Analyzing Security Advice in Functional Aspect-oriented Programming Languages

Daniel S. Dantas

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
By the Department of
Computer Science

September 2007

Abstract

This thesis extends functional programming languages with aspect-oriented features, primarily to explore aspect-oriented enforcement of security policies.

First, this thesis examines an aspect-oriented implementation of the Java security mechanism, which requires the security advice to be triggered by functions with diverse types. I present a new language, AspectML, that allows type-safe polymorphic advice using pointcuts constructed from a collection of polymorphic join points. I then compare my AspectML implementation of the Java security mechanism against the existing Java implementation.

Second, in ordinary aspect-oriented programming, security and other advice added after-the-fact to an existing codebase can disrupt important data invariants and prevent local reasoning. Instead, this thesis shows that many common aspects, including security advice, can be implemented as harmless advice. Harmless advice uses a novel type and effect system related to information-flow type systems to ensure that harmless advice cannot modify the behavior of mainline code. To demonstrate the usefulness of harmless advice for security, I implement many of the security examples used by the Naccio execution monitoring system as harmless advice.

Finally, this thesis expands the harmless advice specification to allow programmers to create interference policies to define how system libraries can be used by aspects. These policies use a combination of compile-time type checking and runtime monitoring to enforce the desired degree of harmlessness on the aspect-oriented program. My thesis formalizes an idealized file I/O library and proves that an interference policy specified by our policy language can continue to enforce our original view of harmlessness for advice that uses file I/O.

Acknowledgments

I would like to thank my advisor, Prof. David Walker, for introducing me to the world of programming languages and computer security. He was always excited to discuss ideas, challenges, and opportunities. He is an excellent researcher, and I feel honored to have worked with him.

I would like to thank my family, Daniel, Angela, Rebecca, and Mary Dantas, and Luis and Rosa Martinez for their constant love and support.

I would like to thank my colleagues at University of Pennsylvania, Geoff Washburn and Prof. Stephanie Weirich. It was a delight to collaborate to create a new programming language with them, and their vast knowledge of type inference systems proved invaluable during the AspectML project. Our trip to the conference in Estonia will not be forgotten.

I would like to thank the members of my thesis committee. Prof. Stephanie Weirich and Prof. Andrew Appel read and provided valuable feedback on my thesis, and Prof. David August and Prof. Edward Felten viewed my pre-FPO presentation and provided valuable guidance as I prepared to finish my doctoral degree.

To my colleagues at Princeton, you have provided me with endless guidance, knowledge, encouragement, and amusement over my five years. Special thanks to my colleagues in programming languages: Frances Perry, Limin Jia, Dr. Yitzhak Mandelbaum, and Prof. Jay Ligatti. Dr. Chris Sadler shared an apartment and lent me his family's furniture for four years. Finally, the participants and leadership of the Princeton Graduate Christian Fellowship provided invaluable spiritual and emotional friendship during my stay at Princeton.

Finally, I'd like to thank Princeton University for providing a Gordon Wu Fellowship, and to thank the United States Department of Defense and Air Force Office

of Strategic Research for providing a National Defense Science and Engineering Graduate Fellowship. I was also funded in part by National Science Foundation grants CCF-0238328 and CNS-0615213. These funding sources were invaluable in supporting my research. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of Princeton University, the US-DOD, the US-AFOSR, or the NSF, and no official enforsement should be inferred.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction to Aspect-oriented Programming

In the design and maintenance of large, complex programs, the program code is typically divided into distinct units of compilation. These units contain code that represent a discrete "idea" that can be separately understood and modified. These units of compilation are represented in the module system of Standard ML [5] and the class mechanism of Java [25]. In both languages, the unit of compilation allows the programmer to specify an external interface that must be used to access the unit and to compile the unit separately from other code in the program. An example of these units of compilation can be found in the Java network I/O library. The library is divided into, among other things, a class that contains code for manipulating IP addresses, a class that contains code for manipulating URLs, a class that contains code to manipulate UDP packets, a class that contains code for operations on UDP sockets, a class that contains code for operations on active TCP sockets, and a class that contains code for operations on passive, server TCP sockets. Having separate

units of compilation enhances the comprehensibility and maintainability of the Java network I/O library.

However, while maintaining and modifying real-world programs, programmers have discovered that it is often useful to add a new feature to an existing program "after the fact" in a manner that cuts across the boundaries of the existing units of compilation. For example, the programmer may wish to add a network security feature that checks whether a user has the proper permissions when executing certain network operations. In a language like Java, code for the security feature will be scattered across the existing network units of compilation. The resulting security feature can be difficult to understand and maintain. In other situations, a programmer may not be able to modify the design of the existing units of compilation, perhaps because they do not have access to the original source code. Aspect-oriented programming is a programming language paradigm designed to allow programmers to easily add such new features after the fact that cut across existing units of compilation.

Aspect-oriented programming languages allow programmers to specify *what* computations to perform as well as *when* to perform them. For example, aspects make it easy to implement an access control mechanism that checks for a "network connect" permission upon attempting to connect through the network to another computer. The *what* in this example is the check for the proper network I/O permission, and the *when* is the instant of time just prior to connecting over the network. In aspect-oriented terminology, the specification of what to do is called *advice* and the specification of when to do it is called a *pointcut*. An event, such as a function call, that may trigger a pointcut is called a *join point*. A collection of pointcut and advice organized to perform a coherent task is called an *aspect*.

The security code described above could be implemented without aspects by placing the access control checks directly into the body of all affected network methods. Indeed, the Java security mechanism works in this very way. However, at least four problems arise when the programmer does the insertion manually.

- First, it is no longer easy to adjust when the advice should execute, as the programmer must explicitly extract and relocate calls to security functions. Therefore, for applications where the "when" is in rapid flux, aspect-oriented languages can be superior to conventional languages.

- Second, there may be a specific convention concerning how to call the security functions. When calls to these functions are spread throughout the code base, it may be difficult to maintain these conventions correctly. For example, IBM [8] experimented with aspects in their middleware product line, finding that aspects aided in the consistent application of cross-cutting features and improved the overall reliability of the system. Aspect-oriented features added structure and discipline to IBM's applications.

- Third, when code is injected directly into the body of each method, the code becomes "scattered," in many cases making it difficult to understand. This problem is particularly relevant to the implementation of security policies for programs. Without centralization of security policy using aspects, security policy implementations can become spread among many units of compilation, making it very difficult for a security expert to audit them effectively.

- Fourth, in some situations, the source code is unavailable to be examined or modified. For example, the source code may belong to a third party who is not willing to provide this proprietary information to its users. Consequently,

manual insertion of function calls is not possible. In these cases, aspects can be used as a robust form of external software patching [20].

To date there has been much success integrating aspects into object-oriented languages. The most widely used aspect-oriented programming language is AspectJ [30], an aspect-oriented extension of the Java programming language. There has been less progress on studying the interactions between aspects and typed functional languages. The first challenge we will address in this thesis is constructing a type system that ensures safety, yet is sufficiently flexible to fit aspect-oriented programming idioms.

There are also disadvantages to aspect-oriented programming. Aspect-oriented programming threatens conventional modularity principles and undermines a programmer's ability to reason locally about the behavior of their code. Aspects can reach inside modules, influence the behavior of local routines, and alter local data structures. As a result, to understand the semantics of code in an aspect-oriented language such as AspectJ, programmers will have to examine all external aspects that might modify local data structures or control flow. As the number and invasiveness of aspects grows, understanding and maintaining their programs may become more and more difficult. The second challenge we will discuss in this thesis will be how to allow programmers to add new, crosscutting features, such as security enforcement, after the fact while also allowing them to enjoy most of the local reasoning principles they have come to depend upon for program understanding, development, and maintenance.

To summarize, in this thesis, we explore the interaction between functional programming and aspect-oriented programming, primarily to explore aspect-oriented enforcement of security policies. We wish to explore more fully what features

an aspect-oriented, functional programming language must have to support the creation and maintenance of security advice.

1.2 Security

We have chosen to focus on security examples and case studies in this thesis because it appears to be one of the best, most convincing applications of aspect-oriented programming technology. Indeed, many previous researchers have argued that aspect-oriented programming mechanisms enable more modular implementation of access control infrastructure than standard programming languages. More specifically, since an aspect-oriented implementation can encapsulate not only the definition of *what* an access control check is supposed to do, but also the complete list of places *where* that access control check should occur, aspect-oriented security policy specifications are easier to understand. This in turn makes aspect-oriented security policy specifications easier to audit – the security auditor need not search through thousands of lines of library or application code to find the relatively few lines of access control checks. In particular, analysis of the pointcut definitions used in a policy can often tell the auditor whether or not access control checks have been omitted. In addition, because all security code is centralized, when security vulnerabilities are identified, security policy updates can be made more easily. Moreover, to distribute the changed policy, a single new aspect can be deployed as opposed to an entire new library or set of libraries. Several systems use this kind of aspect-oriented design (occasionally without acknowledging it as such) in imperative and object-oriented languages including Polymer [4], SASI [14], PoET/PSLang [15], Naccio [16, 17], and Java MAC [32, 35].

1.3 MinAML

This thesis builds upon the aspect-oriented framework, MinAML, proposed by Walker, Zdancewic, and Ligatti [37]. That work distilled aspect-oriented programming into its fundamental components, clearly separating the semantics of aspects (join points and advice) from that of the underlying programming language. They used the simply-typed λ-calculus as the basic model of computation. The reason that they could develop such a minimal model is that their calculus reduced the aspect language to the most critical components, relying on translation from their MinAML source language to generate the full power of modern AOPLs.

MinAML contained advice that executed before, after, and in place of (around) functions, though their *around* advice did not have the conventional nesting behavior of around advice in the literature. MinAML was *oblivious* [19] – programmers could add functionality to a program "after-the-fact" in the typical aspect-oriented style. To provide support for stack-inspection-like security infrastructure, and to emulate the CFlow feature of AspectJ, MinAML also included a general mechanism for analyzing metadata associated with functions on the current call stack.

To specify the dynamic semantics of MinAML, Walker, Zdancewic, and Ligatti gave a type-directed translation from the source into a type-safe, monomorphic core calculus with its own operational semantics. Defining the operational semantics of a complex language via a translation to a simpler one has considerable precedent. Their translation could be seen as providing a denotational semantics for MinAML. Harper and Stone developed an improved semantics for Standard ML by elaboration into a simpler language [22]. More recently, Avgustinov, et al. have given the

first rigorous semantics for the AspectJ pointcut language via a translation to Safe Datalog [3].

Their use of a translation helped to modularize the semantics for MinAML by unraveling complex source-language objects into simple, orthogonal core calculus objects. Indeed, they have attempted to give a clean semantics to each feature in this language, and to separate unrelated concerns. This was instrumental in allowing us to further explore and extend their language.

Furthermore, defining the semantics of MinAML this way makes it easier to understand the language. It is possible to focus on comprehending the core language and the translation separately, rather than the composition of the two. This modular design also makes formal reasoning about the language considerably easier. This is important if mechanically verified proofs become the norm. For example, it is unlikely that Lee, Crary, and Harper would have been able to provide a mechanically verified semantics for Standard ML if they had worked directly from the *The Definition of Standard ML* [34, 40]. Walker, Zdancewic, and Ligatti did not mechanically formalize the definition of MinAML, but the design of their languages has made it possible for us to prove important and nontrivial properties of our extensions to their language.

1.4 Structure of the Thesis

In this thesis, we will explore the interaction between functional programming languages and aspect-oriented programming languages. We will ask what features programmers need to implement security advice, and what tools and analyses we can provide to help programmers better understand the implications of adding

security advice in their program. We will conclude that programmers require polymorphically typed advice to implement useful security advice, and that they can use a static analysis of the "harmlessness" of advice to assure them that adding security advice after the fact to their mainline code will not change their program's behavior. Finally, we will allow programmers to customize the level of harmlessness of their security advice to allow them the language flexibility and customizability that real-world security advice programmers need.

We detail three contributions in this thesis.

1.4.1 Polymorphic Advice

In Chapter 2, we examine an aspect-oriented implementation of the Java security mechanism, which requires the security advice to be triggered by functions with diverse argument and return types. We present a new language, AspectML, that allows a programmer to define type-safe polymorphic advice using pointcuts constructed from a collection of polymorphic join points. In particular, we focus on the synergy between polymorphism and aspect-oriented programming – the combination is clearly more expressive than the sum of its parts. At the simplest level, AspectML allows programmers to reference control-flow points that appear in polymorphic code. However, we have show that polymorphic pointcuts are necessary even when the underlying code base is completely monomorphic. Otherwise, there is no way to assemble a collection of join points that appear in code with different types. In addition, run-time type analysis allows programmers to define polymorphic advice that behaves differently depending upon the type of its argument.

From a technical standpoint, we give AspectML a semantics by compiling it into a typed intermediate calculus, $\mathbb{F}_A$. The definition of this intermediate calculus is

also an important contribution of this work. An interesting element of $\mathbb{F}_A$ is the definition of our label hierarchy, which allows us to form groups of related control flow points. Here, polymorphism is again essential: it is not possible to define these groups in a monomorphic language. The second interesting element of $\mathbb{F}_A$ is our support for reification of the current call stack. In addition to being polymorphic, our treatment of static analysis is more flexible, simpler semantically and easier for programmers to use than the core calculus of MinAML. Moreover, it is a good fit with standard data-driven functional programming idioms.

Finally, we compare our AspectML implementation of the Java security mechanism against the existing Java implementation using polymorphic advice. Through this case study, AspectML seems useful to implement security advice.

This chapter describes joint research in collaboration with Geoff Washburn and Prof. Stephanie Weirich of the University of Pennsylvania. The details of the collaboration will be described fully in Chapter 2. It is an expansion of research published at the ICFP 2005 conference [11] and to be published in the TOPLAS 2007 journal [12].

1.4.2 Harmless Advice

In Chapter 3, we examine how, in ordinary aspect-oriented programming, security and other advice added after the fact to an existing codebase can disrupt important data invariants and prevent local reasoning. Instead, we show that many common aspects, including security advice, can be implemented as harmless advice. Harmless advice has the advantage that it may be added to a program after the fact, in the typical aspect-oriented style, without corrupting important mainline data

invariants. As a result, programmers using harmless advice retain the ability to perform local reasoning about partial correctness of their programs.

Harmless advice uses a novel type and effect system related to information-flow type systems to ensure that harmless advice cannot modify the behavior of mainline code. We prove a noninterference property using a proof technique based on simultaneous execution of programs by Simonet and Pottier [45]. We then use this noninterference property, combined with the type safety of the translation from the source language to the core language, to show that aspects in our source language, HarmlessAML, are harmless.

Finally, to demonstrate the usefulness of harmless advice for security, we implement in HarmlessAML many of the security examples used by the Naccio execution monitoring system by Evans and Twyman [16, 17] as harmless advice. We find that many access control tasks can be performed by harmless advice. As such, aspects to implement security tasks can be added after-the fact without modifying the underlying behavior of the main program.

This chapter is an expansion of research published at the FOOL 2005 workshop [9] and the POPL 2006 conference [10].

1.4.3 Interference Policies

Finally, in Chapter 4, we expand HarmlessAML to allow programmers to create interference policies for system libraries to define how those libraries can be used by aspects. These policies use a selection of compile-time type checking and run-time monitoring to enforce the desired degree of harmlessness on aspect-oriented programs.

To test our interference policy mechanism on a system library, we perform a file I/O case study. We demonstrate that, in addition to many other possibilities, interference policies can certainly be used to enforce the previous chapter's definition of "harmlessness" on a file I/O library. We formalize an underlying noninterfering file I/O library in the core calculus, extending the $\mathbb{F}_{HRM}$ and $\mathbb{F}_{HRM2}$ syntax, operational semantics, and type system to include a idealized file system. We prove that the new core file system is type-safe and preserves strong noninterference properties. We then demonstrate that we can define an interference and resource policy for the source-level file I/O library that defines a type-safe translation to the baked-in noninterfering core-level file I/O library. This allowed us to prove that aspects that use the source-level file system are harmless. Therefore, we show that an interference policy on a system library specified by our policy language can continue to enforce our original view of harmlessness for advice that uses that system library.

Chapter 2

Polymorphic Aspects

2.1 Introduction

Many aspect-oriented programming tasks, including the security tasks mentioned in the previous chapter, are best handled by a single piece of advice that executes before, after, or around many different function calls. In this case, the type of the advice is not directly connected with the type of a single function, but with a whole collection of functions. To type check advice in such situations, one must first determine a type for the collection and then link the type of the collection to the type of the advice. Normally, the type of the collection (the pointcut) will be highly polymorphic, and the type of each element will be less polymorphic than the collection's type.

In addition to finding polymorphic types for pointcuts and advice, it is important for advice to be able to change its behavior depending upon the type of the advised function. For instance, access control advice might be specialized so that on calls to functions with file descriptor arguments, the advice tracks whether the caller

is allowed to access that file. This and other similar examples require that the advice can determine the type of the function argument. In AspectJ and other object-oriented languages where subtype polymorphism is predominant, downcasts are used to determine types. However, in ML, and other functional languages, parametric polymorphism is predominant, and therefore run-time type analysis is the appropriate mechanism.

In this chapter, we extend MinAML to develop AspectML, a typed functional programming language with first-class, polymorphic pointcuts and advice, and run-time type analysis. Like MinAML, AspectML contains before, after, and around advice. However, unlike MinAML, AspectML's around advice correctly emulates the advice-nesting behavior of AspectJ advice. Like MinAML, AspectML is *oblivious* [19]—programmers can add functionality to a program "after-the-fact" in the typical aspect-oriented style. Unlike MinAML, the polymorphic pointcuts in AspectML are first-class objects, an important feature for building effective aspect-oriented libraries. To provide support for stack-inspection-like security infrastructure, and to emulate AspectJ's CFlow, AspectML extends the MinAML general mechanism for analyzing call stack metadata in a polymorphic fashion.

As with MinAML, we specify the dynamic semantics of AspectML with a type-directed translation from the source into a type-safe core calculus with its own operational semantics. This core calculus, though it builds on the WZL core calculus, is itself an important contribution of our work. One of the novelties of the WZL core calculus is its first-class, polymorphic labels, which can be used to mark any control-flow point in a program. Unlike in the WZL core calculus, where labels are monomorphic, polymorphism allows us to structure the labels in a tree-shaped hierarchy. Intuitively, each internal node in the tree represents a group of control-

flow points, while the leaves represent single control-flow points. Depending upon how these labels are used, there could be groups for all points just before execution of a function or just after, groups for all labels in a module, groups for getting or setting references, or groups for raising or catching exceptions. Polymorphism is crucial for defining these groups since the type of a parent label, which represents a group, must be a polymorphic generalization of the type of each member of the group.

We present the following contributions in this chapter.

