{c} (B,x,d,\overrightarrow{\mathsf{RC}},e,A') \in \pi.\overrightarrow{\mathsf{R}} \qquad v_{src} = \mathsf{new}\ v'_{src} \qquad \pi \models S \qquad s;d;S;\cdot \models A : (v'_{src},t) \\ \sigma_c = ((\mathsf{Class}\ d) \mapsto v_{cd}) \qquad \sigma = (\mathsf{Self} \mapsto v_{src}, x \mapsto v_x, d \mapsto v_{dst}) \qquad \sigma(\sigma_c(\overrightarrow{\mathsf{RC}})) \models \mathscr{D} \end{array}}{v_{src};A;\pi \overset{v_x,v_{dst},v_{cd},v_{ev},\mathscr{D}}{\Rightarrow} A';B} \text{ (TX)}$$

$\overrightarrow{\mathsf{RC}} \models \mathscr{D}$ Trace event $\mathscr{D}$ justifies release conditions $\overrightarrow{\mathsf{RC}}$

$$\frac{\begin{array}{c} \cdot;\cdot \vdash v_{ev} : \mathscr{T}(\mathscr{R})\ e_1\ e_2 \qquad \sigma_0 = (x \mapsto v_x) \qquad \sigma_0 A_1 \succ e_1 : \sigma_1 \\ \sigma_1\sigma_0 A_2 \succ e_2 : \sigma_2 \qquad \sigma = \sigma_0,\sigma_1,\sigma_2 \qquad \sigma(\overrightarrow{\mathsf{RC}}) \models \mathscr{D} \end{array}}{\exists x{:}t.A_1\ \mathscr{R}\ A_2,\overrightarrow{\mathsf{RC}} \models v_x,v_{ev},\mathscr{D}} \text{ (CERT)}$$

where $\mathscr{T}(\mathsf{ActsFor}) = ActsFor \quad \mathscr{T}(\mathsf{InState}) = InState \quad \mathscr{T}(\mathsf{IsClass}) = IsClass \quad \mathscr{T}(\leq) = LEQ$

$A \succ e : \sigma$ Unification of atom A with expression e

$$\frac{}{x \succ e : x \mapsto e} \qquad \frac{}{e \succ e :} \qquad \frac{A \succ e : \sigma}{\mathsf{Principal}\ A \succ principal\ e : \sigma} \qquad \frac{A \succ e : \sigma}{\mathsf{Class}\ A \succ class\_of\_inst\ e : \sigma}$$

Figure B.6: Trace acceptance condition defined by an AIR class.

B.2   Proof of Correct API Usage

In this section, we first define a translation that produces a λAIR signature from an AIR policy. We then define a semantics for an AIR policy as a regular language—i.e., an automaton that accepts strings. Finally, we show that the execution trace generated by a λAIR program is a member of the language accepted by the AIR automaton.

Definition 44 (Consistency of a model)**.** *A model M and a policy π are consistent if, and only if,*

$$\pi \models S \;\Rightarrow S \text{ and } M \text{ are type consistent}$$

And, the state transitions specified by M are in accordance with the policy π. I.e., if all of the following are true

1. $(B, x, d, \overrightarrow{\exists x{:}t.\vec{C}}, e, A) \in \pi$

2.
$$\Gamma = src{::}N, dst{::}N, \alpha{::}U, s{:}\textit{¡Instance}^{src},$$
$$x{:}\textit{Protected } \alpha \; src, d{:}\textit{¡Instance}^{dst}, d'{:}\textit{Class}, \overrightarrow{x{:}t}$$

3. $src; dst; S; \Gamma \models A : (e_A, t)$

4. $B : \mathscr{D} \leadsto v \in M$

5. $\sigma = (src \mapsto \mathscr{D}_1, dst \mapsto \mathscr{D}_2, \ldots, x_n \mapsto \mathscr{D}_m)$

   Then, $v = \mathsf{Pair}\,(v_{src})\,(v')$, and $v_{src} = \sigma(e_A)$.

Theorem 45 (Security)**.** *Given all of the following:*

(A1)   An AIR *declaration π of a class with identifier C owned by principal P.*

(A2)   A signature S such that $\pi \models S$.

(A3)   M and π are consistent.

(A4) $\Gamma = src{::}N, dst{::}N, s : \textit{¡Instance}^{src}$.

(A5) $\Gamma; s \vdash e : t; dst$, and e is π-free.

(A6) $v = \mathsf{new}\,\mathsf{Init}\,(\mathsf{Class}\,(\mathsf{C})\,(P))$.

(A7) $M \vdash ((s \mapsto v)e) \xrightarrow{l_1} e_1 \ldots \xrightarrow{l_n} e_n$.

Then

$$Init; \pi \models (l_1, \ldots, l_n); A'$$

Proof. By induction on the length of the string $l_1, \ldots, l_n$. We strengthen the induction hypothesis to actually prove $A; \pi \models (l_1, \ldots, l_n); A'$, where

(A6') $s; d; S; \cdot \models A : (v', (Instance^{src}))$ and $v = \text{new } v'$, such that $src::N; \cdot \vdash v : \text{¡}Instance^{src}$.

Case $(l_1 = \cdot)$: Apply (L-DOT) and use the induction hypothesis in the premise, since the length of the trace is reduced by one.

Case $(l_1 = B : \mathscr{D})$ AND $B \notin \pi.\overrightarrow{\mathsf{R}}$: Apply (L-TR) with (L-NOT-X) in the first premise, and the induction hypothesis in the second premise.

Case $(l_1 = B : \mathscr{D})$ AND $(B, \ldots) \in \pi.\overrightarrow{\mathsf{R}}$: This is the interesting case. We will use (L-TR) with the induction hypothesis in the second premise. Our first goal is to show that the first premise can be satisfied.

By premise (A2), and since we have $(B, \ldots) \in \pi.\overrightarrow{\mathsf{R}}$, we have $(B : t_B) \in S$, where $S' \models_\rho (B, \ldots) : (B : t_B)$ from (S-RULE). Additionally,

$$t_B = \forall src, dst::N, \alpha::U.(s:\text{¡}Instance^{src}) \to (ClassOf\ src\ C_B) \to$$
$$(x:Protected\ \alpha\ src) \to (d:\text{¡}Instance^{dst}) \to$$
$$(d':Class) \to (ClassOf\ dst\ d') \to t'_B$$

where, $C_B = \mathsf{Class}\ (\pi.id)\ (P)$, is the representation of the class of π. Next, from type consistency of M and S we have

$$\mathscr{D} = t_{src}, t_{dst}, t_\alpha, v_{src}, v_{ev}, v_x, v_{dst}, v_{cd}, v_{ev}\mathscr{D}'$$

Thus, for the first premise, we apply (L-SKIP-t) three times to skip past the first three types in $\mathscr{D}$. We must now show

$$A; \pi \models B : v_{src}, v_{ev}, v_x, v_{dst}, v_{cd}, v_{ev}, \mathscr{D}' : A'$$

Again, from type consistency, we can satisfy the first two premises of (L-ENTER). We now must show

$$v_{src}; A; \pi \overset{v_x, v_{dst}, \mathscr{D}'}{\Rightarrow} A'; B$$

The first premise of this judgment (TX) is satisfied by the assumptions of this case. For the second premise, we use inversion on the first premise of (L-ENTER) to establish that $v_{src} = \text{new } v_{src'}$. The third premise is given by (A2). The fourth premise is interesting. We have to show that the current state of the automaton instance v_{src} in the program is the same as the current state A in the trace-acceptance relation. However, from our strengthened induction hypothesis, we have that $s; d; S; \cdot \models A : (v', Instance^{src})$. Additionally, from the second premise of (L-ENTER) we have that v_{src} is an instance of the π class and from (A5) we have that e is π-free. Thus, we can conclude that $v_{src} = v = \text{new } v'$. The fifth and sixth premises are constructed to enable showing that the final premise of (TX) can be satisfied. In particular, a straightforward induction on the length of $\mathscr{D}'$, and by relying on type consistency, we can easily show that (TX) is satisfiable.

The more interesting goal is to show to re-establish the premises of this theorem so

as to be able to apply the induction hypothesis for the second premise of (L-TR). This will follow from the type-soundness result for λ_{AIR}.

From inversion of the reduction relation, we have :

$$\frac{B : \mathscr{D}, v \rightsquigarrow v' \in M \qquad l_1 = B : \mathscr{D}, v}{M \vdash \mathscr{E} \cdot [\![B]\!]^{\mathscr{D}} \; v \xrightarrow{l_1} \mathscr{E} \cdot v'} \text{ (E-B1)}$$

From type consistency of M and S, we have that the type of e is determined by either (T-BODY) or (R-BODY). I.e., e has type

$$(¡Instance^{src} \times ¡Instance^{dst})$$

or

$$(¡Instance^{src} \times ¡Instance^{dst} \times Protected \; t_x \; dst).$$

I.e., $v' = Pair \; (v'_{src}) \; (v'')$. From (A5) we have

$$src{::}N, dst{::}N, s : ¡Instance^{src} \vdash e : t; dst$$

We can use the substitution lemma, relying on the fact that s is an affine assumption, to establish

$$src{::}N, dst{::}N \vdash (s \mapsto v)e : t; dst \uplus src$$

.

Now, from subject reduction, we have

$$src{::}N, dst{::}N \vdash \mathscr{E} \cdot v' : t; dst \uplus src$$

where

$$src{::}N \vdash v'_{src} : ¡Instance^{src}; src$$

Now, applying the converse of the substitution lemma, we can also establish

$$\Gamma; s \vdash \mathscr{E} \cdot Pair \; (s) \; (v'') : t; dst$$

which is the form of (A5) necessary to apply the induction hypothesis. To conclude, we must also establish that assumption (A6')—i.e., that $v'_{src} = \text{new } v''_{src}$ where $s; d; S; \cdot \models A' : (v''_{src}, t'_A)$. However, this follows from the semantic consistency of M with respect to π, as established by (A3). $\qquad \square$

C. Proofs of Theorems Related to FLAIR

C.1 Soundness of FLAIR

Before proceeding to the proof of soundness of FLAIR, we need to address one technicality that we glossed over in Chapter 4. Our formulation of models M and domain equations E for λAIR (in Appendix B) was in a purely functional setting. That is, the model for base terms restrict the operational behavior of these base terms to be free of side effects. Given that FLAIR is not purely functional, and given that we have require base terms like update (Figure 4.5) to mediate the mutation of secure memory locations, we need to lift this restriction on models.

The top of Figure C.1 revises the syntax of models to include side effects. Equations E now relate a pair consisting of an input store Σ and a sequence of values and types $\mathscr{D}$, to an output store Σ' and an expression e. As before, models M are a potentially infinite list of these equations. Certificates are unchanged.

Figure C.1 shows a revision to the four rules pertaining to the reduction of base terms. In the premises of (E-B1) and (E-B3), we now ensure that when reducing a base term application, both the store Σ and domain $\mathscr{D}$ of the equation E match the context in the conclusion. We then thread the store, Σ', and the result of the equation, e, to the conclusion. (E-B2) and (E-B4) are as before, except that the premises now refer to the modified syntax of equations.