- In Section 2.2, we introduce our surface language, idealized AspectML, that includes three novel features essential for aspect-oriented programming in a strongly-typed functional language: polymorphic pointcuts, polymorphic advice and polymorphic analysis of metadata on the current call stack. In addition, we add run-time type analysis, which, though not a new feature, is seamlessly integrated into the rest of the language. A full implementation of AspectML is also described in this section.

- We define a conservative extension of the Hindley-Milner type inference algorithm for idealized AspectML. In the absence of aspect-oriented features and run-time type analysis, type inference works as usual; inference for aspects and run-time type analysis is integrated into the system smoothly through a novel form of local type inference. Additionally, we believe the general principles behind our type inference techniques can be used in other settings. The type inference mechanism of AspectML was not primarily designed by the author of this thesis, and, as such, will not be described in depth.

- In Section 2.3, we have used our full implementation of AspectML to write several example programs to demonstrate the usefulness of AspectML, including a security case study. The case study examines an AspectML implementation of the Java stack inspection security mechanism.

- In Section 2.4, we define an explicitly-typed core calculus $\mathbb{F}_A$ that carefully separates mechanisms for polymorphic first-class function definition, polymorphic advice definition, and run-time type analysis. This core calculus introduces a new primitive notion of polymorphic labeled control flow points, to specify pointcuts in an orthogonal manner. We prove that this core calculus is type-safe.

- In Section 2.5, we define the semantics of AspectML by a translation into $\mathbb{F}_A$. We prove that the translation is type-preserving, and therefore that the surface language is also type safe.

2.1.1 Description of Collaboration

The research described in this chapter was performed in collaboration with Geoffrey Washburn and Prof. Stephanie Weirich at the University of Pennsylvania. It is an elaboration of research published at the ICFP 2005 conference [11] and to be published in the TOPLAS 2007 journal [12]. The TOPLAS journal paper extended the original ICFP conference paper with a fuller language implementation, a security case study, and *around* advice. The collaborative details were as follows:

- The AspectML language design in Section 2.2 was a collaborative effort with Geoffrey Washburn and Prof. Stephanie Weirich.

- The proof-of-concept conference implementation of the language was created by the author, while the full journal implementation was designed by Geoffrey Washburn.

- Geoffrey Washburn's full journal implementation of AspectML was invaluable in the creation by the author of the journal case study of the Java security mechanism in Section 2.3.

- The type inference mechanism for AspectML was primarily designed by Geoffrey Washburn and Prof. Stephanie Weirich and, as such, is not described in detail in this thesis. However, the type inference soundness proof for *around* advice in the journal version was primarily performed by the author.

- The modification and extension of AspectML and the core calculus to allow *around* advice in the journal version was primarily designed by the author.

- In Section 2.4, the type-safety proof for $\mathbb{F}_A$ in the conference version and its extension to *around* advice in the journal version was performed by the author.

- The translation type-safety proof in the conference version was performed by Geoffrey Washburn, while the extension of the proof to *around* advice in the journal version was primarily performed by the author.

2.2 Programming in AspectML

AspectML is a polymorphic functional, aspect-oriented language based on the ML family of languages. Figure 2.1 presents the syntax of the idealized version of the

(*polytypes*) s ::= `<`$\overline{a}$`>` t
(*pointcut typ*) pt ::= `(<`$\overline{a}$`>` t_1 `~>` t_2`)`
(*monotypes*) t ::= a | `Unit` | `String` | `Stack` | t_1 `->` t_2 | `pc` pt
(*trigger time*) tm ::= `before` | `after` | `around`
(*terms*) e ::= x | `()` | c | $e_1 e_2$ | `let` ds `in` e | `stkcase` e_1 `(`$\overline{pat\texttt{=>}e}$ `|_=>` e_2`)`
| `typecase<`t`>` a `(`$\overline{t\texttt{=>}e}$ `|_=>` e`)` | `#`$\overline{x}$`:`pt`#` | `any` | e`:`t
(*stack pats*) pat ::= x | `[]` | f`::`pat
(*frame pats*) f ::= `_` | `(|`e`|)<`$\overline{a}$`>` `(`x`:`t`,`y`)`
(*declarations*) ds ::= `fun` x_1 `<`$\overline{a}$`>` `(`x_2`:`t_1`)`:`$t_2$ `=` e
| `advice` tm `(|`e_1`|)` `<`$\overline{a}$`>` `(`x`:`t_1`,`y`,`z`)`:`$t_2$ `=` e_2
| `case-advice` tm `(|`e_1`|)` `(`x`:`t_1`,`y`,`z`)`:`$t_2$ `=` e_2

Figure 2.1: Syntax of Idealized AspectML

language. However, all of the examples of this section are written in full AspectML, which extends this syntax with many common constructs following Standard ML.[1]

In Figure 2.1 and elsewhere, we use over-bars to denote lists of syntactic objects: $\overline{x}$ refers to a sequence $x_1 \ldots x_n$, and x_i stands for an arbitrary member of this sequence. We assume the usual conventions for variable binding and α-equivalence of types and terms.

As in ML, the type structure of AspectML is divided into *polytypes* and *monotypes*. Polytypes are normally written `<`$\overline{a}$`>` t where $\overline{a}$ is a list of binding type variables and t is a monotype. However, when $\overline{a}$ is empty, we abbreviate `<>` t as t.

In addition to type variables, a, simple base types like `Unit`, `String` and `Stack`, and function types t_1 `->` t_2, the monotypes include `pc` pt, the type of a pointcut, which in turn binds a list of type variables in a pair of monotypes. We explain pointcut types in more detail later. However, note that in AspectML, the word

[1]Details about the full language are available from the documentation that accompanies the implementation.

"monotype" is a slight misnomer for the syntactic category t as some of these types contain internal binding structure.

AspectML expressions include variables x, unit constants `()`, string constants c, function applications, and *let* declarations. New functions may be defined in *let* expressions. These functions may be polymorphic, and they may or may not be annotated with their argument and result types. Furthermore, if the programmer wishes to refer to type parameters within these annotations, they must specify a binding set of type variables `<`$\bar{a}$`>`. When the type annotations are omitted, AspectML will infer them, though that mechanism will not be described in this chapter. It is straightforward to extend idealized AspectML with other features such as integer constants, arithmetic operations, file and network I/O, tuples, and pattern matching, and we will make use of such constructs in our examples.

The most interesting features of our language are pointcuts and advice. Advice in AspectML is second-class and includes two parts: the body, which specifies what to do, and the *pointcut designator*, which specifies when to do it. In AspectML, a pointcut designator has two parts, a *trigger time*, which may either be `before`, `after`, or `around`, and a *pointcut* proper, which is a set of function names. The set of function names may be written out verbatim as `#`$\bar{f}$`#`, or, to indicate all functions, a programmer may use the keyword `any`. In idealized AspectML, it is always necessary to provide a type annotation `#`$\bar{f}$`:`pt`#` on a pointcut formed from a list of functions. In our implementation, this annotation is often not necessary.

The pointcut type, `(<`$\bar{a}$`>` t_1 `~>` t_2`)`, describes the I/O behavior of a pointcut. In AspectML, pointcuts are sets of functions, and t_1 and t_2 approximate the domains and ranges of those functions. For example, if there are functions `f` and `g` with types `String -> String` and `String -> Unit` respectively, the pointcut `#f,g#` has

the pointcut type `pc (<a> String ~> a)`. Because their domains are equal, the type `String` suffices. However, they have different ranges, so we use a polytype that is more general than `String` and `Unit`, `<a> a`. Any approximation is a valid type, so it would have also been fine to annotate the pointcut `#f,g#` with the pointcut type `pc (<a b> a ~> b)`. This latter type is the most general pointcut type, and can be the type for any pointcut, including `any`.

The pointcut designator `before (| #f# |)` represents the point in time immediately before executing a call to the function `f`. Likewise `after (| #f# |)` represents the point in time immediately after execution. The pointcut designator `around (| #f# |)` wraps around the execution of a call to the function `f` – the advice triggered by the pointcut controls whether the function call actually executes or not.

The most basic kind of advice has the form:

$$\texttt{advice}\ tm\ \texttt{(|}e_1\texttt{|) <}\overline{a}\texttt{> (}x\texttt{:}t_1\texttt{,}y\texttt{,}z\texttt{):}t_2\ \texttt{=}\ e_2$$

Here, tm `(|`e_1`|)` is the pointcut designator. When the pointcut designator triggers advice, the variable x is bound either to the argument (in the case of `before` and `around` advice) or to the result of function execution (in the case of `after` advice). [2] The set of binding type variables, `<`$\overline{a}$`>`, allows the types quantified by the pointcut to be named within the advice. However, the binding specification may be omitted if there are no quantified types, or if they are unneeded. The variable x may optionally be annotated with its type t_1. The variable y is bound to the current call stack. We explain stack analysis in Section 2.2.2. The variable z is bound to metadata

[2] After advice traditionally receives the result of the function that triggers it. If the aspect-oriented programmer would like to create *after* advice that depends on a function argument, they would use `around` advice, receiving the argument of the triggering function, that immediately executes the function before executing any new advice code.

describing the function that has been called. In idealized AspectML, this metadata is a string corresponding to the function name as written in the source text, but in the implementation it includes not just the name of the function, but the originating source file and line number. In the future it might also include security information, such as a version number or the name of the code signer. Since advice exchanges data with the designated control flow point, `before` and `after` advice must return a value with the same type as the first argument x. For `around` advice, x has the type of the argument of the triggering function, and the advice must return a value with the result type of the triggering function.

A common use of aspect-oriented programming is to add tracing information to functions. These statements print out information when certain functions are called or return. These trace logs could later be used forensically to decipher security breaches. We can advise the program below to display messages before any function is called and after the functions `f` and `g` return. The trace of the program is shown on the right in comments.

```
(* code *)                  (*  Output trace    *)
fun f x = x + 1             (*  entering g      *)
fun g x = if x then f 1     (*  entering f      *)
                f 0         (*  leaving f => 2  *)
fun h _ = False             (*  leaving g => 2  *)
                            (*  entering h      *)
(* advice *)
advice before (| any |) (arg, _, info) =
  (print ("entering " ^ (getFunName info) ^ "\n"); arg)

advice after (| #f,g# |) (arg, _, info) =
  (print ("leaving " ^ (getFunName info) ^
          " => " ^ (int_to_string arg) ^ "\n");
   arg)

val _ = h (g True)
```

Even though some of the functions in this example are monomorphic, polymorphism is essential. Because the advice can be triggered by any of these functions and they have different types, the advice must be polymorphic. Moreover, since the argument types of functions `f` and `g` have no type structure in common, the argument `arg` of the `before` advice must be completely abstract. On the other hand, the result types of `f` and `g` are identical, so we can fix the type of `arg` to be `Int` in the after advice.

In general, the type of the `after` advice argument may be the most specific type t such that the result types of all functions referenced in the pointcut are

instances of t. Inferring t is not a simple unification problem; instead, it requires *anti-unification* [43, 44]. Our current implementation can often use anti-unification to compute this type.

Finally, we present an example of `around` advice. Again, `around` advice wraps around the execution of a call to the functions in its pointcut designator. The `arg` passed to the advice is the argument that would have been passed to the function had it been called. Finally, `around` advice introduces into the environment the `proceed` function. When applied to a value, `proceed` continues the execution of the advised function with that value as the new argument. Not that `proceed` is not a keyword and may be shadowed by variable binding.

In the following example, a cache is installed "around" the `f` function. First, a cache (`fCache`) is created for the `f` function with the externally-defined `cacheNew` command. Then, `around` advice is installed such that when the `f` function is called, the argument to the function is used as a key in a cache lookup (using the externally-defined `cacheGet` function). If a corresponding entry is found in the cache, the entry is returned as the result of the function. If the corresponding entry is not found, a call to `proceed` is used to invoke the original function. The result of this call is placed in the cache (using the externally-defined `cachePut` function) and is returned as the result of the `f` function.

```
val fCache : Ref List (Int,Int) = cacheNew ()


advice around (| #f# |) (arg, _, _) =
  case (cacheGet (fCache, arg))
    of Some res => res
     | None     => let
                     val res = proceed arg
                     val _ = cachePut (fCache, arg, res)
                   in
                     res
                   end
```

We note that we can transform this example into a general-purpose cache inserter by wrapping the cache creation and `around` advice code in a function that takes a first-class pointcut as its argument as described in Section 2.2.3. This would require a general-purpose hashing function as well. Finally, though not shown here, the `cacheGet` and `cachePut` functions are polymorphic functions that can be called on caches with many types of keys. As such, the key comparisons use a polymorphic equality function that relies on the run-time type analysis described in the next section.

2.2.1 Run-time Type Analysis

We might also want a forensic tracing routine to print not only the name of the function that is called, but also its argument. AspectML makes this extension easy

with an alternate form of advice declaration, called `case-advice`, that is triggered both by the pointcut designator and the specific type of the argument. In the code below, the second piece of advice is only triggered when the function argument is an integer, the third piece of advice is only triggered when the function argument is a boolean, and the first and fourth pieces of advice are triggered by any function call. Advice is maintained as a stack, so all advice that is applicable to a program point is triggered in LIFO order.

```
advice before (| any |)(arg, _, info) = (print "\n"; arg)

case-advice before (| any |)(arg : Int, _, _) =
  (print (" with arg " ^ (int_to_string arg)); arg)

case-advice before (| any |)(arg : Bool, _, _) =
  (print (" with arg "^(if arg then "True" else "False")); arg)

advice before (| any |)(arg, _, info) =
  (print ("entering " ^ (getFunName info)); arg)
```

The code below and its forensic trace demonstrates the execution of the advice. Note that even though `h`'s argument is polymorphic, because `h` is called with an

`Int`, the third advice above triggers instead of the first.

```
(* code *)                  (*  Output trace              *)
fun f x = x + 1             (*                            *)
fun g x = if x then f 1     (*  entering g with arg True  *)
                f 0         (*  entering f with arg 1     *)
fun h _ = False             (*  entering h with arg 2     *)
val _ = h (g True)
```

This ability to conditionally trigger advice based on the type of the argument means that polymorphism is not parametric in AspectML– programmers can analyze the types of values at run-time. However, without this ability we cannot implement this tracing aspect and other similar examples. For further flexibility, AspectML also includes a typecase construct to analyze type variables directly. For example, consider the following advice:

```
advice before (| any |)<a b>(arg : a, _, info) =
  (print ("entering " ^ (getFunName info) ^ "with arg" ^
          (typecase<String> a
             of Int => (int_to_string arg) ^ "\n"
              | Bool => (if arg then "True\n" else "False\n")
              | _    => "<unprintable>\n"));
   arg)
```

This advice is polymorphic, and the argument type `a` is bound by the annotation `<a b>`.[3] Also note that in the example above, to aid typechecking the typecase expression, the return type is annotated with `<String>`.

There is a nice synergy between aspects and run-time type analysis. Converting values to strings is an operation that is generally useful, so one might imagine implementing it as a library function `val_to_string` to be called by the above advice:

```
fun val_to_string <a>(v:a):String =
    typecase<String> a
      of Bool => bool_to_string v
       | String => v
       | Int => int_to_string v
       | (a, b) => "(" ^ (val_to_string (fst v)) ^ ", " ^
                         (val_to_string (snd v)) ^ ")"
       | _ => "<unprintable>"
```

Notice that this solution requires that `val_to_string` be revised every time a new data type is defined by the user (like ML, the full AspectML language allows the creation of new algebraic data types). Instead, we switch to using an `around case-advice` idiom:

[3]Currently this example requires the type variable *b* to be bound, even though it never occurs in the code fragment. The pointcut `any` is of type `pc (<ab> a ~> b)`, and if the programmer wishes to name one of quantified variables, in this case *a*, she must name all of the variables. This is because the quantified variables are unordered and the type inference algorithm cannot naïvely choose which of the two variables to name when only given a single name.

```
fun val_to_string <a> (v:a):String = "<unprintable>"

case-advice around (| #val_to_string# |) (v:Bool, _, _) =
  bool_to_string v

case-advice around (| #val_to_string# |) (v:String, _, _) = v

case-advice around (| #val_to_string# |) (v:Int, _, _) =
  int_to_string v

case-advice around (| #val_to_string# |) (v:(a,b), _, _) =
  "("^(val_to_string (fst v))^", "^(val_to_string (snd v))^")"
```

Each piece of `around case-advice` overrides the default behavior of the function when passed an argument of a particular type. Notice that there are no `proceed` calls – we do not want the function to continue to execute once the correct string conversion function has been selected.

Now, when a programmer defines a data type, they can also update the `val_to_string` function to be able to convert values of their new datatype to a string. For example, we can define a `List` data type and its output function in the

same location[4]

```
datatype List = Cons : <a> a -> List a -> List a
              | Nil  : <a> List a


case-advice around (|#val_to_string#|)(v:List a,_,_):String =
  case v of
    Cons h t => (val_to_string h)^" :: "^(val_to_string t)
  | Nil => "Nil"
```

2.2.2 Reifying the Context

When advice is triggered, often not only is the argument to the function important, but also the context in which it was called. Therefore, this context information is provided to all advice and AspectML includes constructs for analyzing it. For example, below we augment the tracing aspect so that it displays debugging information for the function `f` when it is called directly from `g` and `g`'s argument is the boolean `True`.

```
advice before (| #f# |)(farg, fstk, _) =
  ((case fstk
      of _ :: (| #g# |)(garg, _) :: _ =>
         if garg then
           print "entering f from g(True)\n"
         else ()
       | _ => ());
   farg)
```

[4]Note that AspectML has a different syntax for data type definition than Standard ML. Here a data type definition contains the name of the data type and then a list of the data type constructors, with their types.

The stack argument `fstk` is a list of `Frame`s, which may be examined using case analysis.[5] Each frame in the list `fstk` describes a function in the context and can be matched by a frame pattern: either a wild-card `_` or the pattern `(|`e`|)<`$\overline{a}$`>(`x`,`y`)`. The expression e in a frame pattern must evaluate to a pointcut – the pattern matches if any function in the pointcut matches the function that frame describes. Like in advice, the type variable binders are used to optionally name types quantified by the pointcut. The variable x is the argument of that function, and y is the metadata of the function. The head of the list contains information about the function that triggered the advice (e.g. `f` in the example above).

Consider also the following example, that uses an aspect to implement a stack-inspection-like security monitor for the program. (We will expand the technique of using aspects to provide stack-inspection security in our case study in Section 2.3.) If the program tries to call an operation that has not been enabled by the current context, the security monitor terminates the program. Below, assume the function `enables:FunInfo -> FunInfo -> Bool` determines whether the first argument (a piece of function metadata) provides the capability for the second argument (another piece of function metadata) to execute. We also assume `abort:String -> Unit` terminates the program with an error message.