Definition 46 (Store typing). *An environment Γ models a store Σ (written $\Gamma \models \Sigma$) if and only if all of the following are true.*

1. *$\ell \in dom(\Gamma) \iff \ell \in dom(\Sigma)$*

2. *$\forall \ell \in dom(\Gamma).\exists t, l.\Gamma(\ell) \in \{LabeledRef\,(ref\,t)\,l, ref\,t\}$*

3. *$\forall \ell \in dom(\Gamma).\exists t, l.\Gamma(\ell) \in \{LabeledRef\,(ref\,t)\,l, ref\,t\} \Rightarrow \Gamma \vdash_{\mathsf{term}} \Sigma(\ell) : t; \cdot$*

We extend type consistency of a model an signature to incorporate store typing in the obvious way; namely, the output store in every equation in the model must be typeable in the initial typing context Γ.

Theorem 47 (Progress). *Given all of the following:*

(A1) A well-formed environment Γ, consisting of a signature S, type names, and memory locations
$$\Gamma = S, \alpha_1::\mathbb{N}, \ldots, \alpha_n::\mathbb{N}, \ell_1:t_1, \ldots, \ell_m:t_m$$

$$\begin{array}{llll}
\text{equation} & E & ::= & (\Sigma, \mathcal{D}) \rightsquigarrow (\Sigma', e) \\
\text{eqn. domain} & \mathcal{D} & ::= & v \mid t \mid \mathcal{D}, \mathcal{D} \mid \cdot
\end{array}
\qquad
\begin{array}{llll}
\text{model} & M & ::= & B : \overrightarrow{E} \mid M, M \\
\text{certificates} & e & ::= & \ldots \mid [\![B]\!]^{\mathcal{D}}
\end{array}$$

$M \vdash (\Sigma, e) \overset{l}{\longrightarrow} (\Sigma', e')$ $\hfill$ Dynamic semantics

$$\frac{(\Sigma, \mathcal{D}, v) \rightsquigarrow (\Sigma', e) \in \overrightarrow{E} \qquad l = B : \mathcal{D}, v}{M, B : \overrightarrow{E}, M' \vdash (\Sigma, [\![B]\!]^{\mathcal{D}} \, v) \overset{l}{\longrightarrow} (\Sigma', e)} \text{ (E-B1)}$$

$$\frac{(\Sigma, \mathcal{D}, v) \rightsquigarrow (\Sigma', e) \notin \overrightarrow{E}}{M, B : \overrightarrow{E}, M' \vdash (\Sigma, [\![B]\!]^{\mathcal{D}} \, v) \longrightarrow (\Sigma, [\![B]\!]^{\mathcal{D}, v})} \text{ (E-B2)}$$

$$\frac{(\Sigma, \mathcal{D}, t) \rightsquigarrow (\Sigma', e) \in \overrightarrow{E} \qquad l = B : \mathcal{D}, t}{M, B : \overrightarrow{E}, M' \vdash (\Sigma, [\![B]\!]^{\mathcal{D}} \, [t]) \overset{l}{\longrightarrow} (\Sigma', e)} \text{ (E-B3)}$$

$$\frac{(\Sigma, \mathcal{D}, t) \rightsquigarrow (\Sigma', e) \notin \overrightarrow{E}}{M, B : \overrightarrow{E}, M' \vdash (\Sigma, [\![B]\!]^{\mathcal{D}} \, [t]) \longrightarrow (\Sigma, [\![B]\!]^{\mathcal{D}, t})} \text{ (E-B4)}$$

Figure C.1: Dynamic semantics of FLAIR, revises semantics of λAIR in Figure B.3

A2 A type correct expression e such that $\Gamma; \cdot \vdash_{\mathsf{term}} e : t; \varepsilon$, for some t and ε.

A3 An interpretation M such that M and S are type consistent.

A4 A store Σ such that $\Gamma \models \Sigma$.

Then, $\exists e', \Sigma'. M \vdash (\Sigma, e) \overset{l}{\longrightarrow} (\Sigma', e')$ or e is a value.

Proof. By induction on the structure of (A2), as in the proof of Theorem 31. We only need to consider the new cases of the typing judgment, i.e., those that appear in Figure 4.1.

Case (T-LOC): Locations ℓ are values.

Case (T-DREF): If we have !e, where e is not a value, then reduction can proceed by the congruence (E-CTX) since !$\bullet$ is valid context. If we have !v, by canonical forms, we have that $\exists \ell . v = \ell$. Thus, evaluation can proceed via (E-DEREF), the premise of which is satisfied from the first clause of the definition of store typing.

Case (T-ASN): Evaluation contexts include $\bullet := e$ and $v := \bullet$ and in both cases reduction can proceed using (E-CTX). So, we are only concerned with the case $v_1 := v_2$. By inversion of (A2), we have that v_1 has a reference type and, again, from canonical forms, we have that v_1 is a location ℓ. This is sufficient to show that reduction can proceed via (E-ASN), where the premise follows from the first clause of store typing.

As for base term applications, (E-B1) through (E-B4) remain a complete set of reduction rules, as before. $\hfill \square$

Theorem 48 (Preservation). *Given all of the following:*

1. *A well-formed environment Γ, consisting of a signature S, type names, and memory locations*

$$\Gamma = S, \alpha_1::\mathtt{N}, \ldots, \alpha_n::\mathtt{N}, \ell_1:t_1, \ldots, \ell_m:t_m$$

2. *A type correct expression e such that $\Gamma; \cdot \vdash_{\mathsf{term}} e : t; \varepsilon$, for some t and ε.*

3. *A model M such that M and S are type consistent.*

4. *A store Σ such that $\Gamma \models \Sigma$.*

If $M \vdash (\Sigma, e) \overset{l}{\longrightarrow} (\Sigma', e')$ then $\Gamma; \cdot \vdash_{\mathsf{term}} e' : t; \varepsilon$ and $\Gamma \models \Sigma'$.

Proof. By induction on the structure of (A2), as in the proof of Theorem 43. We only need to consider the new cases of the typing judgment, i.e., those that appear in Figure 4.1.

(Note that in order for the substitution lemma to still hold true, we crucially need to preserve the invariant of Proposition 34, namely that effect-free typings always result in U-kinded types. This follows from our requirement that all locations are given U-kind according to the rule (K-REF).)

Case (T-LOC): Locations ℓ are values; so this case is vacuously true.

Case (T-DEREF): The interesting case is when we have $M \vdash (\Sigma, !\ell) \longrightarrow (\Sigma, v)$, using (E-DEREF), where $\sigma(\ell) = v$. The result is immediate from the second and third clauses of the definition of store typing.

Case (T-ASN): The interesting case is when we have $M \vdash (\Sigma, \ell := v) \longrightarrow (\Sigma', ())$ using (E-ASN). Preservation of types is trivial since we have that (A2) concludes with the type *Unit*. Showing that $\Gamma \models \Sigma'$ is also straightforward, since from the premise of (E-ASN) we have that Σ' differs from Σ only in the location ℓ. And, from the second premise of (A2), we have that $\Gamma; \cdot \vdash v : t; \varepsilon_2$, where t is the type of the referent of ℓ. But, to show that the third clause of store typing is satisfied, we need to show that $\varepsilon_2 = \cdot$. From (K-REF) we have that $\Gamma \vdash t::\mathtt{U}$; thus, from the converse of Proposition 35 (Inversion of non-empty name constraints), we find that ε_2 must be empty.

As previously, the preservation of types by (E-B1) through (E-B4) follows directly from the type consistency of the model M and signature S. □

C.2 Correctness of Static Information Flow

We give a direct proof of noninterference for the encoding of Figure 4.5. The proof uses Pottier and Simonet's "bracket" technique to represent a pair of program executions. Additionally, we use an instrumented operational semantics to prove that the affine program counter capabilities are never duplicated.

Syntax of FLAIR[i]

expressions	e	::=	pop; $e \mid \ldots$
pc stack	pc	::=	$High \mid Low \mid \mathsf{pc}_1 \sqcup \mathsf{pc}_2$
tracked locations	$\hat{\ell}$	::=	$\ell_{PC} \mid \ell_{cap}$
assumptions	A	::=	$x \mid \hat{\ell} \mid A_1, A_2$
environment	Γ	::=	$\ell{:}t, \Gamma \mid x{:}t, \Gamma \mid \hat{\ell}{:}t, \Gamma$
store	Σ	::=	$(\ell, v), \Sigma \mid (\ell_{PC}{:}\mathsf{pc}) \mid (\ell_{cap}, (v, \mathsf{pc}))$

Signature extensions

$$label \; : \; \forall \alpha {::} \mathtt{U}.\alpha \rightarrow (l{:}Lab) \rightarrow (Labeled \; \alpha \; l)$$
$$lref \; : \; \forall \alpha {::} \mathtt{U}.\mathrm{ref}\alpha \rightarrow (l{:}Lab) \rightarrow (LabeledRef \; (\mathrm{ref} \; \alpha) \; l)$$

Extensions to typing judgment

$$\frac{}{\Gamma; \hat{\ell} \vdash_\varphi \hat{\ell} : \Gamma(\hat{\ell}); \cdot} \;\; \text{(T-PCAP)} \qquad \frac{\Gamma_{\mathsf{pc}}; A \vdash_\varphi e : t; \cdot}{\Gamma_{\mathsf{pc} \sqcup v}; A \vdash_\varphi \mathsf{pop}; e : t; \cdot} \;\; \text{(T-POP)}$$

where Γ_{pc} *defined by*	$(\Gamma, \ell_{PC}{:}PC \; l, \Gamma')_{\mathsf{pc}}$	$\equiv$	$\Gamma, \ell_{PC}{:}PC \; [\![\mathsf{pc}]\!], \Gamma'$
	$(\Gamma, \ell_{cap}{:}Cap \; l \; m, \Gamma')_{\mathsf{pc}}$	$\equiv$	$\Gamma, \ell_{cap}{:}Cap \; l \; [\![\mathsf{pc}]\!], \Gamma'$
and $[\![\mathsf{pc}]\!]$ *defined by*	$[\![v]\!]$	$\equiv$	v
	$[\![\mathsf{pc} \sqcup v_1]\!]$	$\equiv$	$v \;\; where \;\; (M \vdash lub \; [\![\mathsf{pc}]\!] \; v_1 \longrightarrow^* v)$

$M \vdash (\Sigma, e) \longrightarrow (\Sigma, e')$ Instrumented operational semantics

$$\frac{\Sigma = \Sigma'', (\ell_{PC}, \mathsf{pc} \sqcup v) \qquad \Sigma' = \Sigma'', (\ell_{PC}, \mathsf{pc})}{M \vdash (\Sigma, \mathsf{pop}; e) \longrightarrow (\Sigma', e)} \;\; \text{(E-POP)}$$

Model M_{Flow} Base term reductions

$pc2cap$: $(\Sigma, (\ell_{pc}, \mathsf{pc}); l, m, \ell_{pc}) \longrightarrow (\Sigma, (\ell_{cap}, ([\![l \sqcup m]\!], \mathsf{pc})); \ell_{cap})$

$cap2pc$: $(\Sigma, (\ell_{cap}, (l, \mathsf{pc})); l, m, \ell_{cap}) \longrightarrow (\Sigma, (\ell_{PC}, \mathsf{pc}); \ell_{PC})$

$update$: $(\Sigma, (\ell, v); t, l, m, \ell_{cap}, [\![lref]\!]^{\ell, l}, v') \longrightarrow (\Sigma, (\ell, v'); (cap2pc \; l \; m \; \ell_{cap}, ()))$

$deref$: $(\Sigma, (\ell, v); t, l, [\![lref]\!]^{\ell, l}) \longrightarrow (\Sigma, (\ell, v); [\![label]\!]^{v, l})$

$$branch \; : \; \frac{b_i \in \{t, f\} \qquad e = (\lambda x{:}\_.(fst \; x, [\![label]\!]^{(snd \; x), m}))(b_i \; (\ell_{PC}, ()))}{(\Sigma, (\ell_{PC}, \mathsf{pc}); t_\alpha, l, \ell_{PC}, m, b, t, f) \longrightarrow (\Sigma, (\ell_{PC}, \mathsf{pc} \sqcup l); (\lambda x{:}\_.\mathsf{pop}; x)e)}$$

$$apply \; : \; \frac{e = (\lambda x{:}\_.(fst \; x, [\![label]\!]^{(snd \; x), m}))(g \; (\ell_{PC}, x))}{(\Sigma, (\ell_{PC}, \mathsf{pc}); t_\alpha, t_\beta, l, \ell_{PC}, m, [\![label]\!]^{g, m}, x) \longrightarrow (\Sigma, (\ell_{PC}, \mathsf{pc} \sqcup l); (\lambda x : \_.\mathsf{pop}; x)e)}$$

Figure C.2: Instrumenting FLAIR to track affine capabilities and program counter tokens

C.2.1 Affinity of Program Counters and Capabilities

Figure C.2 defines an instrumented semantics for FLAIR, in order to show that affine program counter tokens and capabilities are never duplicated by the program. The general strategy is to represent these affine values using special "tracked locations", $\hat{\ell}$, in the store. ℓ_{PC} is a location that stores the current value of the program counter label; ℓ_{cap} stores the current capability of the program to update a normal labeled ref-cell. The dynamic semantics of the program include specific equations for base terms in the model to update the contents of these locations. For example, the *pc2cap* function de-allocates the ℓ_{PC} location and allocates ℓ_{cap} location; *cap2pc* does the reverse. If the type system of FLAIR properly tracks affine assumptions, then after the program calls the *pc2cap* function, the ℓ_{PC} value must have been consumed, and therefore, must not appear anywhere in the program text. To show that this is in fact the case, we prove a subject reduction property for the instrumented calculus. The consequence is that affine program counter assumptions are never duplicated by the program.

Our runtime semantics are also designed to track the program counter label accurately. For example, consider the equation of the *branch* base term. The domain contains a store in which the program counter ℓ_{PC} contains a program counter label pc. The argument t_α is the type used to instantiate the quantified variable in the type of *branch*. The label arguments are l and m. The program counter value passed in by the program is ℓ_{PC}. Then we have the boolean value in the guard b, and the true and false branches t and f. In the range of the equation, notice that the ℓ_{PC} location has been updated to contain the label pc $\sqcup l$, where l is the label of the boolean b. This reflects the fact that the branch e that is evaluated is control-dependent on the boolean; so, the program counter location is "bumped up" to be at least as secret as b.

However, when the branch e has evaluated, the continuation is no longer control dependent on b. So, we need to lower the program counter before returning a value from the branch expression. To model this, we introduce a new syntactic construct (pop; e). The operational semantics of this construct is defined by (E-POP)—it simply pops the top-most label from the stack of program counter labels stored in ℓ_{PC} and then evaluates e. The equation for *branch* carefully uses this construct to lower the program counter only after the branch e has completed evaluating. After the program counter stack has been popped, *branch* returns value returned by the branch. The *apply* function is similar to *branch*.

Strictly speaking, updating the program counter values to reflect the control dependences of the expression being evaluated is unnecessary to show that affine values are tracked properly in FLAIR. However, in the next subsection, when we prove noninterference by modeling a pair of executions in the syntax of a single FLAIR[2] program, this dependence tracking will become important. So, we introduce the dependence tracking in this simpler setting to make the extension to the FLAIR[2] easier.

Finally, in order to give a term representation for labeled values and labeled references we extend the signature with two functions *label* and *lref*. These facilitate the formal proof, but are not otherwise required at the source level.

Definition 49 (Extended store typing). $\Gamma = \Gamma_1, \hat{\ell}{:}t, \Gamma_2$ *models a store* $\Sigma = \Sigma_1, (\hat{\ell}, b), \Sigma_2$, *if*

and only if, $\Gamma_1, \Gamma_2 \models \Sigma_1, \Sigma_2$, *and*

- *If* $\hat{\ell} = \ell_{PC}$ *and* $b = \mathsf{pc}$, *then* $t = PC$ [[pc]].

- *If* $\hat{\ell} = \ell_{cap}$ *and* $b = (v, \mathsf{pc})$, *then* $t = Cap\ v$ [[pc]].

We overload notation so that $\Gamma \models \Sigma$ *can stand for extended store typing according to the context.*

Proposition 50 (Guarded strengthening). *For well-formed* $\Gamma = \Gamma_1, \hat{\ell}{:}t, \Gamma_2$, *if* $\Gamma; \cdot \vdash_\varphi e : t; \varepsilon$, *then* $\Gamma_1, \Gamma_2; \cdot \vdash_\varphi e : t; \varepsilon$

Proposition 51 (Insignificant affine assumptions are unused). *Given* $\Gamma \vdash t_x{::}\mathsf{U}$ *and* $\Gamma \vdash t_y{::}\mathsf{U}$ *and* $t_x \not\cong t_y$, *such that* $\Gamma, x{:}t_x$ *and* $\Gamma, x{:}t_y$ *are both well-formed. Suppose,* $\Gamma, x{:}t_x; A, x \vdash_\varphi e : t; \cdot$ *and* $\Gamma, x{:}t_y; A, x \vdash_\varphi e : t; \cdot$. *Then,* $\Gamma, x{:}t_x; A \vdash_\varphi e : t; \cdot$.

Lemma 52 (Witnesses for changes to tracked locations). *For well-formed* Γ, A *such that* $\Gamma \models \Sigma$, $\Gamma; A \vdash_\varphi e : t; \varepsilon$, *and* $M \vdash (\Sigma, e) \longrightarrow (\Sigma', e')$ *and* $\Gamma \not\models \Sigma'$. *Then,* $A = \hat{\ell}$.

Proof. By induction on the structure of the reduction relation. The interesting base cases are the base-term reductions due to *pc2cap*, *cap2pc*, *branch*, and *apply*, since these are the only ones that change tracked locations. However, in each of these cases, from inspection, it is clear that $\hat{\ell} \in FV(e)$ and, thus, $A = \hat{\ell}$. $\qquad\square$

Lemma 53 (Accuracy of program counter tracking). *If, for* Γ *well-formed, the signature* S_{Flow} *and model* M_{Flow}, *we have each of the following:*

(A1) $\hat{\ell} \in dom(\Gamma) \Rightarrow A(\Gamma) \subseteq \hat{\ell}$

(A2) $\Gamma; A(\Gamma) \vdash_\varphi e : t; \cdot$

(A3) $\Gamma \models \Sigma$, *and* $dom(\Gamma) = dom(\Sigma) \cup \hat{\ell}$

(A4) $M_{Flow} \vdash (\Sigma, e) \longrightarrow (\Sigma', e')$

(A5) $\Gamma' \models \Sigma'$

Then, $\Gamma'; A(\Gamma') \vdash_\varphi e' : t; \cdot$

Proof. We proceed by induction on (A2).

Case (T-PCAP): $\hat{\ell}$ is a value.

Case (T-POP): By inversion, we have that (A4) is an application of (E-POP), where $\Sigma(\ell_{PC}) = \mathsf{pc} \sqcup v$. We have to show that $\Gamma'; A \vdash_\varphi e : t; \cdot$, where $\Gamma' \models \Sigma'$, where $\Sigma'(\ell_{PC}) = \mathsf{pc}$. That is, $\Gamma'(\ell_{PC}) = PC$ [[pc]] $= \Gamma_{\mathsf{pc}}(\ell_{PC})$, and since the $dom(\Sigma) = dom(\Sigma')$, we have $A(\Gamma') = A(\Gamma)$. But, we have exactly this from the premise of (A2).

Case (T-LOC): ℓ is a value.

Case (T-DEREF), (T-ASN), (T-B): Follows from the corresponding case of subject reduction of FLAIR.

Case (T-X), (T-XA): From (A3), open terms are not permissible.

Case (T-ABS): Abstractions are values.

Case (T-APP):

Sub-case $e = e_1 e_2$: Then, from inversion, we have (A4) is an instance of (E-CTX) with $M \vdash (\Sigma, e_1 e_2) \longrightarrow (\Sigma', e_1' e_2)$, and $M \vdash (\Sigma, e_1) \longrightarrow (\Sigma', e_1')$ in the premise (A4.1).

From the premise of (T-APP) we have (A2.1) $\Gamma; A \vdash_\varphi e_1 : t_1; \cdot$ and (A2.2) $\Gamma; A' \vdash_\varphi e_2 : t_2; \cdot$, where $A, A' = A(\Gamma)$. Now, from the induction hypothesis applied to (A2.1) and (A4.1), we have $\Gamma'; A(\Gamma') \vdash_\varphi e_1' : t_1; \cdot$, where $\Gamma' \models \Sigma'$.

If $\Gamma' = \Gamma$ then we are done with an application of (T-APP), using the induction hypothesis for the first premise and an unchanged (A2.2) for the second premise. However, if $\Gamma' \neq \Gamma$, we need to be more careful. In this case, notice that Γ and Γ' can only differ in assumptions about the affine locations $\hat{\ell}$. That is, either $\ell_{PC} \in \text{dom}(\Gamma)$ and $\ell_{cap} \in \text{dom}(\Gamma')$, or vice-versa; or, $\text{dom}(\Gamma) = \text{dom}(\Gamma')$ but, $\Gamma'(\hat{\ell}) \neq \Gamma(\hat{\ell})$. In either event, since we have $A, A' = \hat{\ell}$, we aim to use Proposition 50 (below) to reconstruct the typing derivation.

Sub-sub-case $A = \hat{\ell}$, $A' = \cdot$: In this case, we have (A2.2) $\Gamma; \cdot \vdash_\varphi e_2 : t_2; \cdot$, where $\Gamma = \Gamma_1, \hat{\ell}{:}t, \Gamma_2$. From guarded strengthening, we can conclude $\Gamma_1, \Gamma_2; \vdash_\varphi e_2 : t_2; \cdot$. But, $\Gamma' = \Gamma_1, \hat{\ell}'{:}t', \Gamma_2$. So, from weakening, we get $\Gamma'; \cdot \vdash_\varphi e_2 : t_2; \cdot$. We can therefore construct the derivation using (T-APP) with the induction hypothesis for the first premise and this latter derivation as the second premise.