[5]Idealized AspectML does not make lists primitive, but instead uses the primitive type `Stack` and primitive analysis `stkcase`.

```
fun walk (stk : List Frame, info : FunInfo) =
  case stk of [] => abort "Function not enabled"
            | (| any |)(_, info') :: rest =>
              if (enables info' info) then ()
              else
                walk (rest, info)

advice before (|#f,g,h#|)(arg,stk,info) = (walk (stk,info); arg)
```

2.2.3 First-class Pointcuts

The last interesting feature of our language is the ability to use pointcuts as first-class values. This facility is extremely useful for constructing generic libraries of profiling, tracing or access control advice that can be instantiated with whatever pointcuts are useful for the application. For example, recall the first example in Section 2.2 where we constructed a forensic logger for the `f` and `g` functions. We can instead construct an all-purpose logger that is passed the pointcut designators of the functions we intend to log with the following code. Recall that `val_to_string`

is a function, defined in Section 2.2.1 that converts values of any type to a string.

```
fun startLogger (toLog:pc (<a b> a ~> b)) =
  let advice before (| toLog |)(arg, _, info) =
        ((print ("before " ^ (getFunName info) ^ ": " ^
               (val_to_string arg) ^ "\n")); arg)
      advice after (| toLog |) (res, _, info) =
        ((print ("after " ^ (getFunName info) ^ ":" ^
               (val_to_string res) ^ "\n")); res)
  in () end
```

Another example generalizes the "`f` within `g`" pattern presented above. This is a very common idiom; in fact, AspectJ has a special pointcut designator for specifying it. In AspectML we can implement the `within` combinator using a function that takes two pointcuts – the first for the callee and the second for the caller – as arguments. Whenever we wish to use the `within` combinator, we supply two pointcuts of our choice as shown below.

```
fun within (fpc : pc (<a b> a ~> b),
            gpc : pc (<c> Bool ~> c),
            body : Bool -> Unit) =
  let advice before (| fpc |)(farg, fstk, _) =
        (case fstk
           of _ :: (| gpc |)(garg, _) :: _ => body garg
            | _ => ();
         farg) in () end

fun entering x =
  if x then print "entering f from g\n" else ()

val _ = within (#f#, #g#, entering)
```

Notice that we placed a typing annotation on the formal parameter of `within`. When pointcuts are used as first-class objects, it is not always possible to infer types of function arguments and results. The reason is that pointcut types have binding structure that cannot be determined via unification without the addition of type annotations.

2.2.4 AspectML Implementation

We have implemented an extended version of idealized AspectML for use as a research language. Therefore we included a number of advanced features and libraries necessary for studying the properties and expressiveness of a modern, functional, aspect-oriented language. Our implementation is composed of a series of

several "processes" that consume a stream of data and potentially produce a stream of output. As in the formalized language, a parsed AspectML program is translated in a type-safe manner into a $\mathbb{F}_A$ calculus expression. These $\mathbb{F}_A$ calculus expressions are then evaluated using the reduction rules defined in this chapter. Our implementation is available at `http://www.cs.princeton.edu/sip/projects/aspectml/`

2.2.5 Design Decisions

Anonymous functions Anonymous functions present several design choices for aspect-oriented languages. Because they are nameless, it is impossible to write explicit pointcuts for them. It is possible that generic pointcuts might implicitly advise them, such as `any`, or a new pointcut, `anon` that refers just to anonymous functions. There are no technical difficulties to this extension, but on the other hand, we do not yet see any compelling reasons for advising anonymous functions, either. Therefore, in our implementation, we have decided to make anonymous functions inadvisable until we have more experience with programming in AspectML.

Advising first-class functions Another design choice we made is that explicit pointcuts such as `#f#` may only refer to term variables that were let-bound to functions in the current lexical scope. In other words, as illustrated by the following example, variables corresponding to first-class functions cannot be used in a pointcut.

```
fun f x = x + 1

val ptc = #f#   (* allowed *)


fun h (g : Int -> Int) = #g#  (* not allowed *)
```

We have left this feature out because it increases the complexity of the semantics of AspectML without a corresponding increase in usefulness – we have not found any compelling examples that require advising first-class functions. There are no technical obstacles to advising first-class functions in AspectML. Furthermore, pointcuts are first-class in AspectML, so if the programmer needs access to `g`'s pointcut, `h` can be rewritten to take a pointcut as an additional argument.

Tail recursion Due to the presence of advice and stack analysis, functions in AspectML become more difficult to analyze. First, some traditional program transformations, such as β-reduction are no longer valid in AspectML. Function application can now invoke advice–the evaluation no longer depends solely on the body of the function. Second, the compiler can no longer optimize tail-recursive functions. Because after advice may need to be invoked and the stack may need to be analyzed after each evaluation of a tail-recursive function, the program stack cannot be collapsed in the traditional manner. A potential solution is to include a new type of function declaration in our language. This declaration, perhaps called `tailfun`, could indicate that after advice should not be triggered by the function and that stack analysis should omit the function.

More pointcuts In a larger language, it might be desirable to extend the language of pointcuts to allow advising events other than function invocation. However, given the central importance of functions in AspectML, not as many extensions are necessary as might be needed an imperative language like AspectJ. For example, a Java programmer might find it useful to advise allocating, reading, and writing mutable state. However, all of these behaviors correspond to functions in AspectML that may be directly advised.

2.3 Case Study of the Java Security Mechanism

To demonstrate the usefulness of AspectML, we have implemented a dynamic security policy manager and stack inspection framework with most of the interesting components one finds in Java. In the following sections, we first describe the Java security mechanism and then analyze our implementation of the algorithm in AspectML.

2.3.1 Permissions

The basic unit of protection is the permission. Java security defines many permissions including file system permissions, network socket permissions, system property permissions, and GUI permissions. A permission consists of a permission name and permission arguments that constitute the internal structure of the permission. For example, the file system permission name is `FilePermission`, and the permission arguments contain the access mask (read/write/delete) and the file path.

Permissions consist of granted permissions and requested permissions. Granted permissions represent a list of actions a function is allowed to perform. A requested permission represents a specific action that a function is attempting to perform. If the granted permissions of a function "imply" the requested permission, then the requested action can be performed. The specific mechanisms for granting, requesting, and testing permission implication are described in the following sections.

2.3.2 Policy Parsing

To determine what permissions will be granted to the executing source code, Java security reads from a policy file upon start-up. The policy file consists of a set of

`grant` declarations, each of which specify a segment of source code and the list of permissions granted to that source code. Granted permissions are specified by the name of the permission followed by an list of arguments that describe the details of the permission. We use this same basic format to specify AspectML security files.

The following is the specification of Java and AspectML policy file syntax:

```
grant sourceCodeSpecifier {
  permission permissionName1 permissionArgs1;
  permission permissionName2 permissionArgs2;
  ...
};
...
```

The *sourceCodeSpecifier* selects the source code to which permissions are granted. A policy file writer is allowed to select code by specifying a `codebase` (where the code is located in the file system), a `signer` (the key that has cryptographically signed the code), and a `principal` (the entity that is executing the code). For the purposes of this case study, we have chosen to allow only the `codebase` specifier in AspectML policy files.

The *permissionName* specifies the name of the permission to grant to the selected source code, while the *permissionArgs* specifies the details of the permission. For the purposes of this case study, we have studied file system and network socket

permissions in AspectML. A sample AspectML policy file follows.

```
grant codebase "example/*" {
  permission  FilePermission    "tmp/*", "read write";
  permission  SocketPermission  "*", "listen accept";
};
```

This would give all code in files in the `example/` directory permission to read and write to the `tmp/` directory. It also allows the code in the `example/` directory to start a network server that accepts connections from any host.

2.3.3 Permission Specification

As stated earlier, AspectML security currently specifies file system and network permissions. We have also provided functions in our security implementation that allow a user to add new permission types to the policy file and to the underlying security mechanism. The user must specify the name of the permission (used in policy file parsing), an `addPermission` function of type `String -> permission` that parses the permission arguments from the policy file and returns the resulting permission, and an `impliesPermission` function of type `(permission,permission) -> Bool` that takes in a granted permission and a requested permission and returns whether the first allows the second.

Accordingly, when we added a file system permission type to AspectML security, we specified the name `FilePermission` and an `addFilePermission` function which parses file paths and access mask strings like "read write" to an internal representation. Finally, we specified an `impliesFilePermission` function which takes a granted file system permission such as reading and writing to the `tmp/`

directory and a requested read, write, or delete action and then determines whether the permission allows the action.

2.3.4 Stack Inspection

To determine whether a restricted action should be performed, the permissions granted by the policy file to the currently executing code are examined to determine whether they imply the requested permission required by the restricted action. This test is not enough – if trusted code is called from untrusted code, the untrusted code may perform malicious actions by proxy. Therefore, Java security is a stack inspection mechanism, as described by the pseudocode in Figure 2.2. The current stack frame and all subsequent frames must all have the required permission before an action is approved. Therefore, if the end of the stack is reached, which corresponds to wildcard pattern branch, *True* will be returned indicating that the requested action has been approved.

For example, if function `f` calls function `g` which calls function `h` which then attempts to read from a file, the call stack will look like this: `[fileRead, h, g, f]`. The Java security mechanism will ensure that `h`, `g`, and then `f` all have permission to read from the requested file.

Finally, if trusted code is certain that it can only be used in approved ways, it can disengage any further stack checks by performing a "privileged" action. This is useful, for example, if a trusted function carefully checks its inputs or if the restricted action that it wishes to perform does not depend on data passed to it by its calling function. A "privileged" action is performed by calling a special security function `doPrivileged`, passing it the code to be run. When the security mechanism walks the stack, it will stop at whatever code called the `doPrivileged` function.

Java stack inspection pseudocode

```
public bool inspectStack (Stack currentStack) {
  for (StackFrame sf : currentStack) {
    if (stack frame sf does not allow the action) {
      return false;
    } else if (stack frame sf is marked as "privileged") {
      \\ note that stack frame sf allows the action
      return true;
    }
  }
  return true;
}
```

AspectML stack inspection code

```
fun checkStack (stk, requestedPerm) =
  case stk of
    (| #doPrivileged# |)(_,_) :: (|any|)(_,info) :: _ =>
      let
        val privFun = getFileName info
        val grantedPerms = getPermissions (privFun, policyfile)
      in
        impliesPermissions (grantedPerms, requestedPerm)
      end
  | (|any|)(_,info) :: stktail =>
      let
        val currFun = getFileName info
        val grantedPerms = getPermissions (currFun, policyfile)
      in
        impliesPermissions (grantedPerms, requestedPerm)
        andalso checkStack (stktail, requestedPerm)
      end
  | _ => True
```

Figure 2.2: Stack inspection comparison: Java and AspectML

In the above example, if the function `g` is certain that it cannot be used inappropriately by code that calls it, it can perform a "privileged" call to the `h` function. The stack will appear as `[fileRead, h, doPrivileged, g, f]`. In this case, only `h` and `g` need to have the file read permission – `f` is not examined by the security mechanism.

The AspectML stack inspection algorithm is compared with its Java counterpart in Figure 2.2. As with Java stack inspection, there are three cases, a privileged stack frame, a regular stack frame, and the end of the stack.

2.3.5 Security Triggering

We now have described the policy parsing, permission implication, and stack inspection code. The final step is to trigger this security mechanism when a restricted action is performed. In Java security, every read-file system call in the source code is preceded by a call to `AccessController.checkPermission(requestedAction)`.

In our AspectML implementation, a pointcut is created that contains a list of the restricted actions that should trigger the aspect. For example, all of the system calls that will read from a file will trigger the `readPC` pointcut. Next, an aspect is created that, when the pointcut is triggered (before a restricted action is performed), creates the requested permission and checks that the stack allows this requested permission. In the following example, when code attempts to read from a file, the `readPC` pointcut is triggered, calling the `checkRead` function that creates the

requested `FilePermission` and performs the security stack inspection check.

```
val readPC = #fileCanRead, fileExists, fileIsDirectory,
  fileIsFile, fileLastModified ,fileLength, fileList,
  fileOpenRead#
advice before (| readPC |) (arg, stk, _) =
  if checkRead (stk, arg) then
    arg
  else
    abort "Failed security check"
```

Similar checks are performed when writing files, deleting files, and making and receiving network connections.

2.3.6 Issues

A difficulty we encountered during this stage was similar to a problem also described in another aspect-oriented case study [49] – occasionally the crosscutting location is not easily accessible with a pointcut. In the Java security mechanism, the security checks on the `InetAddress.getLocalHost` and `InetAddress.getHostFromNameService` methods were located in the middle of the methods, not at the beginning or the end of the method. This disallowed easy use of `before` or `after` advice to institute the security check. Instead, we were required to split each affected function into two parts: a function which runs the pre-security-check code, and a second function which is called by the first and runs the post-security-check code. We trigger the security check as `before` advice on this post-security-check function.

Pre-security getAllByName0 Java pseudocode

```
static InetAddress[] getAllByName0 (String host) {...}
```

Post-security getAllByName0 Java pseudocode

```
static InetAddress[] getAllByName0 (String host, boolean check) {
  if (check) { perform security check }
  ...
}

static InetAddress[] getAllByName0 (String host) {
  return getAllByName0(host, true);
}
```

AspectML getAllByName0 security

```
fun shouldCheck (stk) =
  case st of
    (|#checkListen,checkAccept,checkConnect#|)(_,_)::_ => False
  | (|any|) (_,_) :: stktail => shouldCheck stktail
  | _ => True

advice before (| #netGetAllByName0# |) (host, stk, _) =
    if (shouldCheck stk) andalso (not (checkConnect (st,host,~1)))
    then abort "Failed security check"
    else host
```

Figure 2.3: Recursive security: Java and AspectML

Another issue emerged when we discovered that the network security code calls network I/O code which in turn triggers the Java network security code and so on. The Java security mechanism handles this by adding a flag to the argument list of the affected network I/O functions, indicating whether the function is being called from security code or not.

For example, in the top section of Figure 2.3, we display the pre-security pseudocode for the `getAllByName0` function, which looks up the Internet address that correspond to a given hostname string. Because this function both triggers the network security mechanism and is called by it, in the middle section of the figure, `getAllByName0` must be converted to a new function that takes both the hostname string and a flag marking whether the check should be performed or whether `getAllByName0` has been called from within the security mechanism. In addition, the function is overloaded so that calls to the old `getAllByName0` function (calls that are not from within the security mechanism) call the new function with a `check` flag value of `true`

We feel that this solution is suboptimal because it requires modifying the signature of the original network I/O code in order to add the security feature. In our AspectML implementation (displayed in the bottom section of the figure), there is no need to change the argument list of the function. Instead, we add a `shouldCheck` function that performs stack analysis to determine whether the `getAllByName0` function has been called by the security mechanism or not. If it has, then no security check is performed by the advice to avoid infinite recursion.

2.3.7 History Inspection

As an aside, several researchers [48, 21, 1] have questioned whether stack inspection is the correct choice for enforcing common security policies. For example, in order to preserve the confidentiality of certain files, a user may wish to disallow making or receiving network connections after reading from the file system. As any file system reads will have occurred in the past and will no longer be on the stack when a network connection is attempted, a stack inspection mechanism will not suffice to enforce this policy. Instead, the entire execution history – the list of all the functions that have been run – must be examined. AspectML can enforce history-based policies just as easily as the stack-inspection based policies of the previous section. As an example, the following code implements the simple history-based policy for file confidentiality described above.

```
val didRead = ref false

advice after (| #fileOpenRead# |) (res, _, _) =
  ((didRead := true); res)

advice before (|#netConnect,netServerAccept#|)(arg,_,_) =
    if !didRead then
      abort "No network after file read\n"
    else
      arg
```

The first piece of advice sets a flag when any file is read. The second piece of advice disallows all network connections if the flag is set.

2.4 Polymorphic Core Calculus: $\mathbb{F}_A$

In the previous section, we defined the syntax and static semantics for AspectML. One might choose to define the operational semantics for this language directly as a step-by-step term rewriting relation, as is often done for λ-calculi. However, the semantics of certain constructs is very complex. For example, function call, which is normally the simplest of constructs in the λ-calculus, combines the ordinary semantics of functions with execution of advice, the possibility of run-time type analysis and the extraction of metadata from the call stack.

Rather than attempt to specify all of these features directly, we specify the dynamic semantics in stages. First, we show how to compile the high-level constructs into a core calculus, called $\mathbb{F}_A$. The translation breaks down complex high-level objects into substantially simpler, orthogonal concepts. This core calculus is also typed and the translation is type-preserving. Second, we define an operational semantics for the core calculus. Since we have proven that the $\mathbb{F}_A$ type system is sound and the translation from the source is type-preserving, AspectML is safe.

Our core calculus differs from the AspectML in that it is not oblivious – control-flow points that trigger advice must be explicitly annotated. Furthermore, it is explicitly typed – type abstraction and applications must also be explicitly marked in the program, as well as argument types for all functions. Also, we have carefully considered the orthogonality of the core calculus – for example, `case-advice` in AspectML is represented in $\mathbb{F}_A$ with a combination of two orthogonal constructs, primitive advice and type analysis using *typecase* expressions. The primitive advice is always triggered by the pointcut, but the subsequent type analysis determines if the `case-advice` code should be executed or if the advice should simply terminate.

For these reasons, one would not want to program in the core calculus. However, in exchange, the core calculus is much more expressive than the source language.

Because $\mathbb{F}_A$ is so expressive, we can easily experiment with the source language, by adding new features to scale the language up or removing features to improve reasoning power. For instance, by removing the single type analysis construct, we recover a language with parametric polymorphism. In fact, during the process of developing our AspectML, we have made numerous changes. Fortunately, for the most part, we have not had to make many changes in $\mathbb{F}_A$. Consequently, we have not needed to reprove soundness of the target language, only recheck that the translation is type-preserving, a much simpler task. Finally, in our implementation, the type checker for the $\mathbb{F}_A$ has caught many errors in the translation and helped the debugging process tremendously.

In this section, we describe the semantics of $\mathbb{F}_A$, and in Section 2.5, we sketch the translation from AspectML to $\mathbb{F}_A$.

2.4.1 The semantics of explicit join points

The core calculus $\mathbb{F}_A$ is an extension of the core calculus from WZL [37] with polymorphic labels, polymorphic advice, and run-time type analysis. It also improves upon the semantics of context analysis.

For expository purposes, we begin with a simplified version of $\mathbb{F}_A$, and extend it in the following subsections. The initial syntax is summarized below.

$$
\begin{array}{rrcl}
(\textit{types}) & \tau & ::= & 1 \mid \mathsf{string} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \ldots \times \tau_n \mid \alpha \mid \forall\alpha.\tau \mid (\overline{\alpha}.\tau)\ \mathsf{label} \\
& & \mid & (\overline{\alpha}.\tau)\ \mathsf{pc} \mid \mathsf{advice} \\
(\textit{exprs}) & e & ::= & () \mid c \mid x \mid \lambda x;\tau.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid e[\tau] \mid \mathbf{fix}\ x:\tau.e \mid \langle\overline{e}\rangle \\
& & \mid & \mathbf{let}\ \langle\overline{x}\rangle = e_1\ \mathbf{in}\ e_2 \mid \mathbf{new}\ \overline{\alpha}.\tau \leq e \mid \ell \mid \{\overline{e}\} \mid e_1 \cup e_2 \\
& & \mid & e_1[\overline{\tau}][\![e_2]\!] \mid \{e_1[\overline{\alpha}](x:\tau_1, f:\tau_1 \to \tau_2) \rhd e_2\} \mid \Uparrow e \\
& & \mid & \mathbf{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2)
\end{array}
$$

The basis of $\mathbb{F}_A$ is a typed λ-calculus with unit, strings and n-tuples. If $\overline{e}$ is a sequence of expressions $e_1 \ldots e_n$ for $n \geq 2$, then $\langle\overline{e}\rangle$ creates a tuple. The expression $\mathbf{let}\ \langle\overline{x}\rangle = e_1\ \mathbf{in}\ e_2$ binds the contents of a tuple to a vector of variables $\overline{x}$ in the scope of e_2. Unlike WZL, we add impredicative polymorphism to the core calculus, including type abstraction ($\Lambda\alpha.e$) and type application ($e[\tau]$). We write () for the unit value and c for string constants.

Abstract labels, ℓ, play an essential role in the calculus. Labels are used to mark control-flow points where advice may be triggered, with the syntax $\ell[\overline{\tau}][\![e]\!]$. We call such points in the core calculus *join points*. Unlike the labels in WZL, which are designed solely for `before` and `after` advice, labels in our calculus allow `around` advice. The value passed to the join point represents the `proceed` function that can be invoked by advice in the source language. The value returned once the join point executes is a function that executes any advice, or, if there is no advice, is the `proceed` function that was passed to the join point. For example, in the addition expression $v_1 + (\ell[\overline{\tau}][\![e_2]\!]\ v_3)$, after e_2 has been evaluated to a function v_2, evaluation

of the resulting sub-term $\ell[\overline{\tau}][\![v_2]\!]$ returns a function that, when applied to v_3, causes any advice associated with ℓ to be triggered.

Another difference from WZL is that the labels form a tree-shaped hierarchy. The top label in the hierarchy is $\mathcal{U}$. All other labels ℓ sit somewhere below $\mathcal{U}$. If $\ell_1 \leq \ell_2$ then ℓ_1 sits below ℓ_2 in the hierarchy. The expression **new** $\overline{\alpha}.\tau \leq e$ evaluates e, obtaining a label ℓ_2, and generates a new label ℓ_1 defined such that $\ell_1 \leq \ell_2$. This label structure closely resembles the label hierarchy defined by Bruns et al. for their (untyped) μABC calculus [6].

First class labels may be grouped into collections using the label set expression, $\{\overline{e}\}$. Label-sets can then be combined using the union operation, $e_1 \cup e_2$. Label-sets form the basis for specifying when a piece of advice applies.

All advice in $\mathbb{F}_A$ is around advice and exchanges data with a particular join point, making it similar to a function. Note that advice (written $\{e_1[\overline{\alpha}](x : \tau_1, f : \tau_1 \to \tau_2) \rhd e_2\}$) is first-class. The type variables $\overline{\alpha}$ and term variables x and f are bound in the body of the advice e_2. The variable f is bound to a "proceed" function for the around advice, and the variable x is the value that the join point's resulting function will be called upon. The expression e_1 is a label set that describes when the advice is triggered. For example, the advice $\{\{\overline{\ell}\}[](x : \mathsf{int}, f : \mathsf{int} \to \mathsf{int}) \rhd e\}$ is triggered when control-flow reaches a join point marked with ℓ_1, provided ℓ_1 is a descendant of a label in the set $\{\overline{\ell}\}$. If this advice has been installed in the program's dynamic environment, $v_1 + (\ell_1[][\![v_2]\!]\ v_3)$ evaluates to $v_1 + (e[v_2/f][v_3/x])$.

When labels are polymorphic, both types and values are exchanged between labeled control-flow points and advice. For instance, if ℓ_1 is a polymorphic label capable of marking a control-flow point with any type, we might write $v_1 + (\ell_1[\mathsf{int} \to$

Label Subsumption $\Sigma \vdash \ell_1 \leq \ell_2$

$$\frac{\ell ;_1 \overline{\alpha}.\tau \leq \ell_2 \in \Sigma}{\Sigma \vdash \ell_1 \leq \ell_1}\ \mathsf{labsb{:}refl} \qquad \frac{\Sigma \vdash \ell_1 \leq \ell_2 \qquad \Sigma \vdash \ell_2 \leq \ell_3}{\Sigma \vdash \ell_1 \leq \ell_3}\ \mathsf{labsb{:}trans}$$

$$\frac{\ell_1 ; \overline{\alpha}.\tau \leq \ell_2 \in \Sigma}{\Sigma \vdash \ell_1 \leq \ell_2}\ \mathsf{labsb{:}def}$$

Figure 2.4: Label Subsumption in $\mathbb{F}_A$

$\mathsf{int}][\![v_2]\!]\ v_3)$. In this case, if the advice $\{\{\ell_1\}[\alpha](x : \alpha, f : \alpha \to \alpha) \rhd e\}$ has been installed, then the previous expression evaluates to $v_1 + (e[\mathsf{int}/\alpha][v_2/f][v_3/x])$.

Advice is installed into the run-time environment with the expression $\Uparrow e$. Multiple pieces of advice may apply to the same control-flow point, so the order advice is installed in the run-time environment is important. WZL included mechanisms for installing advice both before or after currently installed advice, for simplicity $\mathbb{F}_A$ only allows advice to be installed after.

2.4.2 Operational Semantics

The operational semantics must keep track of both the labels that have been generated and the advice that has been installed. The main operational judgment has the form $(\Sigma; A; e) \mapsto (\Sigma'; A'; e')$. An allocation-style semantics keeps track of the set Σ of labels allocated so far (and their associated types) and A, an ordered list of installed advice. The label hierarchy is determined from the label set Σ by the relation $\Sigma \vdash \ell_1 \leq \ell_2$ in Figure 2.4.

The main rule of the operational semantics, ev:beta in Figure 2.5, decomposes an expression into an evaluation context and primitive reduct. The rules in Figure 2.5

with the form $(\Sigma; A; e) \mapsto_\beta (\Sigma'; A'; e')$ give the primitive β-reductions for expressions in the calculus.

We use the following syntax for values v and evaluation contexts E:

$$
\begin{array}{lrcl}
(\textit{vals}) & v & ::= & () \mid \lambda x; \tau.e \mid \langle \overline{v} \rangle \mid \Lambda\alpha.e \mid \ell \mid \{\overline{v}\} \\
 & & \mid & \{v[\overline{\alpha}](x : \tau_1, f : \tau_1 \to \tau_2) \rhd e\} \\
(\textit{evalctxts}) & E & ::= & [\,] \mid E\ e \mid v\ E \mid E[\tau] \mid \langle E, \ldots, e \rangle \mid \langle v, \ldots, E \rangle \mid E \cup e \mid v \cup E \\
 & & \mid & \{E, \ldots, e\} \mid \{v, \ldots, E\} \mid \textbf{let } \langle \overline{x} \rangle = E \textbf{ in } e \mid E[\overline{\tau}][\![e]\!] \mid v[\overline{\tau}][\![E]\!] \\
 & & \mid & \Uparrow E \mid \{E[\overline{\alpha}](x : \tau, f : \tau_1 \to \tau_2) \rhd e\} \mid \textbf{new } \overline{\alpha}.\tau \leq E
\end{array}
$$

This definition of evaluation contexts gives the core aspect calculus a call-by-value, left-to-right evaluation order, but that choice is orthogonal to the design of the language.

A third judgment form $\Sigma; A; \ell; \tau_1 \to \tau_2 \Rightarrow v$ in Figure 2.6 describes, given a particular label ℓ marking a control-flow point, and type $\tau_1 \to \tau_2$ for the object at that point, how to pick out and compose the advice in context A that should execute at the control-flow point. (Note that the type of the object is not used to select the advice, it merely determines type annotations and instantiations in the result.) The result of this advice composition process is a function v that may be applied to a value with type $(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$. The argument of the function v is the proceed function f with type $(\tau_1 \to \tau_2)$. The result of applying v to the proceed function is a function of type $\tau_1 \to \tau_2$ whose argument (of type τ_1) is passed as the variable x to the advice.

The advice composition judgment is described by three rules. The first composition rule represents when no advice is available, and, when passed a proceed

$\boxed{\beta\text{-reduction } (\Sigma; A; e) \mapsto_\beta (\Sigma'; A'; e')}$

$$\frac{}{(\Sigma; A; (\lambda x; \tau.e)v) \mapsto_\beta (\Sigma; A; e[v/x])}\ \textsf{evb:app}$$

$$\frac{}{(\Sigma; A; (\Lambda\alpha.e)[\tau]) \mapsto_\beta (\Sigma; A; e[\tau/\alpha])}\ \textsf{evb:tapp}$$

$$\frac{}{(\Sigma; A; \mathbf{fix}\ x; \tau.e) \mapsto_\beta (\Sigma; A; e[\mathbf{fix}\ x; \tau.e/x])}\ \textsf{evb:fix}$$

$$\frac{}{(\Sigma; A; \mathbf{let}\ \langle\overline{x}\rangle = \langle\overline{v}\rangle\ \mathbf{in}\ e) \mapsto_\beta (\Sigma; A; e[\overline{v}/\overline{x}])}\ \textsf{evb:let}$$

$$\frac{}{(\Sigma; A; \{\overline{\ell_1}\} \cup \{\overline{\ell_2}\}) \mapsto_\beta (\Sigma; A; \{\overline{\ell_1\ell_2}\})}\ \textsf{evb:union}$$

$$\frac{\ell' \notin \mathrm{dom}(\Sigma)}{(\Sigma; A; \mathbf{new}\ \overline{\alpha}.\tau \leq \ell) \mapsto_\beta (\Sigma, \ell'; \overline{\alpha}.\tau \leq \ell; A; \ell')}\ \textsf{evb:new}$$

$$\frac{}{(\Sigma; A; \Uparrow v) \mapsto_\beta (\Sigma; v, A; \langle\rangle)}\ \textsf{evb:adv-comp}$$

$$\frac{\exists\Theta.\Theta = \textsc{mgu}(\tau_2, \tau_3)}{(\Sigma; A; \mathbf{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2)) \mapsto_\beta (\Sigma; A; \Theta(e_1))}\ \textsf{evb:tcase1}$$

$$\frac{\neg\exists\Theta.\Theta = \textsc{mgu}(\tau_2, \tau_3)}{(\Sigma; A; \mathbf{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2)) \mapsto_\beta (\Sigma; A; e_2[\tau_2/\alpha])}\ \textsf{evb:tcase2}$$

$$\frac{\ell; \overline{\alpha}.\tau_1 \to \tau_2 \leq \ell' \in \Sigma \qquad \Sigma; A; \ell; (\tau_1 \to \tau_2)[\overline{\tau}/\overline{\alpha}] \Rightarrow v'}{(\Sigma; A; \ell[\overline{\tau}][\![v]\!]) \mapsto_\beta (\Sigma; A; v'\, v)}\ \textsf{evb:cut}$$

$\boxed{\text{Reduction } (\Sigma; A; e) \mapsto (\Sigma'; A'; e')}$

$$\frac{(\Sigma; A; e) \mapsto_\beta (\Sigma'; A'; e')}{(\Sigma; A; E[e]) \mapsto (\Sigma'; A'; E[e'])}\ \textsf{ev:beta}$$

Figure 2.5: Operational Semantics for $\mathbb{F}_A$

Advice composition $\Sigma; A; \ell; \tau_1 \to \tau_2 \Rightarrow e$

$$\frac{}{\Sigma; \cdot; \ell; \tau_1 \to \tau_2 \Rightarrow \lambda f; \tau_1 \to \tau_2.f}\ \textsf{adv:empty}$$

$$\frac{\begin{array}{c}\Sigma; A; \ell; \tau_1 \to \tau_2 \Rightarrow v \\ \Sigma \vdash \ell \leq \ell_i \text{ for some } i \qquad \tau_1 \to \tau_2 = \tau_1' \to \tau_2'[\overline{\tau}/\overline{\alpha}]\end{array}}{\begin{array}{l}\Sigma; A, \{\{\overline{\ell}\}[\overline{\alpha}](x : \tau_1', f : \tau_1' \to \tau_2') \rhd e\}; \ell; \tau_1 \to \tau_2 \Rightarrow \\ \quad \lambda f; \tau_1 \to \tau_2.v\ (\lambda x; \tau_1.(e[\overline{\tau}/\overline{\alpha}]))\end{array}}\ \textsf{adv:cons1}$$

$$\frac{\Sigma; A; \ell; \tau_1 \to \tau_2 \Rightarrow v \qquad \Sigma \vdash \ell \not\leq \ell_i}{\Sigma; A, \{\{\overline{\ell}\}[\overline{\alpha}](x : \tau_1', f : \tau_1' \to \tau_2') \rhd e\}; \ell; \tau_1 \to \tau_2 \Rightarrow v}\ \textsf{adv:cons2}$$

Figure 2.6: Advice Composition for $\mathbb{F}_A$

function f and a value x, applies the proceed function f to x. The other rules examine the advice at the head of the advice heap. If the label ℓ is descended from one of the labels in the label set, then that advice is triggered. The head advice is composed with the function produced from examining the rest of the advice in the list. Not only does advice composition determine if ℓ is lower in the hierarchy than some label in the label set, but it also determines the substitution for the abstract types $\overline{\alpha}$ in the body of the advice. The typing rules ensure that if the advice is triggered, this substitution will always exist, so the execution of this rule does not require run-time type information.

Type system

The judgments for well-formed types are straightforward and are described in Figure 2.7. In addition to the standard unit, string, etc. types, there are additional types for labels, pointcuts, advice, and stacks. The same figure also contains the $\mathbb{F}_A$ instance relation, which is similar to the AspectML instance relation.

Well-formed Types $\Delta \vdash \tau$

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha}\ \textsf{wftp:var} \qquad \frac{}{\Delta \vdash 1}\ \textsf{wftp:unit} \qquad \frac{}{\Delta \vdash \textsf{string}}\ \textsf{wftp:str}$$

$$\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}\ \textsf{wftp:arr} \qquad \frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall \alpha.\tau}\ \textsf{wftp:all}$$

$$\frac{\forall i \qquad \Delta \vdash \tau_i}{\Delta \vdash \tau_1 \times \ldots \times \tau_n}\ \textsf{wftp:prod} \qquad \frac{\Delta, \overline{\alpha} \vdash \tau}{\Delta \vdash (\overline{\alpha}.\tau)\ \textsf{label}}\ \textsf{wftp:lab}$$

$$\frac{\Delta, \overline{\alpha} \vdash \tau}{\Delta \vdash (\overline{\alpha}.\tau)\ \textsf{pc}}\ \textsf{wftp:pc} \qquad \frac{}{\Delta \vdash \textsf{advice}}\ \textsf{wftp:advice} \qquad \frac{}{\Delta \vdash \textsf{stack}}\ \textsf{wftp:stk}$$

Instance $\Delta \vdash \overline{\alpha}.\tau_1 \preceq \overline{\beta}.\tau_2$

$$\frac{\Delta, \overline{\alpha} \vdash \tau' \qquad \Delta, \overline{\beta} \vdash \tau'' \qquad (\exists \overline{\tau} \qquad \Delta, \overline{\beta} \vdash \tau_i \qquad \tau'[\overline{\tau}/\overline{\alpha}] = \tau'')}{\Delta \vdash \overline{\alpha}.\tau' \preceq \overline{\beta}.\tau''}\ \textsf{inst}$$

Figure 2.7: Well-formed types in $\mathbb{F}_A$

Well-formed terms $\Delta; \Gamma \vdash e : \tau$

$$\frac{x; \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau}\ \textsf{wft:var} \qquad \frac{}{\Delta; \Gamma \vdash c : \textsf{string}}\ \textsf{wft:str} \qquad \frac{}{\Delta; \Gamma \vdash () : 1}\ \textsf{wft:unit}$$

$$\frac{\Delta; \Gamma, x; \tau \vdash e : \tau \qquad \Delta \vdash \tau}{\Delta; \Gamma \vdash \mathbf{fix}\ x; \tau.e : \tau}\ \textsf{wft:fix} \qquad \frac{\Delta; \Gamma, x; \tau_1 \vdash e : \tau_2 \qquad \Delta \vdash \tau_1}{\Delta; \Gamma \vdash \lambda x; \tau_1.e : \tau_1 \to \tau_2}\ \textsf{wft:abs}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1 e_2 : \tau_2}\ \textsf{wft:app} \qquad \frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \textsf{wft:tabs}$$

$$\frac{\Delta; \Gamma \vdash e : \forall\alpha.\tau \qquad \Delta \vdash \tau'}{\Delta; \Gamma \vdash e[\tau'] : \tau[\tau'/\alpha]}\ \textsf{wft:tapp} \qquad \frac{\Delta; \Gamma \vdash e_i : \tau_i}{\Delta; \Gamma \vdash \langle \overline{e} \rangle : \tau_1 \times \ldots \times \tau_n}\ \textsf{wft:tuple}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \times \ldots \times \tau_n \qquad \Delta; \Gamma, \overline{x; \tau} \vdash e_2 : \tau}{\Delta; \Gamma \vdash \mathbf{let}\ \langle \overline{x} \rangle = e_1\ \mathbf{in}\ e_2 : \tau}\ \textsf{wft:let}$$

$$\frac{\ell; \overline{\alpha}.\tau \in \Gamma}{\Delta; \Gamma \vdash \ell : (\overline{\alpha}.\tau)\ \textsf{label}}\ \textsf{wft:lab} \qquad \frac{\Delta; \Gamma \vdash e : \textsf{advice}}{\Delta; \Gamma \vdash \Uparrow e : 1}\ \textsf{wft:adv-inst}$$