Sub-sub-case $A = \cdot$, $A' = \hat{\ell}$: Since $A = \cdot$, we can conclude that $\hat{\ell} \notin FV(e_1)$. However, we still have $\Gamma' \neq \Gamma$. From Lemma 52, this is impossible.

Sub-case $e = v\, e_2$: Similar to the previous sub-case.

Sub-case $e = v_1 v_2$: Soundness of β-reduction via (E-APP) follows from the soundness proof of FLAIR. The interesting case here is when reduction proceeds using a base term reduction via (E-B2). Usually, we argue for the soundness of these cases from the type-consistency of the model and signature. However, in this case, with the instrumented semantics, the well-typedness of the reduction relies on the updates to the tracked locations in the store. So, we enumerate each case of the base-term reductions here.

Sub-sub-case *pc2cap*: On the LHS, we have $e = [\![pc2cap]\!]^{l,m} \ell_{PC}$, typeable in $\Gamma_{\text{pc}} \models \Sigma, (\ell_{PC}, \text{pc})$ as $Cap\ (lub\ l\ m)\ [\![pc]\!]$, (where $[\![pc]\!] = m$). On the RHS, we have ℓ_{cap} typeable in $\Gamma' \models \Sigma, (\ell_{cap}, ([\![l \sqcup m]\!], \text{pc}))$ as $Cap\ (lub\ l\ m)\ [\![pc]\!]$, according the definition of extended store typing and the definition of $[\![pc]\!]$.

Sub-sub-case *cap2pc*: Similar to the previous case.

Sub-sub-case *update*: On the LHS, we have $e = [\![update]\!]^{t,l,m,\ell_{cap}, [\![lref]\!]^{\ell,l}} v'$, which, from S_{Flow}, is typeable as Boxed m *Unit*. The type correctness of the RHS is immediate from inspection, and store typing follows from the well-formedness of Γ.

Sub-sub-case *deref*: Like, *update*, this is straightforward from inspection.

Sub-sub-case *branch*: In the LHS, we have each branch

$$e{:}(\text{Boxed lub l m } \textit{Unit} \rightarrow \text{Boxed lub l m } t_\alpha)$$

with $[\![pc]\!] = l$, following from the signature of *branch* and the well-typedness of ℓ_{PC}. We

have to prove the type-correctness of the RHS in a context $\Gamma' \models \Sigma, (\ell_{PC}, \mathsf{pc} \sqcup m)$. In the RHS, we have first an application of e to $(\ell_{PC}, ())$, which, since $[\![\mathsf{pc} \sqcup m]\!] = (lub\ l\ m)$, we have the application typeable in Γ' using (T-APP) as $\mathsf{Boxed\ lub\ l\ m}\ t_\alpha$. For the lambda term, the body $\mathsf{pop}; e'$, where $e' = (fst\ x, [\![label]\!]^{(snd\ x),m})$, we can apply (T-POP), and type e' as $\mathsf{Boxed\ l}\ (Labeled\ t_\alpha\ m)$, as required by the signature.

Sub-sub-case *apply*: Identical to the branch sub-sub-case, except using t_β instead of *Unit*.

Case (T-CAP): Identical to the first sub-case of (T-APP), noting that affine assumptions are split between e and e'. The conclusion follows, as before, from a use of Proposition 50 and weakening, or Lemma 52.

Case (T-CASE): Similar to the first sub-case of (T-APP), noting that the affine assumptions are split between the guard and between each of the mutually exclusive branches. The conclusion follows, as before, from a use of Proposition 50 and weakening, or Lemma 52.

Case (T-B1), ... (T-B3), (T-CONV): All follow from the soundness of FLAIR.

$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

C.2.2 Proving Noninterference using FLAIR^2

Definition 54 (Well-formed for information flow). *A well-formed environment Γ is well formed for information flow if and only if $\forall \ell \in dom(\Gamma).\exists t, l.\Gamma(\ell) \in LabeledRef\ (ref\ t)\ l$*

Definition 55 (Typing for bracketed stores). *A well-formed environment Γ models a bracketed store Σ, if and only if, $\forall i \in \{1,2\}$, $\lfloor \Gamma \rfloor_i \models \lfloor \Sigma \rfloor_i$. Additionally, $\Sigma(\ell) = \langle v_1 \parallel v_2 \rangle \Rightarrow High \triangleleft \Gamma(\ell)$. We write $\Gamma \models \Sigma$ for typing of bracketed stores.*

Definition 56 (Closure of projections). *We write $\lfloor e \rfloor_i^*$ to denote the largest closure of the relation $\lfloor e \rfloor_i$ over the structure of e, with the restriction that*

$$\lfloor splitpc\ t\ l\ e \rfloor_i^* \equiv (\lambda x{:}t.\mathsf{pop}; x)(\lfloor e \rfloor_i^*)$$

Proposition 57 (Equivalence of sub-reductions). *If $\mathsf{M}_{Flow} \vdash (\Sigma, e) \longrightarrow_i (\Sigma', e')$, then $M_{Flow} \vdash (\lfloor \Sigma \rfloor_i, e) \longrightarrow (\lfloor \Sigma' \rfloor_i, e')$, and vice versa.*

Lemma 58 (Adequacy of FLAIR^2). *Given an environment Γ well-formed for information flow, such that each of the following are true:*

(A1) $\hat{\ell} \in dom(\lfloor \Gamma \rfloor_i) \Rightarrow \lfloor A(\Gamma) \rfloor_i \subseteq \hat{\ell}$

(A2) $\Gamma; A(\Gamma) \vdash_{\mathsf{term}, \iota} e : t; \cdot$

(A3) *A store Σ such that $\Gamma \models \Sigma$*

(A4) $\exists i.M_{Flow} \vdash (\lfloor \Sigma \rfloor_i^*, \lfloor e \rfloor_i^*) \longrightarrow (\Sigma_i, e_i)$

Then, $\mathsf{M}_{Flow} \vdash (\Sigma, e) \longrightarrow^* (\Sigma', e')$ and $\lfloor \Sigma' \rfloor_i^* = \Sigma_i$ and $\lfloor e' \rfloor_i^* = e_i$.

Syntax of $\textsc{Flair}^2$, as an extension of $\textsc{Flair}^\textsf{i}$

expr.	e	$::=$	$\langle e_1 \parallel e_2 \rangle \mid \ldots$	assumpt.	A	$::=$	$\langle \hat{\ell}_1 \parallel \hat{\ell}_2 \rangle \mid \ldots$
values	v	$::=$	$\langle v_1 \parallel v_2 \rangle$	env.	Γ	$::=$	$\langle \hat{\ell}_1{:}t_1 \parallel \hat{\ell}_2{:}t_2 \rangle \mid \ldots$
index	ι	$::=$	$\cdot \mid 1 \mid 2$	store	Σ	$::=$	$\langle (\hat{\ell}, b) \parallel (\hat{\ell}, b) \rangle \mid \ldots$

Signature extension

$$splitpc : \forall \alpha{::}\textsc{U}.(l{:}Lab) \rightarrow (m{:}Lab) \rightarrow Boxed\ (lub\ l\ m)\ \alpha \rightarrow Boxed\ l\ \alpha$$

Extensions to typing judgment

$$\frac{\forall i \in \{1,2\} \quad \lfloor \Gamma \rfloor_i\ ;\ \lfloor A \rfloor_i \vdash_{\varphi,i} e_i : t; \cdot \quad (\lfloor \Gamma \rfloor_i)_{High}\ ;\ \lfloor A \rfloor_i \vdash_{\varphi,i} e_i : t; \cdot \quad High \vartriangleleft t}{\Gamma; A \vdash_{\varphi,\cdot} \langle e_1 \parallel e_2 \rangle : t; \cdot} \text{(T-BRAC)}$$

$$\frac{\forall i. \lfloor \Gamma \rfloor_i (\ell_{PC}) = PC\ [\![\mathsf{pc} \sqcup v]\!] \quad \Gamma' = \Gamma[\ell_{PC}{:}PC\ [\![\mathsf{pc}]\!]] \quad \Gamma'; A \vdash_{\varphi,\cdot} e : t; \cdot}{\Gamma; A \vdash_{\varphi,\cdot} \mathsf{pop}; e : t; \cdot} \text{(T-POP)}$$

$$High \vartriangleleft Labeled\ t\ High \qquad\qquad High \vartriangleleft LabeledRef\ (\mathrm{ref}\ t)\ High$$

$$\frac{High \vartriangleleft t}{High \vartriangleleft Labeled\ t\ l} \qquad\qquad \frac{High \vartriangleleft t}{High \vartriangleleft Boxed\ l\ t}$$

$$\frac{i \in \{1,2\}}{\lfloor \langle e_1 \parallel e_2 \rangle \rfloor_i \equiv e_i} \qquad \frac{i \in \{1,2\}}{\lfloor \langle \hat{\ell}_1{:}b_1 \parallel \hat{\ell}_2{:}b_2 \rangle \rfloor_i \equiv \hat{\ell}_i{:}b_i} \qquad \lfloor x \rfloor_\cdot \equiv x$$

$M_{\text{Flow}} \vdash (\Sigma, e) \longrightarrow_\iota (\Sigma', e')$ Instrumented operational semantics

$$\frac{\{i,j\} = \{1,2\} \quad M \vdash (\Sigma, e_i) \longrightarrow_i (\Sigma', e_i') \quad e_j = e_j'}{M_{\text{Flow}} \vdash (\Sigma, \langle e_1 \parallel e_2 \rangle) \longrightarrow (\Sigma', \langle e_1' \parallel e_2' \rangle)} \text{(E-BRAC)}$$

$$\frac{\forall i. \lfloor \Sigma \rfloor_i (\ell_{PC}) = \mathsf{pc} \sqcup v \quad \forall \ell. \Sigma(\ell) = \Sigma'(\ell) \quad \Sigma'(\ell_{PC}) = \mathsf{pc}}{M_{\text{Flow}} \vdash (\Sigma, \mathsf{pop}; e) \longrightarrow (\Sigma', e)} \text{(E-POP)}$$

Figure C.3: Semantics of $\textsc{Flair}^2$

Proof. Without loss of generality, we assume $i = 1$. Proof by induction on the structure of (A2).

Case (T-BRAC): From (A4), we have $M_{\text{Flow}} \vdash (\lfloor \Sigma \rfloor_1^*, e_1) \longrightarrow (\Sigma, e_1)$. For the conclusion, we want to show that (E-BRAC) is applicable. But, the premise of (E-BRAC) is immediate from (A4) and Proposition 57.