Figure 2.8: Term Typing for $\mathbb{F}_A$: Part 1

Well-formed terms $\Delta; \Gamma \vdash e : \tau$

$$\frac{\Delta; \Gamma \vdash e_i : (\overline{\alpha}_i.\tau_i)\ \mathsf{label} \qquad \Delta \vdash \overline{\beta}.\tau \preceq \overline{\alpha}_i.\tau_i}{\Delta; \Gamma \vdash \{\overline{e}\} : (\overline{\beta}.\tau)\ \mathsf{pc}}\ \mathsf{wft{:}pc}$$

$$\frac{\Delta; \Gamma \vdash e_i : (\overline{\alpha}.\tau_i)\ \mathsf{pc} \qquad \Delta \vdash \overline{\beta}.\tau \preceq \overline{\alpha}.\tau_i}{\Delta; \Gamma \vdash e_1 \cup e_2 : (\overline{\beta}.\tau)\ \mathsf{pc}}\ \mathsf{wft{:}union}$$

$$\frac{\Delta; \Gamma \vdash e : (\overline{\beta}.\tau_2)\ \mathsf{label} \qquad \Delta \vdash \overline{\beta}.\tau_2 \preceq \overline{\alpha}.\tau_1}{\Delta; \Gamma \vdash \mathbf{new}\ (\overline{\alpha}.\tau_1) \leq e : (\overline{\alpha}.\tau_1)\ \mathsf{label}}\ \mathsf{wft{:}new}$$

$$\frac{\Delta; \Gamma \vdash e_1 : (\overline{\alpha}.\tau' \to \tau'')\ \mathsf{label} \qquad \Delta \vdash \tau_i \qquad \Delta; \Gamma \vdash e_2 : (\tau' \to \tau'')[\overline{\tau}/\overline{\alpha}]}{\Delta; \Gamma \vdash e_1[\overline{\tau}][\![e_2]\!] : (\tau' \to \tau'')[\overline{\tau}/\overline{\alpha}]}\ \mathsf{wft{:}cut}$$

$$\frac{\Delta; \Gamma \vdash e_1 : (\overline{\alpha}.\tau_1 \to \tau_2)\ \mathsf{pc} \qquad \Delta, \overline{\alpha}; \Gamma, x; \tau_1, f; \tau_1 \to \tau_2 \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \{e_1[\overline{\alpha}](x : \tau_1, f : \tau_1 \to \tau_2) \rhd e_2\} : \mathsf{advice}}\ \mathsf{wft{:}advice}$$

$$\frac{\begin{array}{c}\Delta, \alpha \vdash \tau_1 \qquad \Delta \vdash \tau_2 \qquad \Delta' = \mathrm{FTV}(\tau_3) - \Delta \\ (\Theta = \textsc{mgu}(\tau_2, \tau_3) \text{ implies } \Delta, \Delta'; \Theta(\Gamma) \vdash \Theta(e_1) : \Theta(\tau_1[\tau_3/\alpha])) \\ \Delta, \alpha; \Gamma \vdash e_2 : \tau_1\end{array}}{\Delta; \Gamma \vdash \mathbf{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2) : \tau_1[\tau_2/\alpha]}\ \mathsf{wft{:}tcase}$$

Figure 2.9: Term Typing for $\mathbb{F}_A$: Part 2

The primary judgment of the $\mathbb{F}_A$ type system, $\Delta;\Gamma \vdash e : \tau$, indicates that the term e can be given the type τ, where free type variables appear in Δ and the types of term variables and labels appear in Γ. The typing rules for this judgment appear in Figures 2.8 and 2.9.

The novel aspect of the $\mathbb{F}_A$ type system is how it maintains the proper typing relationship between labels, label sets and advice. Because data is exchanged between labeled control-flow points and advice, these two entities must agree about the type of data that will be exchanged. To guarantee agreement, we must be careful with the types of labels (Rule wft:lab), which have the form $\overline{\alpha}.\tau$ label. To mark a control-flow point (Rule wft:cut), the label's type τ must be a function type. A label of type $\overline{\alpha}.\tau_1 \to \tau_2$ label may mark control-flow points containing a proceed function of type $\tau_1 \to \tau_2$ and values of any type τ_1, where free type variables $\overline{\alpha}$ are replaced by other types $\overline{\tau}$. For example, a label ℓ with the type $\alpha, \beta.\alpha \to \beta$ label may mark any control flow point, as α and β may be instantiated with any type. For example, below is a well-typed tuple in which ℓ marks two different control flow points, one of type $\gamma \to \gamma$ and the other of type bool $\to$ int:

$$\langle \Lambda\gamma.\lambda x;\gamma.(\ell[\gamma,\gamma][\![\lambda y;\gamma.y]\!]\ x), (\ell[\mathsf{bool},\mathsf{int}][\![\lambda y;\mathsf{bool}.\mathbf{if}\ y\ \mathbf{then}\ 1\ \mathbf{else}\ 0]\!]\ \mathbf{true})\rangle$$

Notice that marking control flow points that occur inside polymorphic functions is no different from marking other control flow points even though ℓ's abstract type variables α and β may be instantiated with different types each time the polymorphic function is called.

Labeling control-flow points correctly is one side of the equation. Constructing sets of labels and using them in advice safely is the other. Typing label set

construction in the core calculus is quite similar to typing pointcuts in the source. Each label in the set must be a generic instance of the type of the set. For example, given labels ℓ_1 of type $1 \to 1$ label and ℓ_2 of type $(1 \to \mathsf{bool})$ label, a label set containing them can be given the type $(\alpha.1 \to \alpha)$ pc because $\alpha.1 \to \alpha$ can be instantiated to either $1 \to 1$ or $1 \to \mathsf{bool}$. The rules for label sets and label set union (wft:pc and wft:union) ensure these invariants.

When typing advice in the core calculus (Rule wft:advice), the advice body must not make unwarranted assumptions about the types and values it is passed from labeled control flow points. Consequently, if the label set e_1 has type $\overline{\alpha}.\tau_1 \to \tau_2$ pc then advice $\{e_1[\overline{\alpha}](f : \tau_1' \to \tau_2', x : \tau_1') \rhd e_2\}$ type checks only when $\tau_1' \to \tau_2'$ is $\tau_1 \to \tau_2$. The type $\tau_1' \to \tau_2'$ cannot be more specific than $\tau_1 \to \tau_2$. If advice needs to refine the type of $\tau_1 \to \tau_2$, it must do so explicitly with type analysis. In this respect the core calculus obeys the principle of *orthogonality*: advice is completely independent of type analysis.

The label hierarchy may be dynamically extended with **new** $\overline{\alpha}.\tau \leq e$ (Rule wft:new). The argument e becomes the parent of the new label. For soundness, there must be a connection between the types of the child and parent labels: the child label must have a more specific type than its parent (written $\Delta \vdash \tau_1 \preceq \tau_2$ if τ_2 is more specific than τ_1). To see how label creation, labeled control flow points and advice are all used together in the core calculus, consider the following example. It creates a new label, installs advice for this label (that calls the proceed function f on its argument x – essentially an identity function) and then uses this label to mark a join point inside a polymorphic function.

$$\begin{array}{l}\textbf{let } l = \textbf{new } \alpha.\alpha \to \alpha \leq \mathcal{U} \textbf{ in} \\ \quad \textbf{let } \_ = \Uparrow \{l[\beta](x : \beta, f : \beta \to \beta) \rhd f\ x\} \textbf{ in} \\ \quad\quad \Lambda\gamma.l[\gamma][\![\lambda z; \gamma.z]\!]\end{array}$$

The **typecase** expression is slightly more general in the core calculus than in the source language. To support the preservation theorem, we must allow arbitrary types, not just type variables, to be the object of scrutiny. In each branch of **typecase**, we know that the scrutinee is the same as the pattern. In the source language, we substituted the pattern for the scrutinized type variable when typechecking the branches. In the core calculus, however, we must compute the appropriate substitution, using the most general unifier (MGU). If no unifier exists, the branch can never be executed. In that case, the branch need not be checked.

The typing rules for the other constructs in the language including strings, unit, functions and tuples are fairly standard.

2.4.3 Stacks and Stack Analysis

Languages such as AspectJ include pointcut operators such as CFlow to enable advice to be triggered in a context-sensitive fashion. In $\mathbb{F}_A$, we not only provide the ability to reify and pattern match against stacks, as in AspectML, but also allow manual construction of stack frames. In fact, managing the structure of the stack is entirely up to the program itself. Stacks are just one possible extension enabled by $\mathbb{F}_A$'s orthogonality.

Stack reification $data(E)$

$$\begin{array}{rcl} data([\,]) & = & \bullet \\ data(\mathbf{store}\ \ell[\overline{\tau}][\![v]\!]\ \mathbf{in}\ E) & = & data(E) :: \ell[\overline{\tau}][\![v]\!] \\ data(E[E']) & = & data(E')\ \text{otherwise} \end{array}$$

β-reduction $(\Sigma; A; e) \mapsto_\beta (\Sigma'; A'; e')$

$$\frac{}{(\Sigma; A; \mathbf{store}\ \ell[\overline{\tau}][\![v_1]\!]\ \mathbf{in}\ v_2) \mapsto_\beta (\Sigma; A; v_2)}\ \textsf{evb:store}$$

$$\frac{\Sigma \vdash v \simeq \varphi \rhd \Theta}{(\Sigma; A; \mathbf{stkcase}\ v\ (\varphi \Rightarrow e_1, x \Rightarrow e_2)) \mapsto_\beta (\Sigma; A; \Theta(e_1))}\ \textsf{evb:scase1}$$

$$\frac{\Sigma \vdash v \not\simeq \varphi \rhd \Theta}{(\Sigma; A; \mathbf{stkcase}\ v\ (\varphi \Rightarrow e_1, x \Rightarrow e_2)) \mapsto_\beta (\Sigma; A; e_2[v/x])}\ \textsf{evb:scase2}$$

Reduction $(\Sigma; A; e) \mapsto (\Sigma'; A'; e')$

$$\frac{data(E) = v}{(\Sigma; A; E[\mathbf{stack}]) \mapsto (\Sigma; A; E[v])}\ \textsf{ev:stk} \qquad \frac{(\Sigma; A; e) \mapsto_\beta (\Sigma'; A'; e')}{(\Sigma; A; E[e]) \mapsto (\Sigma'; A'; E[e'])}\ \textsf{ev:beta}$$

Stack-matching $\Sigma \vdash v \simeq \varphi \rhd \Theta$

$$\frac{}{\Sigma \vdash \bullet \simeq \bullet \rhd \cdot}\ \textsf{sm:nil}$$

$$\frac{\begin{array}{c}\Sigma \vdash v_2 \simeq \varphi \rhd \Theta \\ \ell; \overline{\beta}.\tau_2 \leq \ell' \in \Sigma \qquad \Sigma \vdash \ell \leq \ell_i \text{ for some } i \qquad \exists \overline{\sigma}.\tau_2[\overline{\tau}/\overline{\beta}] = \tau_1[\overline{\sigma}/\overline{\alpha}]\end{array}}{\Sigma \vdash \ell[\overline{\tau}][\![v_1]\!]::v_2 \simeq \{\overline{\ell}\}[\overline{\alpha}][\![x]\!]; \tau_1::\varphi \rhd \Theta, \overline{\sigma}/\overline{\alpha}, v_1/x}\ \textsf{sm:cons}$$

$$\frac{\Sigma \vdash v' \simeq \varphi \rhd \Theta}{\Sigma \vdash \ell[\overline{\tau}][\![v]\!]::v' \simeq \_::\varphi \rhd \Theta}\ \textsf{sm:wild} \qquad \frac{}{\Sigma \vdash v \simeq x \rhd \cdot, v/x}\ \textsf{sm:var}$$

Figure 2.10: Stack Operational Semantics for $\mathbb{F}_A$

Well-formed terms $\Delta; \Gamma \vdash e : \tau$

$$\frac{\Delta; \Gamma \vdash e_1 : (\overline{\alpha}.\tau)\ \mathsf{label} \qquad \Delta \vdash \tau_i \qquad \Delta; \Gamma \vdash e_2 : \tau[\overline{\tau}/\overline{\alpha}] \qquad \Delta; \Gamma \vdash e_3 : \tau'}{\Delta; \Gamma \vdash \mathbf{store}\ e_1[\overline{\tau}][\![e_2]\!]\ \mathbf{in}\ e_3 : \tau'}\ \mathsf{wft{:}store}$$

$$\frac{}{\Delta; \Gamma \vdash \mathbf{stack} : \mathsf{stack}}\ \mathsf{wft{:}stk} \qquad \frac{}{\Delta; \Gamma \vdash \bullet : \mathsf{stack}}\ \mathsf{wft{:}stk\text{-}nil}$$

$$\frac{\ell; \overline{\alpha}.\tau \in \Gamma \qquad \Delta \vdash \tau_i \qquad \Delta; \Gamma \vdash v_1 : \tau[\overline{\tau}/\overline{\alpha}] \qquad \Delta; \Gamma \vdash v_2 : \mathsf{stack}}{\Delta; \Gamma \vdash \ell[\overline{\tau}][\![v_1]\!]::v_2 : \mathsf{stack}}\ \mathsf{wft{:}stk\text{-}cons}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \mathsf{stack} \\ \Delta; \Gamma \vdash \rho \dashv \Delta'; \Gamma' \qquad \Delta, \Delta'; \Gamma, \Gamma' \vdash e_2 : \tau \qquad \Delta; \Gamma, x; \mathsf{stack} \vdash e_3 : \tau\end{array}}{\Delta; \Gamma \vdash \mathbf{stkcase}\ e_1\ (\rho \Rightarrow e_2, x \Rightarrow e_3) : \tau}\ \mathsf{wft{:}scase}$$

Well-formed patterns $\Delta; \Gamma \vdash \rho \dashv \Delta'; \Gamma'$

$$\frac{}{\Delta; \Gamma \vdash \bullet \dashv \cdot; \cdot}\ \mathsf{wfpt{:}nil} \qquad \frac{}{\Delta; \Gamma \vdash x \dashv \cdot; \cdot, x; \mathsf{stack}}\ \mathsf{wfpt{:}var}$$

$$\frac{\Delta; \Gamma \vdash \rho \dashv \Delta'; \Gamma'}{\Delta; \Gamma \vdash \_::\rho \dashv \Delta'; \Gamma'}\ \mathsf{wfpt{:}wild}$$

$$\frac{\Delta; \Gamma \vdash e : (\overline{\alpha}.\tau)\ \mathsf{pc} \qquad \Delta; \Gamma \vdash \rho \dashv \Delta'; \Gamma'}{\Delta; \Gamma \vdash e[\overline{\alpha}][\![x]\!]; \tau::\rho \dashv \Delta', \overline{\alpha}; \Gamma', x : \tau}\ \mathsf{wfpt{:}store}$$

Figure 2.11: Stack typing for $\mathbb{F}_A$

WZL's monomorphic core calculus also contained the ability to query the stack, but the stack was not first-class and queries had to be formulated as regular expressions. Our pattern matching facilities are simpler and more general. Moreover, they fit perfectly within the functional programming idiom.

Below are the necessary new additions to the syntax of $\mathbb{F}_A$ for storing type and value information on the stack, capturing and representing the current stack as a data structure, and analyzing a reified stack. The operational rules for execution of stack commands may be found in Figure 2.10 and the typing rules in Figure 2.11.

$$
\begin{aligned}
\tau &::= \ldots \mid \mathsf{stack} \\
e &::= \ldots \mid \mathbf{stack} \mid \bullet \mid \ell[\overline{\tau}][\![v_1]\!];;v_2 \mid \mathbf{store}\ e_1[\overline{\tau}][\![e_2]\!]\ \mathbf{in}\ e_3 \\
&\quad \mid \mathbf{stkcase}\ e_1\ (\rho \Rightarrow e_2, x \Rightarrow e_3) \\
E &::= \ldots \mid \mathbf{store}\ v[\overline{\tau}][\![E]\!]\ \mathbf{in}\ e \mid \mathbf{store}\ v_1[\overline{\tau}][\![v_2]\!]\ \mathbf{in}\ E \\
&\quad \mid \mathbf{stkcase}\ E\ (\rho \Rightarrow e_1, x \Rightarrow e_2) \\
&\quad \mid \mathbf{stkcase}\ v\ (P \Rightarrow e_1, x \Rightarrow e_2) \\
\rho &::= \bullet \mid e[\overline{\alpha}][\![y]\!];\tau;;\rho \mid x \mid \_;;\rho \\
\varphi &::= \bullet \mid v[\overline{\alpha}][\![y]\!];;\varphi \mid x \mid \_;;\varphi \\
P &::= E[\overline{\alpha}][\![y]\!];;\varphi \mid e[\overline{\alpha}][\![y]\!];;P \mid \_;;P
\end{aligned}
$$

Data is explicitly allocated on the stack using the command $\mathbf{store}\ e_1[\overline{\tau}][\![e_2]\!]\ \mathbf{in}\ e_3$, where e_1 is a label. Because this label may be polymorphic, it must be instantiated with type arguments $\overline{\tau}$. e_2 represents a value associated with the label and is typically used to store the value passed into the control flow point marked by the label. The **store** command evaluates e_1 to a label l and e_2 to a value v_2, places $\ell[\overline{\tau}][\![v_1]\!]$ on the stack, evaluates e_3 to a value v_3 and finally removes $\ell[\overline{\tau}][\![v_1]\!]$ from the stack and returns v_3. The term **stack** captures the data stored in its execution context E as a first-class data structure. This context is converted into a data

structure, using the auxiliary function $data(E)$. We represent a stack using the list with terms $\bullet$ for the empty list and ;; (cons) to prefix an element onto the front of the list. A list of stored stack information may be analyzed with the pattern matching term **stkcase** e_1 $(\rho \Rightarrow e_2, x \Rightarrow e_3)$. This term attempts to match the pattern ρ against e_1, a reified stack. Note that stack patterns, ρ, include first-class pointcuts so they must be evaluated to pattern values, φ, to resolve these pointcuts before matching.

If, after evaluation, the pattern value successfully matches the stack, then the expression e_2 evaluates, with its pattern variables replaced with the corresponding part of the stack. Otherwise execution continues with e_3. These rules rely on the stack matching relation $\Sigma \vdash v \simeq \varphi \rhd \Theta$ that compares a stack pattern value φ with a reified stack v to produce a substitution Θ.

2.4.4 Type Safety

We have shown that $\mathbb{F}_A$ is type sound through the usual Progress and Preservation theorems. Figure 2.12 describes the typing rules for the term variable and label context Γ, the label heap Σ, and the advice heap A. It should be noted that Γ and Σ always contain the top label $\mathcal{U}$. We use the judgment $\vdash (\Sigma; A; e)$ ok in Rule (wfcfg) to denote a well-formed abstract machine state.

Lemma 2.4.1 (Inversion) *The rules in the following judgments are invertible: well-formed types, generalization, variable contexts, label heaps, advice heaps, term typing, patterns, machine configurations, stack data, β -reductions, context reductions, and stack matching. The rules in the judgments for the label subsumption and advice composition rules are not invertible.*

Proof: By inspection of the rules for each judgment. □

Well-formed Term Variable and Label Context $\Delta \vdash \Gamma$

$$\frac{}{\Delta \vdash \mathcal{U}; \alpha.\alpha}\ \textsf{wfc:base} \qquad \frac{\Delta \vdash \tau \qquad \Delta \vdash \Gamma}{\Delta \vdash \Gamma, x; \tau}\ \textsf{wfc:var}$$

$$\frac{\Delta, \overline{\alpha} \vdash \tau \qquad \Delta \vdash \Gamma}{\Delta \vdash \Gamma, \ell; \overline{\alpha}.\tau}\ \textsf{wfc:lab}$$

Well-formed Label Heaps $\vdash \Sigma : \Gamma$

$$\frac{}{\vdash (\mathcal{U}; \alpha.\alpha \leq \mathcal{U}) : (\mathcal{U}; \alpha.\alpha)}\ \textsf{wflh:base}$$

$$\frac{\ell_2; \overline{\beta}.\tau_2 \leq \ell_3 \in \Sigma \qquad \cdot \vdash \overline{\beta}.\tau_2 \preceq \overline{\alpha}.\tau_1 \qquad \vdash \Sigma : \Gamma}{\vdash (\Sigma, \ell_1; \overline{\alpha}.\tau_1 \leq \ell_2) : (\Gamma, \ell_1; \overline{\alpha}.\tau_1)}\ \textsf{wflh:cons}$$

Well-formed Advice Heaps $\Gamma \vdash A$ ok

$$\frac{}{\Gamma \vdash \cdot \text{ ok}}\ \textsf{wfah:base} \qquad \frac{\cdot; \Gamma \vdash v : \textsf{advice} \qquad \Gamma \vdash A \text{ ok}}{\Gamma \vdash A, v \text{ ok}}\ \textsf{wfah:cons}$$

Well-formed Machine Configurations $\vdash (\Sigma; A; e)$ ok

$$\frac{\vdash \Sigma : \Gamma \qquad \Gamma \vdash A \text{ ok} \qquad \cdot; \Gamma \vdash e : \tau}{\vdash (\Sigma; A; e) \text{ ok}}\ \textsf{wfcfg}$$

Figure 2.12: Well-formed Machine Configurations in $\mathbb{F}_A$

Progress. In this section, we present the lemmas used to prove Progress Theorem 2.4.11.

The following lemma says that if a hierarchy relation between two labels is well-typed with regards to the label store, both labels exist in the store.

Lemma 2.4.2 (Label subsumption) *If* $\vdash \Sigma : \Gamma$ *and* $\Sigma \vdash \ell_1 \leq \ell_2$ *then* $\ell_1 : \overline{\alpha}.\tau_1 \leq \ell_1' \in \Sigma$ *and* $\ell_2 : \overline{\beta}.\tau_2 \leq \ell_2' \in \Sigma$.

Proof: Straightforward induction on the structure of $\Sigma \vdash \ell_1 \leq \ell_2$. □

The following lemma states that if a label is lower in the hierarchy than another, the type of the higher label is more general (more polymorphic) than the lower.

Lemma 2.4.3 (Label generalization) *If* $\vdash \Sigma : \Gamma$ *and* $\Sigma \vdash \ell_1 \leq \ell_2$ *and* $\ell_1 : \overline{\alpha}.\tau_1 \leq \ell_1' \in \Sigma$ *and* $\ell_2 : \overline{\beta}.\tau_2 \leq \ell_2' \in \Sigma$ *then* $\cdot \vdash \overline{\beta}.\tau_2 \preceq \overline{\alpha}.\tau_1$.

Proof: By induction on the structure of $\Sigma \vdash \ell_1 \leq \ell_2$, with use of the Inversion Lemma 2.4.1 and the Label Subsumption Lemma 2.4.2. □

The following lemma states that if one type is more general (more polymorphic) than a second, and the second more general (more polymorphic) than a third, that the first type is more general (more polymorphic) than the third.

Lemma 2.4.4 (Instance transitivity) *If* $\Delta \vdash \overline{\alpha}.\tau_1 \preceq \overline{\beta}.\tau_2$ *and* $\Delta \vdash \overline{\beta}.\tau_2 \preceq \overline{\gamma}.\tau_3$ *then* $\Delta \vdash \overline{\alpha}.\tau_1 \preceq \overline{\gamma}.\tau_3$.

Proof: Straightforward, with uses of the Inversion Lemma 2.4.1. □

The following lemma states that if a label is lower in the label hierarchy than a label in a pointcut, the pointcut type is more general (more polymorphic) than than the label type.

Lemma 2.4.5 (Pointcut match progress) *If* $\vdash \Sigma : \Gamma$ *and* $(\cdot \vdash \overline{\tau_i})^{1 \leq i \leq n}$ *and* $\ell; \overline{a}.\tau \leq \ell' \in \Sigma$ *and* $\cdot; \Gamma \vdash \{\overline{\ell}\} : (\overline{\beta}.\tau')$ pc *and* $\Sigma \vdash \ell \leq \ell_j$ *and* $(\cdot \vdash \overline{\tau_i'})^{1 \leq i \leq n}$
then $\tau[\overline{\tau}/\overline{a}] = \tau'[\overline{\tau'}/\overline{\beta}]$.

Proof: Straightforward, with uses of the Inversion Lemma 2.4.1, the Label Generalization Lemma 2.4.3 and the Instance Transitivity Lemma 2.4.4. □

The following lemma states that the advice triggering mechanism cannot get "stuck." If a label in a joinpoint triggers a pointcut in a piece of advice, then the pointcut type is more general (more polymorphic) than the label type. It is a straightforward use of the previous lemma.

Lemma 2.4.6 (Cut progress) *If* $\vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A, \{\{\overline{\ell}\}.\overline{\beta}x; \tau' \rightarrow e\}$ ok *and* $\cdot; \Gamma \vdash \ell[\overline{\tau}][\![v]\!] : \tau[\overline{\tau}/\overline{\alpha}]$ *and* $\Sigma \vdash \ell \leq \ell_j$ *and* $(\cdot \vdash \overline{\tau_i'})^{1 \leq i \leq n}$
then $\tau[\overline{\tau}/\overline{a}] = \tau'[\overline{\tau'}/\overline{\beta}]$.

Proof: Straightforward use of the Pointcut Match Progress Lemma 2.4.5, with uses of the Inversion Lemma 2.4.1. □

The following lemma states that the stack pattern matching mechanism cannot get stuck. If a label in a stack matches a stack pattern, then the stack pattern type is more general (more polymorphic) than the label type. Like the previous lemma, it is a straightforward use of the Pointcut Match Progress Lemma 2.4.5.

Lemma 2.4.7 (Stack-case progress) *If* $\vdash \Sigma : \Gamma$ *and*
$c \; \cdot; \Gamma \vdash$ **stkcase** $\ell[\overline{\tau}][\![v_1]\!]; ; v_2 \; (\{\overline{\ell}\}[\overline{\beta}][\![x]\!]; \tau'; ; \rho \Rightarrow e_2, x \Rightarrow e_3) : \tau$ *and* $\Sigma \vdash \ell \leq \ell_j$ *and* $(\cdot \vdash \overline{\tau_i'})^{1 \leq i \leq n}$
then $\tau[\overline{\tau}/\overline{a}] = \tau'[\overline{\tau'}/\overline{\beta}]$.

Proof: Straightforward use of the Pointcut Match Progress Lemma 2.4.5, with uses of the Inversion Lemma 2.4.1. □

The following lemma states that if a value can be given a particular type, it has a particular form.

Lemma 2.4.8 (Canonical forms) *Suppose that $v : \tau$ is a closed, well-formed value and τ is a closed, well-formed type.*

- *If $\tau = 1$, then $v = ()$.*
- *If $\tau = \mathsf{string}$, then $v = s$.*
- *If $\tau = \tau_1 \to \tau_2$, then $v = \lambda x; \tau_1.e$.*
- *If $\tau = \forall\alpha.\tau'$, then $v = \Lambda\alpha.e$.*
- *If $\tau = (\overline{\alpha}.\tau')\ \mathsf{label}$, then $v = \ell$.*
- *If $\tau = (\overline{\alpha}.\tau')\ \mathsf{pc}$, then $v = \{\overline{v}\}$.*
- *If $\tau = \mathsf{advice}$, then $v = \{v'.\overline{\alpha}x; \tau' \to e\}$.*
- *If $\tau = \mathsf{stack}$, then either $v = \bullet$ or $\ell[\overline{\tau'}][\![v']\!];;v''$.*
- *If $\tau = \tau_1 \times \ldots \times \tau_n$, then $v = \langle\overline{v}\rangle$.*

Proof: By induction on the structure of $\Delta; \Gamma \vdash v : \tau$, using the fact that v is a value. □

The following lemma states that if a label store and an expression is well-typed, then the expression is either a value, or is an evaluation context with an inner expression that can take a step.

Lemma 2.4.9 (Context decomposition) *If* $\vdash \Sigma : \Gamma$ *and* $\cdot; \Gamma \vdash e : t$ *then* e *is a value or* $E[e']$ *where* e' *is either* **stack** *or the left-hand side of one of the* β*-reduction rules.*

Proof: By induction on on the structure of $\cdot; \Gamma \vdash e : t$ □

The following lemma states that if a label store, an aspect store, and an expression are well-formed, then the expression is a value or the machine state can take a step.

Lemma 2.4.10 (Progress lemma) *If* $\vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A$ ok *and* $\cdot; \Gamma \vdash e : \tau$ *then either* e *is a value, or there exists another configuration* $(\Sigma'; A'; e')$ *such that* $(\Sigma; A; e) \mapsto (\Sigma'; A'; e')$.

Proof: By induction on the structure of $\Delta; \Gamma \vdash e : \tau$, with uses of the Inversion Lemma 2.4.1 and the Canonical Forms Lemma 2.4.8, and the Context Decomposition Lemma 2.4.9.

- Case wft:cut uses the Cut Progress Lemma 2.4.6 to show that the advice triggering mechanism cannot get "suck."
- Case wft:scase uses the Stack-case Progress Lemma 2.4.7 to show that the stack pattern matching mechanism cannot get "stuck."

□

The Progress Theorem states that if a machine configuration is well-formed, then either the expression in it is a value, or the machine configuration can take a step. It is a straightforward use of the previous lemma.

Theorem 2.4.11 (Progress) *If* $\vdash (\Sigma; A; e)$ ok
then either e *is a value, or there exists another configuration* $(\Sigma'; A'; e')$ *such that* $(\Sigma; A; e) \mapsto (\Sigma'; A'; e')$.

Proof: Straightforward use of the Progress Lemma 2.4.10, with uses of the Inversion Lemma 2.4.1. □

Preservation. In this section, we present the lemmas used to prove Preservation Theorem 2.4.23.

The following definitions state what it means for one environment or store to "extend" another environment or store. The extended environment is a superset of the original environment.

Definition 2.4.12 (Γ' extends Γ) *If* $dom(\Gamma) \subseteq dom(\Gamma')$ *and* $\forall x \in dom(\Gamma), \Gamma(x) = \Gamma'(x)$, *and* $\forall l \in dom(\Gamma), \Gamma(l) = \Gamma'(l)$,
then Γ' *extends* Γ.

Definition 2.4.13 (Σ' extends Σ) *If* $dom(\Sigma) \subseteq dom(\Sigma')$ *and* $\forall l \in dom(\Sigma), \Sigma(l) = \Sigma'(l)$,
then Σ' *extends* Σ.

Definition 2.4.14 (A' extends A) *If* $\forall v \in A$, $v \in A'$,
then A' *extends* A.

The following lemma states that if an evaluation context is well-typed, then the expression inside the evaluation context is also well-typed.

Lemma 2.4.15 (Evaluation context inversion) *If* $\Delta; \Gamma \vdash E[e] : \tau$
then $\Delta; \Gamma \vdash e : \tau'$.

Proof: By induction on the structure of E, with uses of the Inversion Lemma 2.4.1. □

The following lemma states that if an evaluation context is well-typed, and the expression inside the evaluation context is also well-typed, then substituting a different inner expression with the same type, preserves the type of the evaluation context.

Lemma 2.4.16 (Evaluation context substitution) *If* $\Delta; \Gamma \vdash E[e] : \tau$ *and* $\Delta; \Gamma \vdash e : \tau'$ *and* $\Delta; \Gamma' \vdash e' : \tau'$ *and* Γ' *extends* Γ *then* $\Delta; \Gamma' \vdash E[e'] : \tau$.

Proof: By induction on the structure of E, with uses of the Inversion Lemma 2.4.1. □

The following lemma states that if an evaluation context is well-typed, then running the *data* function on the evaluation context returns a value with type stack.

Lemma 2.4.17 (Data function typing) *If* $\cdot; \Gamma \vdash E[e] : \tau$ *and* $data(E) = v$ *then* $\cdot; \Gamma \vdash v : \mathsf{stack}$.

Proof: By induction on the structure of the $data(E)$ function, with uses of the Inversion Lemma 2.4.1 and the Evaluation Context Inversion Lemma 2.4.15. □

The following lemma states if a stack is matched to a stack pattern to produce a substitution, then the substitution preserves typing.

Lemma 2.4.18 (Pattern matching) *If* $\vdash \Sigma : \Gamma''$ *and* Γ *extends* Γ'' *and* $\Delta \vdash \Gamma$ *and* $\Delta; \Gamma \vdash v : \mathsf{stack}$ *and* $\Delta; \Gamma \vdash \rho \dashv \Delta'; \Gamma'$ *and* $\Delta, \Delta'; \Gamma, \Gamma' \vdash e : \tau$ *and* $\Sigma \vdash v \simeq \rho \triangleright \Theta$ *then* $\Delta; \Gamma \vdash \Theta(e) : \tau$.

Proof: By induction on the structure of $\Sigma \vdash v \simeq \rho \rhd \Theta$, with uses of the Inversion Lemma 2.4.1. □

The following lemma states that when the advice composition function returns an "around" function when triggered by a joinpoint, the "around" function has the proper type to replace the joinpoint.

Lemma 2.4.19 (Advice composition) *If* $\vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A$ ok *and* $\cdot; \Gamma \vdash \ell : (\overline{\alpha}.\tau)$ label *and* $\Sigma; A; \ell; \tau[\overline{\tau}/\overline{\alpha}] \Rightarrow v$ *and* $(\cdot \vdash \tau_i)^{1 \leq i \leq n}$
then $\cdot; \Gamma \vdash v : \tau[\overline{\tau}/\overline{\alpha}] \rightarrow \tau[\overline{\tau}/\overline{\alpha}]$.

Proof: By induction on the structure of $\Sigma; A; \ell; \tau[\overline{\tau}/\overline{\alpha}] \Rightarrow v$, with uses of the Inversion Lemma 2.4.1. □

The following lemma states that when type variables are replaced by well-formed types in an expression typing judgment, the expression typing judgment still holds.

Lemma 2.4.20 (Type Substitution) *If* $\Delta, \alpha; \Gamma \vdash e : \tau$ *and* $\Delta \vdash \tau'$
then $\Delta; \Gamma[\tau'/\alpha] \vdash e[\tau'/\alpha] : \tau[\tau'/\alpha]$

Proof: By induction on the structure of $\Delta; \Gamma \vdash e : \tau$. □

The following lemma states that if a label store, advice store, and expression of a machine configuration are well-typed and take a step using the β-redex rules, the resulting machine configuration is also well-typed.

Lemma 2.4.21 (β-redex preservation) *If* $\vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A$ ok *and* $\cdot; \Gamma \vdash e : \tau$ *and* $(\Sigma; A; e) \mapsto_\beta (\Sigma'; A'; e')$
then $\vdash \Sigma' : \Gamma'$ *and* $\Gamma' \vdash A'$ ok *and* $\cdot; \Gamma' \vdash e' : \tau$ *and* Γ' *extends* Γ.

Proof: By induction on the structure of $(\Sigma; A; e) \mapsto_\beta (\Sigma'; A'; e')$, with uses of the Inversion Lemma 2.4.1, the Instance Transitivity Lemma 2.4.4, and the Type Substitution Lemma 2.4.20.

- Case evb:scase1 uses the Pattern Matching Lemma 2.4.18 to show that the stack pattern machine algorithm preserves typing.
- Case evb:cut uses the Advice Composition Lemma 2.4.19 to show that the advice composition algorithm preserves typing.

□

The following lemma states that if a label store, advice store, and expression of a machine configuration are well-typed and take a step using the reduction rules, the resulting machine configuration is also well-typed.

Lemma 2.4.22 (Preservation lemma) *If* $\vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A$ ok *and* $\cdot; \Gamma \vdash e : \tau$ *and* $(\Sigma; A; e) \mapsto (\Sigma'; A'; e')$
then $\vdash \Sigma' : \Gamma'$ *and* $\Gamma' \vdash A'$ ok *and* $\cdot; \Gamma' \vdash e' : \tau$.

Proof: By induction on the structure of $(\Sigma; A; e) \mapsto (\Sigma'; A'; e')$, with uses of the Inversion Lemma 2.4.1.

- Case ev:beta uses the Evaluation Context Inversion Lemma 2.4.15, the Evaluation Context Substitution Lemma 2.4.16, and the β-redex Preservation Lemma 2.4.21 to show that using the β-redex rules on an expression inside an evaluation context preserves typing.
- Case ev:stk uses the Data Typing Lemma 2.4.17 to show that running the *data* function on an evaluation context returns a value that preserves typing.

□

The following lemma states that if a machine configuration is well-formed and takes a step, then the resulting machine configuration is also well-formed. It is a straightforward use of the previous lemma.

Theorem 2.4.23 (Preservation) *If* $\vdash (\Sigma; A; e)$ ok *and* $(\Sigma; A; e) \mapsto (\Sigma'; A'; e')$, *then* Σ' *and* A' *extend* Σ *and* A *such that* $\vdash (\Sigma'; A'; e')$ ok.

Proof: Straightforward use of the Preservation Lemma 2.4.22, with uses of the Inversion Lemma 2.4.1. □

The end result is that well-typed programs do not go wrong.

2.5 Translation from AspectML to $\mathbb{F}_A$

2.5.1 Definition of Translation

We give a semantics to well-typed AspectML programs by defining a type-preserving translation into the $\mathbb{F}_A$ language. This translation is defined by the following mutually recursive judgments for over terms, types, patterns, declarations and pointcut designators.

Throughout the translation we assume that there exists an implicit injection from AspectML type and term variables ($a, b, \ldots$ and $x, y, \ldots$) and $\mathbb{F}_A$ type and term variables ($\alpha, \beta, \ldots$ and $x, y, \ldots$).

We begin by defining several translation abbreviations in Figure 2.13. These allow us to specify function abstraction and application translation rules, type abstraction and application rules, **stkcase** and **typecase** rules, and other rules more succinctly.