Case (T-POP): Here, from inversion, we have (A4) is an application of (E-POP), of the form

$$\frac{\Sigma = \Sigma'', (\ell_{PC}, \mathsf{pc} \sqcup v) \quad \Sigma' = \Sigma'', (\ell_{PC}, \mathsf{pc})}{M_{\text{Flow}} \vdash (\Sigma, \mathsf{pop}; e) \longrightarrow (\Sigma', e)}$$

$pc2cap$:
$$(\Sigma,(\ell_{pc},\mathsf{pc});l,m,\ell_{pc})\longrightarrow(\Sigma,(\ell_{cap},(\llbracket l\sqcup m\rrbracket,\mathsf{pc}));\ell_{cap})$$
$$(\Sigma,(\ell_{pc},\mathsf{pc});l,m,\ell_{pc})\longrightarrow_1(\Sigma,\langle(\ell_{cap},(\llbracket l\sqcup m\rrbracket,\mathsf{pc}))\parallel(\ell_{pc},\mathsf{pc})\rangle;\ell_{cap})$$
$$(\Sigma,\langle(\ell_{pc},\mathsf{pc})\parallel c\rangle;l,m,\ell_{pc})\longrightarrow_1(\Sigma,\langle\ell_{cap},(l,\mathsf{pc})\parallel c\rangle;\ell_{cap})$$

$cap2pc$:
$$(\Sigma,(\ell_{cap},(l,\mathsf{pc}));l,m,\ell_{cap})\longrightarrow(\Sigma,(\ell_{PC},\mathsf{pc});\ell_{PC})$$
$$(\Sigma,(\ell_{cap},(l,\mathsf{pc}));l,m,\ell_{cap})\longrightarrow_1(\Sigma,\langle(\ell_{PC},\mathsf{pc})\parallel(\ell_{cap},(l,\mathsf{pc}))\rangle;\ell_{PC})$$
$$(\Sigma,\langle(\ell_{cap},(l,\mathsf{pc}))\parallel c\rangle;l,m,\ell_{pc})\longrightarrow_1(\Sigma,\langle\ell_{pc},\mathsf{pc}\parallel c\rangle;\ell_{PC})$$

$update$:
$$(\Sigma,(\ell,v);t,l,m,\ell_{cap},\llbracket lref\rrbracket^{\ell,l},v')\longrightarrow(\Sigma,(\ell,v');(cap2pc\ l\ m\ \ell_{cap},()))$$
$$(\Sigma,(\ell,v);t,l,m,\ell_{cap},\llbracket lref\rrbracket^{\ell,l},v')\longrightarrow_1(\Sigma,(\ell,\langle v'\parallel\lfloor v\rfloor_2\rangle));(cap2pc\ l\ m\ \ell_{cap},()))$$

$$\frac{\Sigma(\ell_i)=v_i'\qquad\Sigma'=\Sigma[\ell_1=\langle\lfloor v\rfloor_i\parallel\lfloor v_2'\rfloor_2\rangle][\ell_2=\langle\lfloor v_1'\rfloor_1\parallel\lfloor v_2'\rfloor_2\rangle]}{(\Sigma;t,l,m,\ell_{cap},\langle\llbracket lref\rrbracket^{\ell_1,l}\parallel\llbracket lref\rrbracket^{\ell_2,l}\rangle,v)\longrightarrow(\Sigma',(cap2pc\ l\ m\ \ell_{cap},()))}$$

$deref$:
$$(\Sigma,(\ell,v);t,l,\llbracket lref\rrbracket^{\ell,l})\longrightarrow_t(\Sigma,(\ell,v);\llbracket label\rrbracket^{\lfloor v\rfloor_t,l})$$

$$(\Sigma,(\ell_1,v_1),(\ell_2,v_2);t,l,\langle\llbracket lref\rrbracket^{\ell_1,l}\parallel\llbracket lref\rrbracket^{\ell_2,l}\rangle)\longrightarrow$$
$$(\Sigma,(\ell_1,v_1),(\ell_2,v_2);\langle\llbracket label\rrbracket^{\lfloor v_1\rfloor_1,l}\parallel\llbracket label\rrbracket^{\lfloor v_2\rfloor_2,l}\rangle)$$

$branch$:
$$\frac{b\neq\langle v_1\parallel v_2\rangle\qquad e=(\lambda x:\_.(fst\ x,\llbracket label\rrbracket^{(snd\ x),m}))(g(\ell_{PC},()))\qquad g\in\{t,f\}}{(\Sigma,(\ell_{PC},\mathsf{pc});t_\alpha,l,\ell_{PC},m,b,t,f)\longrightarrow(\Sigma,(\ell_{PC},\mathsf{pc}\sqcup l);splitpc\ [t_\alpha]\ l\ m\ e)}$$

$$\frac{e_i=(\lambda x:\_.(fst\ x,\llbracket label\rrbracket^{(snd\ x),m}))(b_i(\ell_{PC},()))\qquad b_i\in\{t,f\}}{(\Sigma,(\ell_{PC},\mathsf{pc});t_\alpha,l,\ell_{PC},m,\langle v_1\parallel v_2\rangle,t,f)\longrightarrow}$$
$$(\Sigma,(\ell_{PC},\mathsf{pc}\sqcup m);splitpc\ [t_\alpha]\ l\ m\ \langle e_1\parallel e_2\rangle)$$

$apply$:
$$\frac{g\neq\langle g_1\parallel g_2\rangle\qquad e=(\lambda x:\_.(fst\ x,\llbracket label\rrbracket^{(snd\ x),m}))(g(\ell_{PC},x))}{(\Sigma,(\ell_{PC},\mathsf{pc});t_\alpha,t_\beta,l,\ell_{PC},m,\llbracket label\rrbracket^{g,m},x)\longrightarrow(\Sigma,(\ell_{PC},\mathsf{pc}\sqcup l);splitpc\ [t_\beta]\ l\ m\ e)}$$

$$\frac{e_i=(\lambda x:\_.(fst\ x,\llbracket label\rrbracket^{(snd\ x),m}))(f_i(\ell_{PC},\lfloor x\rfloor_i))}{(\Sigma,(\ell_{PC},\mathsf{pc});t_\alpha,t_\beta,l,\ell_{PC},m,\langle\llbracket label\rrbracket^{f_1,m}\parallel\llbracket label\rrbracket^{f_2,m}\rangle,x)\longrightarrow}$$
$$(\Sigma,(\ell_{PC},\mathsf{pc}\sqcup l);splitpc\ [t_\beta]\ l\ m\ \langle e_1\parallel e_2\rangle)$$

$splitpc$:
$$(\Sigma;t,l,m,(\ell_{PC},v))\longrightarrow(\Sigma;\mathsf{pop};(\ell_{PC},v))$$
$$(\Sigma;t,l,m,\langle(\ell_{PC},v_1)\parallel(\ell_{PC},v_2)\rangle)\longrightarrow(\Sigma;\mathsf{pop};(\ell_{PC},\langle v_1\parallel v_2\rangle))$$

Figure C.4: Dynamic semantics of $\textsc{Flair}^2$

where pop; $e = $ pop; $\lfloor e \rfloor_1^*$

For the conclusion, we apply (E-POP) in $\textsc{Flair}^2$. From the premise of (A2) and from (A3), we have that the store Σ satisfies the premise of (E-POP). That $\Sigma' = \lfloor \Sigma' \rfloor_1$ and $e = \lfloor e \rfloor_1^*$ follows from construction.

Case (T-LOC): Is a value in both $\textsc{Flair}^i$ and in $\textsc{Flair}^2$.

Case (T-DREF), (T-ASN), (T-B): All follow from the induction hypothesis.

Case (T-X), (T-XA): Only close terms can step in $\textsc{Flair}^i$.

Case (T-ABS): Irreducible.

Case (T-APP): If we have $e_1\ e_2$, and reduction in (A4) via (E-CTX), the the result follows from the induction hypothesis. If we have, $v_1\ v_2$, we can be sure that $v_1 \neq \langle v_1' \parallel v_1'' \rangle$, since, from the last premise of (T-BRAC), the latter term has a type like *Labeled t High*, which doesn't match the first premise of (T-APP). Of course, v_2 can be a bracketed term $\langle v_2' \parallel v_2'' \rangle$. If, reduction in (A4) is via (E-APP), then since $\langle v_2' \parallel v_2'' \rangle$ is a value the conclusion follows from (E-APP) in $\textsc{Flair}^2$. The equivalence of the stores and an expressions after substitution straightforward.

The interesting cases are when (A4) in $\textsc{Flair}^i$ is an instance of (E-B2) and relies on an equation in M_{Flow}. We enumerate the cases.

Sub-case *pc2cap*: If $\iota = \cdot$, then the first reduction rule in M_{Flow} is identical to the corresponding rule in M_{Flow}. If $\iota = 1$, then from (A4) we have $\lfloor \Sigma \rfloor_i(\ell_{PC}) = \text{pc}$, which can be fulfilled in one of two ways—i.e., either $\Sigma(\ell_{PC}) = \text{pc}$ or $\langle (\ell_{PC}, \text{pc}) \parallel c \rangle \in \Sigma(\ell_{PC})$. These cases are covered by each of the next two rules in M_{Flow}.

Sub-case *cap2pc*: Similar to the previous sub-case.

Sub-case *update*: If in (A2), we are typing an application of the form

$$\mathsf{e} = update\ \ldots\ [\![lref]\!]^{\ell,l}$$

then $\lfloor e \rfloor_1 = \mathsf{e}$, and the reduction (A4) in $\textsc{Flair}^i$ is matched by the first rule for *update* in M_{Flow}. If, however, we have $\mathsf{e} = update\ \ldots\ \langle v_1 \parallel v_2 \rangle$, then $\lfloor e \rfloor_1 = update\ \ldots\ v_1$. In this case, the reduction of (A4) is mimicked in $\textsc{Flair}^2$ by the second (lifting) rule of *update*, which allows reduction to proceed in both sub-executions.

Sub-case *deref*: Similar to the previous sub-case.

Sub-case *branch*: If we have in (A2) *branch* $\ldots\ b\ \ldots$, where $\lfloor b \rfloor_1 = b$, then in (A4), we have a reduction to $(\lambda x{:}_{-}.\text{pop}; x)(\lambda x : _{-}.(fst\ x, \ldots))(g(\ell_{PC}, ()))$. The first rule of *branch* in M_{Flow} handles exactly this case, with *splitpc* $\ldots$ in the RHS. From the definition of $\lfloor \cdot \rfloor_i^*$, we have the desired equivalence.

If, we have in (A2) *branch* $\ldots\ \langle v_1 \parallel v_2 \rangle\ \ldots$, then in (A4), we have on the RHS $(\lambda x{:}_{-}.\text{pop}; x)e_1$. The second rule of *branch* in M_{Flow} handles this case. On the RHS, we have $\mathsf{e}' = splitpc\ \ldots\ \langle e_1 \parallel e_2 \rangle$, and from the definition of $\lfloor \cdot \rfloor_i^*$, we get the desired equivalence.

Sub-case *apply*: Similar to *branch*.

Sub-case *splitpc*: If we have *splitpc* $\ldots\ v_1$ in (A2), then, by the definition of projection, in (A4) we have $(\lambda x{:}_{-}.\text{pop}; x)\ v_1$ which reduces by (E-APP) to pop; v. In $\textsc{Flair}^2$, we

have a reduction from the second rule of *splitpc* to the same term. In the case where in (A2) we have *splitpc* ... $\langle v_1 \parallel v_2 \rangle$, the result in (A4) is the same. In this case, the second rule of *splitpc* produces an RHS with an equivalent projection.

Case (T-CASE): As in (T-APP), the expression in the guard cannot be bracketed, since that would require it to have a labeled type.