Simple abbreviations

$$\begin{array}{rcl}
\textbf{let } x : \tau = e_1 \textbf{ in } e_2 & \triangleq & (\lambda x; \tau.e_2)e_1 \\
\forall \overline{a}.\tau & \triangleq & \forall\alpha_1 \ldots \forall\alpha_n.\tau \\
\Lambda\overline{a}.e & \triangleq & \Lambda\alpha_1 \ldots \forall\alpha_n.e \\
e\overline{[\tau]} & \triangleq & e[\tau_1] \ldots [\tau_n] \\
_ & \triangleq & x
\end{array}$$

(where x fresh)

Multi-arm **stkcase** abbreviation **stkcase** e_1' $(\overline{\rho} \Rightarrow \overline{e}, \_ \Rightarrow e_2')$

$$\begin{array}{l}
\textbf{stkcase } e_1 \ (\overline{\rho} \Rightarrow \overline{e}, _ \Rightarrow e_2) \triangleq \\
\quad \textbf{let } y : \mathsf{stack} = e_1 \textbf{ in } \textbf{stkcase}' \ y \ (\overline{\rho} \Rightarrow \overline{e}, _ \Rightarrow e_2)
\end{array}$$

$$\begin{array}{ll}
\textbf{stkcase}' \ e_1 \ (\rho \Rightarrow e_2, x \Rightarrow e_3) & \triangleq \textbf{stkcase } e_1 \ (\rho \Rightarrow e_2, _ \Rightarrow e_3) \\
\textbf{stkcase}' \ e_1 \ (\rho \Rightarrow e_2, \overline{\rho} \Rightarrow \overline{e}, x \Rightarrow e_2) \triangleq & \\
\quad \textbf{stkcase } e_1 \ (\rho \Rightarrow e_2, _ \Rightarrow \textbf{stkcase}' \ e_1 \ (\overline{\rho} \Rightarrow \overline{e}, _ \Rightarrow e_2)) &
\end{array}$$

(where y fresh)

Multi-arm **typecase** abbreviation **typecase**$[\alpha.\tau_1']$ α $(\overline{\tau} \Rightarrow \overline{e}, \alpha \Rightarrow e')$

$$\begin{array}{ll}
\textbf{typecase}[\alpha.\tau'] \ \alpha \ (\tau \Rightarrow e_1, \alpha \Rightarrow e_2) & \triangleq \textbf{typecase}[\alpha.\tau'] \ \alpha \ (\tau \Rightarrow e_1, \alpha \Rightarrow e_2) \\
\textbf{typecase}[\alpha.\tau'] \ \alpha \ (\tau \Rightarrow e_1, \overline{\tau} \Rightarrow \overline{e}, \alpha \Rightarrow e_2) \triangleq & \\
\quad \textbf{typecase}[\alpha.\tau'] \ \alpha \ (\tau \Rightarrow e_1, \alpha \Rightarrow \textbf{typecase}[\alpha.\tau'] \ \alpha \ (\overline{\tau} \Rightarrow \overline{e}, \alpha \Rightarrow e_2)) &
\end{array}$$

Pointcut splitting helper $\pi(tm, e)$

$$\begin{array}{rcl}
\pi(\texttt{around}, e) & = & \textbf{let } \langle x, _ \rangle = e \textbf{ in } x \\
\pi(\texttt{stk}, e) & = & \textbf{let } \langle _, x \rangle = e \textbf{ in } x
\end{array}$$

(where x fresh)

Figure 2.13: Translation Abbreviations

Polytype translation $\Delta \vdash s \overset{type}{\Longrightarrow} \tau$

$$\frac{\Delta, \overline{a} \vdash t \overset{type}{\Longrightarrow} \tau'}{\Delta \vdash \texttt{<}\overline{a}\texttt{>}\ t \overset{type}{\Longrightarrow} \forall \overline{\alpha}.\tau'}\ \textsf{tpy:all}$$

Monotype translation $\Delta \vdash t \overset{type}{\Longrightarrow} \tau$

$$\frac{a \in \Delta}{\Delta \vdash a \overset{type}{\Longrightarrow} \alpha}\ \textsf{ttp:var} \qquad \frac{}{\Delta \vdash X \overset{type}{\Longrightarrow} 1}\ \textsf{ttp:unif} \qquad \frac{}{\Delta \vdash \texttt{Unit} \overset{type}{\Longrightarrow} 1}\ \textsf{ttp:unit}$$

$$\frac{}{\Delta \vdash \texttt{String} \overset{type}{\Longrightarrow} \textsf{string}}\ \textsf{ttp:str} \qquad \frac{}{\Delta \vdash \texttt{Stack} \overset{type}{\Longrightarrow} \textsf{stack}}\ \textsf{ttp:stk}$$

$$\frac{\Delta \vdash t_1 \overset{type}{\Longrightarrow} \tau_1' \qquad \Delta \vdash t_2 \overset{type}{\Longrightarrow} \tau_1'}{\Delta \vdash t_1\ \texttt{->}\ t_2 \overset{type}{\Longrightarrow} \tau_1' \to \tau_2'}\ \textsf{ttp:fun}$$

$$\frac{\Delta, \overline{a} \vdash t_1 \overset{type}{\Longrightarrow} \tau_1' \qquad \Delta, \overline{a} \vdash t_2 \overset{type}{\Longrightarrow} \tau_2'}{\begin{array}{l}\Delta \vdash \texttt{pc (<}\overline{a}\texttt{>}\ t_1\ \texttt{~>}\ t_2\texttt{)} \overset{type}{\Longrightarrow} \\ \quad (\overline{\alpha}.(\tau_1' \times \textsf{stack} \times \textsf{string}) \to (\tau_2' \times \textsf{stack} \times \textsf{string}))\ \textsf{pc} \times (\overline{\alpha}.\tau_1' \times \textsf{string})\ \textsf{pc}\end{array}}\ \textsf{ttp:pc}$$

Figure 2.14: Type translation from AspectML to $\mathbb{F}_A$

Type variable context translation $\Delta \Longrightarrow \Delta'$

$\Delta \Longrightarrow \Delta'$ iff for all $a \in \Delta$, $\alpha \in \Delta'$.

Term variable context translation $\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma'$

$$\frac{}{\begin{array}{l}\Delta; \Phi \vdash \cdot \Longrightarrow \\ \quad \mathcal{U}_{\texttt{around}}; (\alpha\beta.(\alpha \times \mathsf{stack} \times \mathsf{string}) \to (\beta \times \mathsf{stack} \times \mathsf{string}))\ \mathsf{label}, \\ \quad \mathcal{U}_{\texttt{stk}}; (\alpha.\alpha \times \mathsf{string})\ \mathsf{label}\end{array}}\ \textsf{tctx:empty}$$

$$\frac{\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma' \qquad \Delta \vdash s \stackrel{type}{\Longrightarrow} \tau}{\Delta; \Phi \vdash \Gamma, x :: s \Longrightarrow \Gamma', x; \tau}\ \textsf{tctx:lc}$$

$$\frac{\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma' \qquad \Delta \vdash s \stackrel{type}{\Longrightarrow} \tau}{\Delta; \Phi \vdash \Gamma, x : s \Longrightarrow \Gamma', x; \tau}\ \textsf{tctx:gc}$$

$$\frac{\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma' \qquad f \in \Phi \qquad \Delta \vdash s \stackrel{type}{\Longrightarrow} \forall\overline{\alpha}.\tau_1 \to \tau_2}{\begin{array}{l}\Delta; \Phi \vdash \Gamma, f :: s \Longrightarrow \\ \quad \Gamma', f_{\texttt{around}}; (\overline{\alpha}.(\tau_1 \times \mathsf{stack} \times \mathsf{string}) \to (\tau_2 \times \mathsf{stack} \times \mathsf{string}))\ \mathsf{label}, \\ \quad\quad f_{\texttt{stk}}; (\overline{\alpha}.\tau_1 \times \mathsf{string})\ \mathsf{label}, \\ \quad\quad f; \forall\overline{\alpha}.\tau_1 \to \tau_2\end{array}}\ \textsf{tctx:lc-fun}$$

$$\frac{\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma' \qquad f \in \Phi \qquad \Delta \vdash s \stackrel{type}{\Longrightarrow} \forall\overline{\alpha}.\tau_1 \to \tau_2}{\begin{array}{l}\Delta; \Phi \vdash \Gamma, f : s \Longrightarrow \\ \quad \Gamma', f_{\texttt{around}}; (\overline{\alpha}.(\tau_1 \times \mathsf{stack} \times \mathsf{string}) \to (\tau_2 \times \mathsf{stack} \times \mathsf{string}))\ \mathsf{label}, \\ \quad\quad f_{\texttt{stk}}; (\overline{\alpha}.\tau_1 \times \mathsf{string})\ \mathsf{label}, \\ \quad\quad f; \forall\overline{\alpha}.\tau_1 \to \tau_2\end{array}}\ \textsf{tctx:gc-fun}$$

Figure 2.15: Context translation from AspectML to $\mathbb{F}_A$

Programs $e : t \stackrel{prog}{\Longrightarrow} e'$

$$\frac{\cdot;\cdot;\cdot \vdash e : t \stackrel{term}{\Longrightarrow} e'}{e : t \stackrel{prog}{\Longrightarrow}} \text{ tprog}$$

$$\begin{array}{l}\textbf{let } \mathcal{U}_{\texttt{around}} : (\alpha\beta.(\alpha \times \mathsf{stack} \times \mathsf{string}) \to (\beta \times \mathsf{stack} \times \mathsf{string}))\ \mathsf{label} = \\ \quad \textbf{new } (\alpha\beta.(\alpha \times \mathsf{stack} \times \mathsf{string}) \to (\beta \times \mathsf{stack} \times \mathsf{string})) \leq \mathcal{U} \textbf{ in} \\ \textbf{let } \mathcal{U}_{\texttt{stk}} : (\alpha.\alpha \times \mathsf{string})\ \mathsf{label} = \\ \quad \textbf{new } (\alpha.\alpha \times \mathsf{string}) \leq \mathcal{U} \textbf{ in } e'\end{array}$$

Figure 2.16: Program Translation from AspectML to $\mathbb{F}_A$

The essence of the translation is that join points must be made explicit in $\mathbb{F}_A$. Therefore, we translate functions so that that they include an explicitly labeled join point surrounding the function body and another that stores information on the stack as the function executes. More specifically, for each function we create two labels $f_{\texttt{around}}$, and $f_{\texttt{stk}}$ for these join points. So that AspectML programs can refer to the pointcut designators of any function using the `any` keyword , all labels $f_{\texttt{around}}$ are derived from a distinguished label $\mathcal{U}_{\texttt{around}}$. Likewise, $\mathcal{U}_{\texttt{stk}}$ is the parent of all $f_{\texttt{stk}}$. These constructions can be seen in Figure 2.16, where in Rule tctx:empty, the $\mathcal{U}_{\texttt{around}}$ and $\mathcal{U}_{\texttt{stk}}$ labels are created in the $\mathbb{F}_A$ context. Similarly, in Rules tctx:lc-fun and tctx:gc-fun, $f_{\texttt{around}}$ and $f_{\texttt{stk}}$ labels are created in the $\mathbb{F}_A$ term variable context for each f in the AspectML context. Finally, the actual $\mathcal{U}_{\texttt{around}}$ and $\mathcal{U}_{\texttt{stk}}$ labels are created in Rule tprog directly underneath $\mathcal{U}$ in the $\mathbb{F}_A$ label hierarchy.

Pointcuts are translated into a tuple of two $\mathbb{F}_A$ pointcuts in Figure 2.17. The pointcut `any` becomes a tuple containing the $\mathcal{U}_{\texttt{around}}$ and $\mathcal{U}_{\texttt{stk}}$ pointcuts, which, as explained previously, contain the parents of all `around` and `stk` labels respectively. Sets of functions are translated into tuples of pointcuts containing their associated `before`, `after`, and `stk` labels. We then use a pointcut splitting helper function to

Local term translation $\Delta; \Phi; \Gamma \vdash^{\text{loc}} e : t \xRightarrow{term} e'$

$$\frac{\Delta \vdash t \qquad \Delta; \Phi; \Gamma \vdash e : t \xRightarrow{term} e'}{\Delta; \Phi; \Gamma \vdash^{\text{loc}} e\texttt{:}t : t \xRightarrow{term} e'}\ \textsf{lttm:cnv} \qquad \frac{x :: t \in \Gamma}{\Delta; \Phi; \Gamma \vdash^{\text{loc}} x : t \xRightarrow{term} x}\ \textsf{lttm:var}$$

$$\frac{}{\Delta; \Phi; \Gamma \vdash^{\text{loc}} c : \texttt{String} \xRightarrow{term} c}\ \textsf{lttm:string}$$

$$\frac{}{\Delta; \Phi; \Gamma \vdash^{\text{loc}} \texttt{()} : \texttt{Unit} \xRightarrow{term} ()}\ \textsf{lttm:unit}$$

$$\frac{\Gamma(f_i) = \texttt{<}\overline{a}\texttt{>}\ t_{1,i}\ \texttt{->}\ t_{2,i} \qquad \begin{array}{c}\forall i \qquad f_i \in \Phi \\ \Delta \vdash \texttt{<}\overline{b}\texttt{>}\ t_1\ \texttt{->}\ t_2 \preceq \texttt{<}\overline{a}\texttt{>}\ t_{1,i}\ \texttt{->}\ t_{2,i}\end{array}}{\begin{array}{l}\Delta; \Phi; \Gamma \vdash^{\text{loc}} \texttt{\#}\overline{f}\texttt{:}\ (\texttt{<}\overline{b}\texttt{>}\ t_1\ \texttt{~>}\ t_2)\texttt{\#} : \texttt{pc}\ (\texttt{<}\overline{b}\texttt{>}\ t_1\ \texttt{~>}\ t_2) \xRightarrow{term} \\ \quad \langle\{\overline{f_{\texttt{around}}}\}, \{\overline{f_{\texttt{stk}}}\}\rangle\end{array}}\ \textsf{lttm:set-ann}$$

$$\frac{}{\Delta; \Phi; \Gamma \vdash^{\text{loc}} \texttt{any} : \texttt{pc}\ (\texttt{<}ab\texttt{>}\ a\ \texttt{~>}\ b) \xRightarrow{term} \langle\{\mathcal{U}_{\texttt{around}}\}, \{\mathcal{U}_{\texttt{stk}}\}\rangle}\ \textsf{lttm:any}$$

$$\frac{\Delta; \Phi; \Gamma \vdash ds\,; e : t \xRightarrow{decs} e'}{\Delta; \Phi; \Gamma \vdash^{\text{loc}} \texttt{let}\ ds\ \texttt{in}\ e : t \xRightarrow{term} e'}\ \textsf{lttm:let}$$

Figure 2.17: Local Expression Translation from AspectML to $\mathbb{F}_A$

Declarations $\Delta; \Phi; \Gamma \vdash d\,; e : t \stackrel{decs}{\Longrightarrow} e'$

$$\frac{\begin{array}{c}\Delta, \overline{a} \vdash t_1 \texttt{ -> } t_2 \stackrel{type}{\Longrightarrow} \tau_1' \to \tau_2' \\ \Delta, \overline{a}; \Phi; \Gamma, f :: t_1 \texttt{ -> } t_2, x :: t_1 \vdash e_1 : t_2 \stackrel{term}{\Longrightarrow} e_1' \\ \Delta; \Phi, f; \Gamma, f :: \texttt{<}\overline{a}\texttt{>}\ t_1 \texttt{ -> } t_2 \vdash e_2 : t \stackrel{term}{\Longrightarrow} e_2'\end{array}}{\begin{array}{l}\Delta; \Phi\,; \Gamma \vdash \texttt{fun}\ \ f\ \texttt{<}\overline{a}\texttt{>}\ (x\texttt{:}\,t_1)\,\texttt{:}\,t_2\ \texttt{=}\ e_1; e_2 : t \stackrel{decs}{\Longrightarrow} \\ \quad \textbf{let}\ f_{\texttt{around}} : (\overline{\alpha}.(\tau_1' \times \mathsf{stack} \times \mathsf{string}) \to (\tau_2' \times \mathsf{stack} \times \mathsf{string}))\ \mathsf{label} = \\ \quad\quad \textbf{new}\ (\overline{\alpha}.(\tau_1' \times \mathsf{stack} \times \mathsf{string}) \to (\tau_2' \times \mathsf{stack} \times \mathsf{string})) \leq \mathcal{U}_{\texttt{around}}\ \textbf{in} \\ \quad \textbf{let}\ f_{\texttt{stk}} : (\overline{\alpha}.\tau_1' \times \mathsf{string})\ \mathsf{label} = \textbf{new}\ (\overline{\alpha}.\tau_1' \times \mathsf{string}) \leq \mathcal{U}_{\texttt{stk}}\ \textbf{in} \\ \quad \textbf{let}\ f : \forall\overline{\alpha}.\tau_1' \to \tau_2' = \Lambda\overline{\alpha}.\textbf{fix}\ f : \tau_1' \to \tau_2'.\lambda x; \tau_1'. \\ \quad\quad \textbf{store}\ f_{stk}[\overline{\alpha}][\![\langle x, \text{“f”}\rangle]\!]\ \textbf{in} \\ \quad\quad (\textbf{let}\ \langle x'', \_, \_\rangle = f_{\texttt{around}}[\overline{\alpha}][\![\lambda w : (\tau_1' \times \mathsf{stack} \times \mathsf{string}). \\ \quad\quad\quad \textbf{let}\ \langle x, y, z\rangle = w\ \textbf{in}\ \langle e_1', y, z\rangle]\!]\ (x', \textbf{stack}, \text{“f”})\ \textbf{in}\ x'') \\ \quad e_2'\end{array}}\ \textsf{tds:fun-ann}$$

$$\frac{\Delta, \overline{a} \vdash t_1 \qquad \Delta, \overline{a} \vdash t_2 \qquad \Delta; \Phi; \Gamma \vdash \texttt{fun}\ \ f\ \texttt{<}\overline{a}\texttt{>}\ (x\texttt{:}\,t_1)\,\texttt{:}\,t_2\ \texttt{=}\ e_1\,; e_2 : t \stackrel{decs}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash \texttt{fun}\ \ f\ x\ \texttt{=}\ e_1\,; e_2 : t \stackrel{decs}{\Longrightarrow} e'}\ \textsf{tds:fun}$$

Figure 2.18: Function Declaration Translation from AspectML to $\mathbb{F}_A$

pick either the first or second element of the pointcut tuple depending on whether we attempting to use the pointcut in advice or in a stack pattern.

The translation of functions (Rule tds:fun-ann) in Figure 2.18 begins by creating the labels, $f_{\texttt{around}}$ and $f_{\texttt{stk}}$ for the function's join points. Inside the body of the translated function, a **store** statement marks the function's stack frame with the $f_{\texttt{stk}}$ label. The function's body is η-expanded and passed to the join point to be used as the proceed function f by any advice triggered by the $f_{\texttt{around}}$ label. Because AspectML advice expects the current stack and a string of the function name, we also insert **stack**s and string constants into the join points.