Case (T-CAP): Similar.

Case (T-B1), ..., (T-B4), (T-CONV): Trivial. □

Lemma 59 (Subject reduction for FLAIR^2). *If, for Γ well-formed for information flow we have each of the following:*

(A1) $\hat{\ell} \in dom(\lfloor \Gamma \rfloor_i) \Rightarrow \lfloor A(\Gamma) \rfloor_i \subseteq \hat{\ell}$

(A2) $\lfloor \Gamma \rfloor_\iota ; \lfloor A(\Gamma) \rfloor_\iota \vdash_{\text{term},\iota} e : t; \cdot$

(A3) $\lfloor \Gamma \rfloor_\iota \models \lfloor \Sigma \rfloor_\iota$

(A4) $\mathsf{M}_{Flow} \vdash (\Sigma, e) \longrightarrow_\iota (\Sigma', e')$

(A5) $\lfloor \Gamma' \rfloor_\iota \models \lfloor \Sigma' \rfloor_\iota$

Then, $\lfloor \Gamma' \rfloor_\iota ; \lfloor A(\Gamma') \rfloor_\iota \vdash_{\text{term},\iota} e' : t; \cdot$

Proof. By induction on the structure of (A2). In the cases where $(\iota = \cdot)$ we omit projections for convenience.

Case (T-BRAC): By inversion, we have (A4) is an applicaton of (E-BRAC), and from the premise, we get $\mathsf{M}_{Flow} \vdash (\Sigma, e_i) \longrightarrow_i (\Sigma', e_i')$. Our goal is to show, for $\Gamma' \models \Sigma'$, that, without loss of generality, $\Gamma' ; A(\Gamma') \vdash_{\text{term},\cdot} (\![e_j']\!]) : t; \cdot$.

We apply (T-BRAC), and for the e_j, since from Lemma 58 the $\lfloor \Sigma \rfloor_j = \lfloor \Sigma' \rfloor_j$, we have $\lfloor \Gamma \rfloor_j = \lfloor \Gamma' \rfloor_j$ and can simply reuse the corresponding premises of (A2).

For e_i, we begin by observing that $\lfloor \Gamma \rfloor_i \models \lfloor \Sigma \rfloor_i$, from (A3). So, we apply Lemma 53, to obtain (A2.i') $\lfloor \Gamma' \rfloor_i ; \lfloor A(\Gamma') \rfloor_i \vdash_{\varphi,i} e_i' : t; \cdot$, where $\Gamma' \models \Sigma'$. This is sufficient for the first typing derivation for e_i.

To conclude, we still need to show $(\lfloor \Gamma' \rfloor_i)_{High} ; \lfloor A(\Gamma') \rfloor_i \vdash_{\text{term},i} e' : t; \cdot$. If $\lfloor \Gamma \rfloor_i = (\lfloor \Gamma \rfloor_i)_{High}$, then, clearly, we are done—both derivations are identical. However, if they are not the same we first use the typing derivation of e_i from the premises of (A2) with Lemma 51 to conclude that $\hat{\ell} \notin FV(e_i)$. We then use guarded strengthening, Lemma 50 to conclude $\lfloor \Gamma \rfloor_i \setminus \hat{\ell} ; A \setminus \hat{\ell} \vdash e_i' : t; \cdot$ from (A2.i'). Finally, we use weakening to conclude $(\lfloor \Gamma \rfloor_i)_{High} ; A \vdash e_i' : t; \cdot$, as needed.

Note that we need two typing derivations for e_i and e_j for the following reason. In order to apply the induction hypothesis to the premise of (E-BRAC), (both in this lemma and in adequacy), we need to show that the context used to type e_i models the store in which e_i is reduced. From the conclusion of (T-BRAC), we have that $\Gamma \models \Sigma$; so, we need a derivation of e_i in the context $\lfloor \Gamma \rfloor_i$ to satisfy the requirements of the induction hypothesis. However, the purpose of FLAIR^2 is to prove noninterference. This means that we have to show that the memory effects of all bracketed terms (since they can differ

in each sub-execution) must be limited to the fragment of memory with types t, where $High \lhd t$. The second typing derivation types e_i in a context $(\lfloor \Gamma \rfloor_i)_{High}$, meaning that the program counter/capability tokens in e_i must only be valid for mutating $High$-memory. (Similarly for e_j.) Note that it is possible for $\lfloor \Gamma \rfloor_i \neq (\lfloor \Gamma \rfloor_i)_{High}$—however, we will show that if this is the case, then $\hat{\ell} \notin FV(e_i)$.

Case (T-POP): By inversion, we have that (A4) is an application of (E-POP). We have to show that $\Gamma'; A \vdash_{\varphi, \cdot} e : t; \cdot$, where $\Gamma' \models \Sigma'$. That is, $\Gamma'(\ell_{PC}) = PC \, [\![\mathsf{pc}]\!]$. But, we have exactly this from the premise of (A2).

Case (T-APP): Here, the interesting cases are each of the reduction rules for the base terms. In all other cases, we can either appeal to the induction hypothesis or the soundness result of Lemma 53.

Sub-case *pc2cap*: When $\iota \in \{1, 2\}$, the result follows from Lemma 53. When $\iota = \cdot$ follows a similar reasoning.

Sub-case *cap2pc*: Same as previous case.

Sub-case *update*: When $\iota = \cdot$, in the first rule, store typing on the RHS follows from type-correctness of the LHS. When $\iota \in \{1, 2\}$, from the second premise of (T-BRAC), we know that the LHS is typeable in a context with Γ_{High}. Thus, we have that $m = High = l$. So, in the second rule, on the LHS, we have that the reference term has a reference type t where $High \lhd t$. Thus, store typing on the RHS follows, since it is permissible to have bracketed values in $High$ location in the store. In the third rule, when $\iota = \cdot$, from well-typedness of the LHS, we have that the reference argument $\langle v_{\ell 1} \parallel v_{\ell 2} \rangle$ has a type t, where, from (T-BRAC), $High \lhd t$. In each case, the term on the RHS, $(cap2pc \; l \; m, ())$ is obviously correct wrt the signature of *update*.

Sub-case *deref*: The first case is trivial. In the second case, on the LHS, the reference value has a type t, where $High \lhd t$. That means that the label of the location $l = High$. On the RHS, we produce a bracketed term, where each side of the bracket is labeled with $l = [T]$, which suffices to satisfy the last premise of (T-BRAC). The first premise follows from store typing of the RHS. The second premise follows because affine values are not allowed to escape into the heap; so $\lfloor v_i \rfloor_i \neq \hat{\ell}$.

Sub-case *branch*: The first rule isomorphic to the corresponding rule in FLAIR$^\mathsf{i}$—the reasoning is similar, except we use the type of *splitpc* in the conclusion, rather than reasoning about pop; e direction.

The second rule, each e_i (following reasoning similar to Lemma 53), has the type *Boxed* $(lub \; l \; m)$ $(Labeled \; m \; t_\beta)$, where, we know that $m = High$, since it is the label of the bracketed term on the LHS. Furthermore, since on the RHS, we have $\ell_{PC}, \mathsf{pc} \sqcup High$ in the store, satisfying the constraint of the second premise of (T-BRAC) is trivial.

Sub-case *apply*: Identical to branch, except using t_β instead of *Unit*.

Sub-case *splitpc*: In the first rule, type correctness of the LHS requries $\Sigma = \Sigma, \ell_{PC}{:}(l \sqcup m)$ or $\langle \ell_{PC}{:}l \sqcup m \parallel \ell_{PC}{:}l \sqcup m \rangle$—in either case, the constraints of (T-POP) are satisfied to type the RHS.

In the second rule, since both sides of the bracket are required to have the same type, once again type correctness of the LHS requires $\Sigma = \Sigma, \ell_{PC}{:}(l \sqcup m)$ or $\langle \ell_{PC}{:}l \sqcup m \parallel \ell_{PC}{:}l \sqcup$

$m\rangle$. To type the RHS, we apply (T-POP) as before, but then we must type $(\ell_{PC}, \langle v_1 \parallel v_2\rangle$ in a context Γ', with $\Gamma'(\ell_{PC}) = PC\ l$, where, it is possible that $l \neq High$. However, in typing the pair, we split the affine assumptions between the ℓ_{PC} and $\langle v_1 \parallel v_2\rangle$ sub-terms, giving the former the type $PC\ l$ as required using (T-PCAP).

This means that from the LHS, we have $\lfloor \Gamma \rfloor_i; \cdot \vdash_{\mathsf{term},i} v_i : t; \cdot$. From guarded strengthening, Proposition 50, we have $\lfloor \Gamma \rfloor_i \setminus \hat{\ell}; \cdot \vdash_{\mathsf{term},i} v_i : t; \cdot$. This latter derivation is the same as $\lfloor \Gamma' \rfloor_i \setminus \hat{\ell}; \cdot \vdash_{\mathsf{term},i} v_i : t; \cdot$, for each i. Thus, we can use this to construct $\Gamma' \setminus \hat{\ell}; \cdot \vdash_{\varphi,\iota} \langle v_1 \parallel v_2\rangle : t; \cdot$. Finally, we conclude with weakening to establish $\Gamma'; \cdot \vdash_{\varphi,\iota} \langle v_1 \parallel v_2\rangle : t; \cdot$, as needed.

Case (T-CASE), (T-CAP): Induction hypothesis.

Case (T-B1), (T-B2), (T-B3), (T-CONV): As with FLAIR[i]. $\qquad\square$

Theorem 60 (Noninterference for FLAIR, with S_{Flow})**.** *If, for well-formed Γ, the signature S_{Flow}, model M type-consistent with S_{Flow}, such that $M(\mathsf{lub}) = E_{lub}$, we have $\Gamma; \mathsf{initpc} \vdash_{\mathsf{term}} e : t; \cdot$, where $Low \lhd t$. Then, for any two stores Σ and Σ', such that $\Gamma \models \Sigma$ and $\Gamma \models \Sigma'$, and $\forall \ell \in dom(\Sigma). \mathsf{Low} \lhd \Gamma(\ell) \Rightarrow \Sigma(\ell) = \Sigma'(\ell)$, if we have*

$$M \vdash (\Sigma, e) \longrightarrow (\Sigma_1, e_1) \longrightarrow \dots \longrightarrow (\Sigma_n, e_n)$$
$$M \vdash (\Sigma', e) \longrightarrow (\Sigma'_1, e'_1) \longrightarrow \dots \longrightarrow (\Sigma'_m, e'_m)$$

Then, the sequences $\Sigma, \Sigma_1, \dots \Sigma_n$ and $\Sigma', \Sigma'_1, \dots, \Sigma'_m$ are equivalent up to stuttering.

Proof. As a corollary of Lemma 59, following a standard argument [104]. In particular, for Σ, such that $\lfloor \Sigma \rfloor_i = \Sigma_i$, $M_{\mathrm{Flow}} \vdash (\Sigma, e) \longrightarrow^* (\Sigma', e')$, from Lemma 59 we have that Σ' is well-formed, i.e., the *Low*-fragment of Σ' is bracket free. $\qquad\square$

Bibliography

[1] ABADI, M., AND FOURNET, C. Access control based on execution history, 2003.

[2] ALTENKIRCH, T., MCBRIDE, C., AND MCKINNA, J. Why dependent types matter. In preparation; available from `http://e-pig.org`.

[3] ANDREAE, C., NOBLE, J., MARKSTRUM, S., AND MILLSTEIN, T. A framework for implementing pluggable type systems. In *OOPSLA '06* (2006), ACM Press.

[4] ASPINALL, D., AND HOFFMANN, M. *Advanced Topics in Types and Programming Languages*. MIT Press, 2004, ch. Dependent Types.

[5] Authentication in ASP.NET: .NET Security Guidance. `http://msdn2.microsoft.com/en-us/library/ms978378.aspx`.

[6] AUGUSTSSON, L. Cayenne–a language with dependent types. In *ICFP '98* (New York, NY, USA, 1998), ACM Press.

[7] AYDEMIR, B., CHARGUÉRAUD, A., PIERCE, B. C., POLLACK, R., AND WEIRICH, S. Engineering formal metatheory. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2008), ACM.

[8] BANDHAKAVI, S., WINSBOROUGH, W., AND WINSLETT, M. A trust management approach for flexible policy management in security-typed languages. *IEEE Symposium on Computer Security Foundations 0* (2008), 33–47.

[9] BANERJEE, A., NAUMANN, D. A., AND ROSENBERG, S. Expressive declassification policies and modular static enforcement. *IEEE Symposium on Security and Privacy* (2008).

[10] BARENDREGT, H. Lambda calculi with types. In *Handbook of Logic in Computer Science, Volumes 1 (Background: Mathematical Structures) and 2 (Background: Computational Structures), Abramsky & Gabbay & Maibaum (Eds.), Clarendon*, vol. 2. 1992.

[11] BARENDREGT, H., AND GEUVERS, H. Proof-assistants using dependent type systems. In *Handbook of Automated Reasoning*. 2001, pp. 1149–1238.

[12] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Robust defenses for cross-site request forgery. In *To appear at the 15th ACM Conference on Computer and Communications Security (CCS 2008)* (2008).

[13] BAUER, L., LIGATTI, J., AND WALKER, D. More enforceable security policies. In *Foundations of Computer Security* (2002).

[14] BECKER, M. Y. Cassandra: flexible trust management and its application to electronic health records. Tech. Rep. UCAM-CL-TR-648, University of Cambridge, Computer Laboratory, 2005.

[15] BECKER, M. Y., AND NANZ, S. A logic for state-modifying authorization policies. In *ESORICS '07* (2007), Springer-Verlag.

[16] BENGSTON, J., BHARGAVAN, K., FOURNET, C., GORDON, A. D., AND MAFFEIS, S. Refinement types for secure implementations. In *IEEE Symposium on Computer Security Foundations* (2008).

[17] BERTOT, Y., AND CASTÉRAN, P. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag,

2004.

[18] BISHOP, M. *Computer Security: Art and Science*. Addison Wesley, 2003.

[19] BLAZE, M., FEIGENBAUM, J., AND LACY, J. Decentralized trust management. *IEEE Symposium on Security and Privacy 00* (1996), 0164.

[20] BOLAND, R. Network centricity requires more than circuits and wires. *SIGNAL* (Sept. 2006).

[21] BRUIJN, N. D. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem. *The Journal of Symbolic Logic 40*, 3 (1975).

[22] BUNEMAN, P., CHAPMAN, A., AND CHENEY, J. Provenance management in curated databases. In *SIGMOD* (2006).

[23] BUNEMAN, P., KHANNA, S., AND TAN, W. C. Why and where: A characterization of data provenance. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory* (London, UK, 2001), Springer-Verlag, pp. 316–330.

[24] CECCHET, E., MARGUERITE, J., AND ZWAENEPOEL, W. Performance and scalability of ejb applications. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), ACM, pp. 246–261.

[25] CHENEY, J. Program slicing and data provenance. *IEEE Data Eng. Bull. 30*, 4 (2007), 22–28.

[26] CHENEY, J., AHMED, A., AND ACAR, U. Provenance as dependency analysis. *Database Programming Languages* (2007).

[27] CHENG, P.-C., ROHATGI, P., KESER, C., KARGER, P. A., WAGNER, G. M., AND RENINGER, A. S. Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. *IEEE Symposium on Security and Privacy* (2007).

[28] CHIN, B., MARKSTRUM, S., AND MILLSTEIN, T. Semantic type qualifiers. In *PLDI '05* (2005), ACM Press.

[29] CHONG, S., LIU, J., MYERS, A. C., QI, X., VIKRAM, K., ZHENG, L., AND ZHENG, X. Secure web application via automatic partitioning. In *SOSP '07* (2007), ACM Press.

[30] CHONG, S., AND MYERS, A. C. Security policies for downgrading. In *CCS* (2004).

[31] CHONG, S., MYERS, A. C., NYSTROM, N., ZHENG, L., AND ZDANCEWIC, S. Jif: Java + information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2006.

[32] CHONG, S., VIKRAM, K., AND MYERS, A. C. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX Security '07* (2007).

[33] CLARKSON, M. R., AND SCHNEIDER, F. B. Hyperproperties. *csf 0* (2008), 51–65.

[34] COMPUTER WORLD, June 2008. Top Secret: CIA explains its Wikipedia-like national security project.

[35] COOPER, E., LINDLEY, S., WADLER, P., AND YALLOP, J. Links: Web programming without tiers. In *FMCO '06* (2006).

[36] COQUAND, T., AND HUET, G. The calculus of constructions. *Inf. Comput. 76*,

2-3 (1988), 95–120.

[37] CRARY, K., WALKER, D., AND MORRISETT, G. Typed memory management in a calculus of capabilities. In *Proceedings of POPL '99* (1999).

[38] DAMIANOU, N., DULAY, N., LUPU, E., AND SLOMAN, M. The ponder policy specification language. In *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks* (London, UK, 2001), Springer-Verlag, pp. 18–38.

[39] DANTAS, D. S., AND WALKER, D. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 2006), ACM, pp. 383–396.

[40] DE MOURA, L., AND BJÓRNER, N. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

[41] DELINE, R., AND FÄHNDRICH, M. Enforcing high-level protocols in low-level software. *SIGPLAN Not. 36*, 5 (2001).

[42] DENNING, D. E. A lattice model of secure information flow. *Communications of the ACM 19*, 5 (May 1976), 236–243.

[43] DUBOCHET, G. The SLinks Language. Tech. rep., University of Edinburgh, School of Informatics, 2005.

[44] The Erlang Programming Language. `http://www.erlang.org/`.

[45] ERLINGSSON, U. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, 2004. Cornell University.

[46] ERLINGSSON, U., AND SCHNEIDER, F. B. SASI enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop* (2000), ACM Press.

[47] FINDLER, R. B., AND FELLEISEN, M. Contracts for higher-order functions. *SIGPLAN Not. 37*, 9 (2002), 48–59.

[48] FOCKE, M. W., KNOKE, J. E., BARBIERI, P. A., WHERLEY, R. D., ATA, J. G., AND ENGEN, D. B. Trusted computing system. United States Patent No. 7,103,914, 2006. issued to BAE Systems Information Technology LLC.

[49] FOGARTY, S., PASALIC, E., SIEK, J., AND TAHA, W. Concoqtion: indexed types now! In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (New York, NY, USA, 2007), ACM, pp. 112–121.

[50] FOSTER, J. S., TERAUCHI, T., AND AIKEN, A. Flow sensitive type qualifiers. In *PLDI '02* (2002), ACM Press.

[51] FOURNET, C., AND GORDON, A. D. Stack inspection: theory and variants. In *POPL '02* (2002), ACM Press.

[52] FOURNET, C., AND REZK, T. Cryptographically sound implementations for typed information-flow security. *SIGPLAN Not. 43*, 1 (2008), 323–335.

[53] FRASER, T., NICK L. PETRONI, J., AND ARBAUGH, W. A. Applying flow-sensitive CQUAL to verify MINIX authorization check placement. In *PLAS* (2006), ACM Press.

[54] GARRETT, J. J. Ajax: A new approach to web applications. `http://www.adaptivepath.com/publications/essays/archives/000385.php`, feb 2005.

[55] GIRARD, J.-Y. *Interprétation fonctionelle et élimination des coupures de*

l'arithmétique d'ordre supérieur. PhD thesis, Université Paris VI I, 1972.

[56] GONG, L. *Inside Java 2 platform security architecture, API design, and implementation*. Addison-Wesley, 1999.

[57] Google Web Toolkit. `http://code.google.com/webtoolkit/`.

[58] GROSSMAN, D., MORRISETT, G., AND ZDANCEWIC, S. Syntactic type abstraction. *ACM TOPLAS 22*, 6 (2000), 1037–1080.

[59] HAMLEN, K. W., MORRISETT, G., AND SCHNEIDER, F. B. Computability classes for enforcement mechanisms. *ACM TOPLAS 28*, 1 (2006), 175–205.

[60] HARPER, R., AND MORRISETT, G. Compiling polymorphism using intensional type analysis. In *POPL '95* (1995), ACM Press.

[61] HICKS, B., KING, D., MCDANIEL, P., AND HICKS, M. Trusted declassification: high-level policy for a security-typed language. In *PLAS '06* (2006), ACM Press.

[62] HICKS, B., MISIAK, T., AND MCDANIEL, P. Channels: Runtime system infrastructure for security-typed languages. *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (2007).

[63] HUGHES, J. Generalising monads to arrows. *Science of Computer Programming 37*, 1–3 (2000), 67–111.

[64] JIA, L., VAUGHAN, J., MAZURAK, K., ZHAO, J., ZARKO, L., SCHORR, J., AND ZDANCEWIC, S. Aura: A programming language for authorization and audit. In *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008)* (2008).

[65] JIM, T. Sd3: A trust management system with certified evaluation. In *SP '01: Proceedings of the IEEE Symposium on Security and Privacy* (2001), IEEE Computer Society, p. 106.

[66] JIM, T. System call monitoring using authenticated system calls. *IEEE Trans. Dependable Secur. Comput. 3*, 3 (2006), 216–229. Member-Mohan Rajagopalan and Member-Matti A. Hiltunen and Fellow-Richard D. Schlichting.

[67] JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference* (2002).

[68] JIM, T., SWAMY, N., AND HICKS, M. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the International World Wide Web Conference (WWW)* (May 2007), pp. 601–610.

[69] JONES, S. L. P., AND WADLER, P. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1993), ACM, pp. 71–84.

[70] JONES, S. P. Wearing the hair shirt: A retrospective on haskell. Invited Talk at POPL 2003, 2003.

[71] KELSEY, R., CLINGER, W., AND (EDITORS), J. R. Revised[5] report on the algorithmic language Scheme. *ACM SIGPLAN Notices 33*, 9 (1998), 26–76.

[72] KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *ECOOP '07*. Springer-Verlag, 1997.

[73] KISELYOV, O., AND CHIEH SHAN, C. Functional pearl: implicit configurations– or, type classes reflect the values of types. In *Haskell '04: Proceedings of the*

2004 ACM SIGPLAN workshop on Haskell (New York, NY, USA, 2004), ACM, pp. 33–44.

[74] KRISHNAMURTHI, S. The continue server (or, how i administered padl 2002 and 2003). In *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages* (London, UK, 2003), Springer-Verlag, pp. 2–16.

[75] LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. 3*, 2 (1977), 125–143.

[76] LI, N., MITCHELL, J. C., AND WINSBOROUGH, W. H. Design of a Role-Based Trust-Management Framework. In *S&P '02* (2002), IEEE Computer Society Press.

[77] LI, N., MITCHELL, J. C., AND WINSBOROUGH, W. H. Beyond proof-of-compliance: security analysis in trust management. *J. ACM 52*, 3 (2005), 474–514.

[78] LI, P., AND ZDANCEWIC, S. Encoding information flow in Haskell. In *CSFW '06* (2006), IEEE Computer Society Press.

[79] LIGATTI, J., BAUER, L., AND WALKER, D. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security* (2003).

[80] LOSCOCCO, P., AND SMALLEY, S. Integrating flexible support for security policies into the Linux operating system. In *FREENIX 2001* (2001), USENIX Association.

[81] MA, Q., AND REYNOLDS, J. C. Types, abstractions, and parametric polymorphism, part 2. In *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics* (London, UK, 1992), Springer-Verlag, pp. 1–40.

[82] MARINO, D., CHIN, B., MILLSTEIN, T., TAN, G., SIMMONS, R. J., AND WALKER, D. Mechanized metatheory for user-defined type extensions. In *WMM '06* (2006).

[83] MEIJER, E., BECKMAN, B., AND BIERMAN, G. Linq: reconciling object, relations and xml in the .net framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2006), ACM, pp. 706–706.

[84] MICROSOFT LIVE LABS. Volta Technology Preview. Available at `http://labs.live.com/volta`.

[85] MITCHELL, J. C. *Foundations of Programming Languages*. MIT Press, 1996.

[86] MITRE. Common vulnerabilities and exposures. `http://cve.mitre.org`.

[87] MOGGI, E. Computational lambda-calculus and monads. *Logic in Computer Science, 1989. LICS '89, Proceedings., Fourth Annual Symposium on* (Jun 1989), 14–23.

[88] MYERS, A. C., AND LISKOV, B. A Decentralized Model for Information Flow Control. In *Proc. Symp. On Operating System Principles (SOSP)* (1997).

[89] MYERS, A. C., AND LISKOV, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (2000), 410–442.

[90] Security privileges provided by MySQL. `http://dev.mysql.com/doc/refman/5.1/en/privileges-provided.html`.

[91] NANEVSKI, A., MORRISETT, G., AND BIRKEDAL, L. Polymorphism and sepa-

ration in hoare type theory. In *ICFP '06: Proceedings of the eleventh ACM SIG-PLAN international conference on Functional programming* (New York, NY, USA, 2006), ACM, pp. 62–73.

[92] NATIONAL COMMISSION ON TERRORIST ATTACKS UPON THE UNITED STATES. *The 9/11 Commission Report: Final Report of the National Commission on Terrorist Attacks Upon the United States (Indexed Hardcover, Authorized Edition)*. W. W. Norton & Company, August 2004.

[93] NATIONAL HEALTH SERVICE. Spine. `http://www.connectingforhealth.nhs.uk/systemsandservices/spine`.

[94] NIPKOW, T., PAULSON, L. C., AND WENZEL, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.

[95] National Institute of Standards and Technology : Role-based access control. `http://csrc.nist.gov/rbac/`.

[96] NYSTROM, N., SARASWAT, V., PALSBERG, J., AND GROTHOFF, C. Constrained types for object-oriented languages. In *OOPSLA '08* (2008).

[97] ORACLE CORPORATION. Oracle 10g release documentation, 2007. Available at `http://www.oracle.com/technology/documentation/database10g.html`.

[98] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project, 1998.

[99] PETRONI, JR., N. L., AND HICKS, M. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (Oct. 2007), pp. 103–115.

[100] PHP Security Consortium: PHP Security Guide. `http://phpsec.org/projects/guide/`.

[101] PIERCE, B. *Types and Programming Languages*. MIT Press, 2002.

[102] POSTGRESQL GLOBAL DEVELOPMENT GROUP. Postgresql 8.2.1 software release, 2007. Available at `http://www.postgresql.org`.

[103] Security privileges provided by PostgreSQL. `http://www.postgresql.org/docs/8.2/static/ddl-priv.html`.

[104] POTTIER, F., AND SIMONET, V. Information flow inference for ML. *ACM TOPLAS 25*, 1 (Jan. 2003).

[105] PRATIKAKIS, P., FOSTER, J. S., AND HICKS, M. Context-sensitive correlation analysis for detecting races. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2006).

[106] PUGH, B., AND SPACCO, J. Rubis revisited: why j2ee benchmarking is hard. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2004), ACM, pp. 204–205.

[107] Role based access control. `http://csrc.nist.gov/rbac/`, 2006.

[108] REUTERS, October 2006. U.S. Intelligence Unveils Spy Version of Wikipedia.

[109] RONDON, P. M., KAWAGUCI, M., AND JHALA, R. Liquid types. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2008), ACM, pp. 159–169.

[110] Securing Your Rails Application. `http://manuals.rubyonrails.com/read/`

```
book/8.
```

[111] SABELFELD, A., AND MYERS, A. C. Language-based information-flow security. *JSAC 21*, 1 (Jan. 2003), 5–19.

[112] SABELFELD, A., AND SANDS, D. Dimensions and principles of declassification. In *CSFW '05* (2005), IEEE Computer Society.

[113] SCHNEIDER, F. B. Enforceable security policies. *ACM Trans. Inf. Syst. Secur. 3*, 1 (2000), 30–50.

[114] SecurityFocus: Access control bypass vulnerabilities. `http://search.securityfocus.com/swsearch?metaname=alldoc&query=access+control+bypass`.

[115] SERRANO, M., GALLESIO, E., AND LOITSCH, F. Hop: a language for programming the web 2.0. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (New York, NY, USA, 2006), ACM, pp. 975–985.

[116] SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. Detecting Format String Vulnerabilities with Type Qualifiers. In *USENIX Security '01* (2001), USENIX Association.

[117] SIMMHAN, Y. L., PLALE, B., AND GANNON, D. A survey of data provenance in e-science. *SIGMOD Rec. 34*, 3 (2005), 31–36.

[118] SIMONET, V. FlowCaml in a nutshell. In *APPSEM-II* (Mar. 2003), G. Hutton, Ed., pp. 152–165.

[119] Authorization and permissions in SQLServer. `http://msdn2.microsoft.com/en-us/library/bb669084.aspx`.

[120] STANDISH, T. A. Extensibility in programming language design. *SIGPLAN Not. 10*, 7 (1975), 18–21.

[121] STONE, C. A. *Singleton kinds and singleton types*. PhD thesis, Pittsburgh, PA, USA, 2000. Chair-Robert Harper.

[122] STROM, R. E., AND YEMINI, S. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng. 12*, 1 (1986), 157–171.

[123] SWAMY, N., AND HICKS, M. Fable: A language for enforcing user-defined security policies (extended version). University of Maryland, Technical Report, CS-TR-4876; `http://www.cs.umd.edu/projects/PL/fable/TR.pdf`, 2007.

[124] SWAMY, N., HICKS, M., MORRISETT, G., GROSSMAN, D., AND JIM, T. Safe manual memory management in cyclone. *Sci. Comput. Programming 62*, 2 (Oct. 2006), 122–144. Special issue on memory management. Expands ISMM conference paper of the same name.

[125] SWAMY, N., HICKS, M., TSE, S., AND ZDANCEWIC, S. Managing policy updates in security-typed languages. In *Proc. Computer Security Foundations Workshop (CSFW)* (July 2006), pp. 202–216.

[126] TERAUCHI, T., AND AIKEN, A. Secure information flow as a safety problem. In *Proceedings of the Twelfth International Static Analysis Symposium* (Sept. 2005).

[127] TSE, S., AND ZDANCEWIC, S. Run-time Principals in Information-flow Type Systems. In *S&P '04* (2004), IEEE Computer Society Press.

[128] UNITED STATES DEPARTMENT OF DEFENSE. Department of defense directive number 5230.11 : Disclosure of classified military information to foreign govern-

ments and international organizations, 1992.

[129] UNITED STATES GOVERNMENT ACCOUNTABILITY OFFICE. Major federal networks that support homeland security functions. Tech. Rep. GAO-04-375, United States Government Accountability Office, 2004.

[130] VAUGHAN, J. A., JIA, L., MAZURAK, K., AND ZDANCEWIC, S. Evidence-based audit. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium* (Pittsburgh, PA, USA, June 2008).

[131] VAUGHAN, J. A., AND ZDANCEWIC, S. A cryptographic decentralized label model. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 192–206.

[132] VOLPANO, D., SMITH, G., AND IRVINE, C. A sound type system for secure flow analysis. *Journal of Computer Security 4*, 3 (1996), 167–187.

[133] VON DINCKLAGE, D., AND DIWAN, A. Explaining failures of program analyses. *SIGPLAN Not. 43*, 6 (2008), 260–269.

[134] VORONKOV, A. Easychair. Software distribution available at `http://www.easychair.org/`.

[135] WADLER, P. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987, ch. List Comprehensions.

[136] WADLER, P. Theorems for free! In *Proceedings 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989*. ACM Press, New York, 1989, pp. 347–359.

[137] WADLER, P., AND BLOTT, S. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1989), ACM, pp. 60–76.

[138] WADLER, P., AND FINDLER, R. B. Well-typed programs can't be blamed. In *International Conference of Functional Programming* (September 2007).

[139] WALKER, D. A type system for expressive security policies. In *POPL '00* (2000), ACM Press.

[140] WALKER, D. *Advanced Topics in Types and Programming Languages*. MIT Press, 2004, ch. Substructural Type Systems.

[141] WEISER, M. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (1981), IEEE Computer Society Press, pp. 439–449.

[142] WONG, L. Kleisli, a functional query system. *Journal of Functional Programming 10*, 1 (2000).

[143] XACML AND OASIS SECURITY SERVICES TECHNICAL COMMITTEE. eXtensible Access Control Markup Language (XACML) Committee Specificaion 2.0.

[144] XI, H. Applied Type System (extended abstract). In *TYPES 2003* (2004), Springer-Verlag LNCS 3085.

[145] XI, H., AND PFENNING, F. Dependent types in practical programming. In *POPL '99* (1999), ACM Press.

[146] Xquery 1.0: An xml query language. XML Query and XSL Working Groups, W3C Working Draft, 2005.

[147] ZDANCEWIC, S., AND MYERS, A. C. Robust declassification. In *Proc. of 14th IEEE Computer Security Foundations Workshop* (Cape Breton, Canada, June

2001), pp. 15–23.

[148] ZHANG, X., EDWARDS, A., AND JAEGER, T. Using CQUAL for static analysis of authorization hook placement. In *USENIX Security* (2002).

[149] ZHENG, L., AND MYERS, A. C. Dynamic security labels and noninterference. In *FAST '04* (2004), Springer.