Declarations $\Delta; \Phi; \Gamma \vdash d \,;\, e : t \stackrel{decs}{\Longrightarrow} e'$

$$\frac{tm \in \{\texttt{before}, \texttt{after}\} \qquad \Delta, \overline{a} \vdash t_1 \qquad \Delta; \Phi; \Gamma \vdash \texttt{advice}\ tm\ \texttt{(|}e_1\texttt{|) <}\overline{a}\texttt{> (}x\texttt{:}t_1\texttt{,}y\texttt{,}z\texttt{) = }e_2 \,;\, e_3 : t_2 \stackrel{decs}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash \texttt{advice}\ tm\ \texttt{(|}e_1\texttt{|) (}x\texttt{,}y\texttt{,}z\texttt{) = }e_2 \,;\, e_3 : t_2 \stackrel{decs}{\Longrightarrow} e'}\ \text{tds:adv}$$

$$\frac{\Delta, \overline{a} \vdash t_2 \qquad \Delta; \Phi; \Gamma \vdash \texttt{advice around (|}e_1\texttt{|) <}\overline{a}\texttt{> (}x\texttt{:}t_1\texttt{,}y\texttt{,}z\texttt{) :}t_2\texttt{ = } \textit{proceed}\ e_2 \,;\, e_3 : t_3 \stackrel{decs}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash \texttt{advice before (|}e_1\texttt{|) <}\overline{a}\texttt{> (}x\texttt{:}t_1\texttt{,}y\texttt{,}z\texttt{) = } e_2 \,;\, e_3 : t_3 \stackrel{decs}{\Longrightarrow} e'}\ \text{tds:adv-bef}$$

$$\frac{\Delta, \overline{a} \vdash t_2 \qquad \Delta; \Phi; \Gamma \vdash \texttt{advice around (|}e_1\texttt{|) <}\overline{a}\texttt{> (}x\texttt{:}t_1\texttt{,}y\texttt{,}z\texttt{) :}t_2\texttt{ = } (\textbf{let}\ x : t_2 = (\textit{proceed}\ x)\ \textbf{in}\ e_2) \,;\, e_3 : t_3 \stackrel{decs}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash \texttt{advice after (|}e_1\texttt{|) <}\overline{a}\texttt{> (}x\texttt{:}t_2\texttt{,}y\texttt{,}z\texttt{) = }e_2 \,;\, e_3 : t_3 \stackrel{decs}{\Longrightarrow} e'}\ \text{tds:adv-aft}$$

$$\frac{\Delta, \overline{a} \vdash t_1 \qquad \Delta, \overline{a} \vdash t_2 \qquad \Delta; \Phi; \Gamma \vdash \texttt{advice around (|}e_1\texttt{|) <}\overline{a}\texttt{> (}x\texttt{:}t_1\texttt{,}y\texttt{,}z\texttt{) :}t_2\texttt{ = } e_2 \,;\, e_3 : t_3 \stackrel{decs}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash \texttt{advice around (|}e_1\texttt{|) (}x\texttt{,}y\texttt{,}z\texttt{) = } e_2 \,;\, e_3 : t_3 \stackrel{decs}{\Longrightarrow} e'}\ \text{tds:adv-aro}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash^{\text{loc}} e_1 : \texttt{pc}\ pt \stackrel{term}{\Longrightarrow} e_1' \qquad \pi(\texttt{around}, pt) = \texttt{<}\overline{a}\texttt{>}\ t_1 \to t_2 \\ \pi(\texttt{around}, e_1') = e_1'' \qquad \Delta, \overline{a} \vdash t_1 \to t_2 \stackrel{type}{\Longrightarrow} \tau_1' \to \tau_2' \\ \Delta, \overline{a}; \Phi; \Gamma, x; t_1, y; \texttt{Stack}, z; \texttt{String}, \textit{proceed}; t_1 \to t_2 \vdash \\ e_2 : t_2 \stackrel{term}{\Longrightarrow} e_2' \\ \Delta; \Phi; \Gamma \vdash e_3 : t_3 \stackrel{term}{\Longrightarrow} e_3'\end{array}}{\begin{array}{l}\Delta; \Phi \,; \Gamma \vdash \texttt{advice around (|}e_1\texttt{|) <}\overline{a}\texttt{> (}x\texttt{:}t_1\texttt{,}y\texttt{,}z\texttt{) :}t_2\texttt{ = }e_2\texttt{;}\, e_3 : t_3 \stackrel{decs}{\Longrightarrow} \\ \quad \textbf{let}\ \_ : 1 =\Uparrow \{e_1''.\overline{\alpha}(x; (\tau_1' \times \mathsf{stack} \times \mathsf{string}), \\ \qquad f; (\tau_1' \times \mathsf{stack} \times \mathsf{string}) \to (\tau_2' \times \mathsf{stack} \times \mathsf{string})) \to \\ \qquad\quad \textbf{let}\ \langle x, y, z\rangle = x\ \textbf{in} \\ \qquad\quad \langle e_2'[\lambda x : \tau_1'.\textbf{let}\ \langle x, \_, \_\rangle = f\ \langle x, y, z\rangle\ \textbf{in}\ x/\textit{proceed}], y, z\rangle\} \\ \quad \textbf{in}\ e_3'\end{array}}\ \text{tds:adv-ann}$$

Figure 2.19: Advice Declaration Translation from AspectML to $\mathbb{F}_A$

$$\boxed{\text{Declarations } \Delta;\Phi;\Gamma \vdash d\,;e : t \overset{decs}{\Longrightarrow} e'}$$

$$\frac{\begin{array}{l}\Delta;\Phi;\Gamma \vdash \texttt{case-advice around (|}e_1\texttt{|) (}x\texttt{:}t_1\texttt{,}y\texttt{,}z\texttt{):}a\texttt{ =}\\ \quad \mathit{proceed}\ e_2\,;e_3 : t_2 \overset{decs}{\Longrightarrow} e'\end{array}}{\begin{array}{l}\Delta;\Phi;\Gamma \vdash \texttt{case-advice before (|}e_1\texttt{|) (}x\texttt{:}t_1\texttt{,}y\texttt{,}z\texttt{) = }e_2\,;\\ \quad e_3 : t_2 \overset{decs}{\Longrightarrow} e'\end{array}}\ \text{tds:cadv-bef}$$

$$\frac{\begin{array}{l}\Delta;\Phi;\Gamma \vdash \texttt{case-advice around (|}e_1\texttt{|) (}x\texttt{:}a\texttt{,}y\texttt{,}z\texttt{):}t_1\texttt{ =}\\ \quad (\textbf{let}\ x : t_1 = (\mathit{proceed}\ x)\ \textbf{in}\ e_2)\,;e_3 : t_2 \overset{decs}{\Longrightarrow} e'\end{array}}{\begin{array}{l}\Delta;\Phi;\Gamma \vdash \texttt{case-advice after (|}e_1\texttt{|) (}x\texttt{:}t_1\texttt{,}y\texttt{,}z\texttt{) = }e_2\,;\\ \quad e_3 : t_2 \overset{decs}{\Longrightarrow} e'\end{array}}\ \text{tds:cadv-aft}$$

$$\frac{\begin{array}{c}\Delta;\Phi;\Gamma \overset{\text{loc}}{\vdash} e_1 : \texttt{pc}\ pt \overset{term}{\Longrightarrow} e_1' \qquad \pi(\texttt{around}, pt) = \texttt{<}\overline{a}\texttt{>}\ t_1 \to t_2\\ \pi(\texttt{around}, e_1') = e_1'' \qquad \overline{b} = \mathrm{FTV}(t_3) \cup \mathrm{FTV}(t_4) - \Delta\\ \Delta, \overline{a} \vdash t_1 \to t_2 \overset{type}{\Longrightarrow} \tau_1' \to \tau_2' \qquad \Delta, \overline{b} \vdash t_3 \to t_4 \overset{type}{\Longrightarrow} \tau_3' \to \tau_4'\\ \Delta, \overline{b}; \Phi; \Gamma, x; t_3, y; \mathsf{Stack}, z; \mathsf{String}, \mathit{proceed}; t_3 \to t_4 \vdash e_2 : t_4 \overset{term}{\Longrightarrow} e_2'\\ \Delta;\Phi;\Gamma \vdash e_3 : t \overset{term}{\Longrightarrow} e_3'\end{array}}{\begin{array}{l}\Delta;\Phi\,;\Gamma \vdash \texttt{case-advice around (|}e_1\texttt{|) (}x\texttt{:}t_3\texttt{,}y\texttt{,}z\texttt{):}t_4\texttt{ = }e_2;\\ \quad e_3 : t \overset{decs}{\Longrightarrow}\\ \qquad \textbf{let}\ \_ : 1 = \Uparrow \{e_1''.\overline{\alpha}x; (\tau_1' \times \mathsf{stack} \times \mathsf{string}),\\ \qquad\quad f; (\tau_1' \times \mathsf{stack} \times \mathsf{string}) \to (\tau_2' \times \mathsf{stack} \times \mathsf{string}) \to\\ \qquad\qquad \textbf{let}\ \langle x, y, z\rangle = x\ \textbf{in}\\ \qquad\qquad \langle \textbf{typecase}[\alpha.\alpha]\ \tau_1' \to \tau_2'\\ \qquad\qquad\quad (\tau_3' \to \tau_4' \Rightarrow\\ \qquad\qquad\qquad e_2'[\lambda x : \tau_1'.\textbf{let}\ \langle x, \_, \_\rangle = f\ \langle x, y, z\rangle\ \textbf{in}\ x/\mathit{proceed}],\\ \qquad\qquad\quad \alpha \Rightarrow x), y, z\rangle\}\\ \qquad \textbf{in}\ e_3'\end{array}}\ \text{tds:cadv-aro}$$

Figure 2.20: Case-advice Declaration Translation from AspectML to $\mathbb{F}_A$

Advice and case-advice translation is defined in Figures 2.19 and 2.20. The most significant difference between advice in AspectML and $\mathbb{F}_A$ is that $\mathbb{F}_A$ has no notion of a trigger time. Because the pointcut argument of the advice will translate into a tuple of two $\mathbb{F}_A$ pointcuts, *tm* is used to determine which component is used. The translation also splits the input of the advice into the two arguments that AspectML expects and immediately installs the advice.

To simplify the translation, only the rules for `around` advice (tds:adv-ann and tds:cadv-aro) are directly defined – `before` advice (Rules tds:adv-bef and tds:cadv-bef) becomes `around` advice such that the `before` advice is executed, then `proceed` is called on the result. Similarly, `after` advice (Rules tds:adv-aft and tds:cadv-aft) becomes `around` advice such that the `proceed` function executes the function body, and then the `after` advice is run on the result. Finally, the `around case-advice` declaration (Rule tds:cadv-aro) uses a **typecase** expression to perform the necessary type analysis.

Finally, we include the global expression translation rules (Figure 2.21) and the pattern translation rules (Figure 2.22). In Rule tpat:cons-ann, the second, stack element of the pointcut tuple is selected by the π function as the pointcut to be used by the **stkcase** expression.

2.5.2 Translation Type Safety

We prove that the translation is type-preserving, and therefore that AspectML is also type safe.

Pattern splitting helper $split(\Xi, e)$

$$\begin{array}{rcl} split(\cdot, e) & = & e \\ split(\Xi, x \mapsto (y, z), e) & = & split(\Xi, \textbf{let } \langle y, z\rangle = x \textbf{ in } e) \end{array}$$

Global term translation $\Delta; \Phi; \Gamma \vdash e : t \stackrel{term}{\Longrightarrow} e'$

$$\frac{\Delta; \Phi; \Gamma \vdash^{\text{loc}} e : t \stackrel{term}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash e : t \stackrel{term}{\Longrightarrow} e'}\ \textsf{gttm:cnv} \qquad \frac{\Gamma(x) = \texttt{<}\overline{a}\texttt{>}\ t \qquad \Delta \vdash t_i \stackrel{type}{\Longrightarrow} \tau_i'}{\Delta; \Phi; \Gamma \vdash x : t[\overline{t}/\overline{a}] \stackrel{term}{\Longrightarrow} x[\overline{\tau'}]}\ \textsf{gttm:var}$$

$$\frac{\Delta; \Phi; \Gamma \vdash e_1 : t_1 \texttt{ -> } t_2 \stackrel{term}{\Longrightarrow} e_1' \qquad \Delta; \Phi; \Gamma \vdash e_2 : t_1 \stackrel{term}{\Longrightarrow} e_2'}{\Delta; \Phi; \Gamma \vdash e_1 e_2 : t_2 \stackrel{term}{\Longrightarrow} e_1' e_2'}\ \textsf{gttm:app}$$

$$\frac{\begin{array}{c} \Delta; \Phi; \Gamma \vdash e : \texttt{Stack} \stackrel{term}{\Longrightarrow} e_t \\ \left(\begin{array}{ll} \forall i & \Delta; \Phi; \Gamma \vdash pat_i \stackrel{pat}{\Longrightarrow} \rho_i' \dashv \Delta_i; \Gamma_i; \Xi_i \\ & \Delta, \Delta_i; \Phi; \Gamma, \Gamma_i \vdash e_i : t \stackrel{term}{\Longrightarrow} e_i' \\ & \Delta; \Phi; \Gamma \vdash e' : t \stackrel{term}{\Longrightarrow} e_t' \end{array}\right) \end{array}}{\begin{array}{l} \Delta; \Phi; \Gamma \vdash \texttt{stkcase } e\ (\overline{pat\texttt{=>}e}\ \texttt{|\_=>}\ e') : t \stackrel{term}{\Longrightarrow} \\ \quad \textbf{stkcase } e_t\ (\overline{\rho'} \Rightarrow \overline{split(\Xi, e')}, x \Rightarrow e_t') \end{array}}\ \textsf{gttm:scase}$$

$$\frac{\begin{array}{c} \Delta, a \vdash t \stackrel{type}{\Longrightarrow} \tau' \qquad \Delta; \Phi; \Gamma \vdash e : t \stackrel{term}{\Longrightarrow} e' \\ \left(\begin{array}{lll} \forall i & \Delta_i = \mathrm{FTV}(t_i) - \Delta & \Delta, \Delta_i \vdash t_i \stackrel{type}{\Longrightarrow} \tau_i' \quad a \notin \mathrm{FTV}(t_i) \\ & \Delta, \Delta_i; \Phi; \Gamma\langle t_i/a\rangle \vdash e_i[t_i/a] : t[t_i/a] \stackrel{term}{\Longrightarrow} e_i' & \end{array}\right) \end{array}}{\begin{array}{l} \Delta; \Phi; \Gamma \vdash \texttt{typecase<}t\texttt{>}\ a\ (\overline{t\texttt{=>}e}\ \texttt{|\_=>}\ e) : t \stackrel{term}{\Longrightarrow} \\ \quad \textbf{typecase}[\alpha.\tau']\ \alpha\ (\overline{\tau'} \Rightarrow \overline{e'}, \alpha \Rightarrow e') \end{array}}\ \textsf{gttm:tcase}$$

$$\frac{\Delta; \Phi; \Gamma \vdash ds; e : t \stackrel{decs}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash \texttt{let } ds \texttt{ in } e : t \stackrel{term}{\Longrightarrow} e'}\ \textsf{gttm:let}$$

Figure 2.21: Global Expression Translation from AspectML to $\mathbb{F}_A$

Splitting context translation $\Delta; \Gamma \vdash \Xi \Longrightarrow \Gamma'$

$$\frac{}{\Delta; \cdot \vdash \cdot \Longrightarrow \cdot}\ \textsf{tsctx:empty} \qquad \frac{\Delta; \Gamma \vdash \Xi \Longrightarrow \Gamma'}{\Delta; \Gamma, x; \mathtt{Stack} \vdash \Xi \Longrightarrow \Gamma', x; \mathsf{stack}}\ \textsf{tsctx:cons1}$$

$$\frac{\Delta; \Gamma \vdash \cdot \Longrightarrow \Gamma'}{\Delta; \Gamma, x; t \vdash \cdot \Longrightarrow \Gamma'}\ \textsf{tsctx:cons2}$$

$$\frac{\Delta; \Gamma \vdash \Xi \Longrightarrow \Gamma' \qquad \Delta \vdash t \overset{type}{\Longrightarrow} \tau}{\Delta; \Gamma, y; t, z; \mathtt{String} \vdash \Xi, x \mapsto (y, z) \Longrightarrow \Gamma', x; \tau \times \mathsf{string},}\ \textsf{tsctx:cons3}$$

Patterns $\Delta; \Phi; \Gamma \vdash pat \overset{pat}{\Longrightarrow} \rho \dashv \Delta'; \Gamma'; \Xi$

$$\frac{}{\Delta; \Phi; \Gamma \vdash \mathtt{[]} \overset{pat}{\Longrightarrow} \bullet \dashv \cdot; \cdot; \cdot}\ \textsf{tpat:nil} \qquad \frac{}{\Delta; \Phi; \Gamma \vdash x \overset{pat}{\Longrightarrow} x \dashv \cdot; \cdot, x; \mathtt{Stack}; \cdot}\ \textsf{tpat:var}$$

$$\frac{\Delta; \Phi; \Gamma \vdash pat \overset{pat}{\Longrightarrow} \rho' \dashv \Delta'; \Gamma'; \Xi}{\Delta; \Phi; \Gamma \vdash \mathtt{\_::}pat \overset{pat}{\Longrightarrow} \mathtt{\_::}\rho' \dashv \Delta'; \Gamma'; \Xi}\ \textsf{tpat:wild}$$

$$\frac{\Delta, \overline{a} \vdash t \qquad \Delta; \Phi; \Gamma \vdash \mathtt{(|}e\mathtt{|)<}\overline{a}\mathtt{>(}x\mathtt{:}t\mathtt{,}z\mathtt{)::}pat \overset{pat}{\Longrightarrow} \rho' \dashv \Delta'; \Gamma'; \Xi}{\Delta; \Phi; \Gamma \vdash \mathtt{(|}e\mathtt{|)(}x\mathtt{,}z\mathtt{)::}pat \overset{pat}{\Longrightarrow} \rho' \dashv \Delta'; \Gamma'; \Xi}\ \textsf{tpat:cons}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} e : \mathtt{pc\ (<}\overline{a}\mathtt{>}\ t_1\ \mathtt{\tilde{}>}\ t_2\mathtt{)} \overset{term}{\Longrightarrow} e' \\ \pi(\mathtt{stk}, e') = e'' \qquad \Delta; \Phi; \Gamma \vdash pat \overset{pat}{\Longrightarrow} \rho' \dashv \Delta'; \Gamma'; \Xi \qquad y \text{ fresh}\end{array}}{\begin{array}{l}\Delta; \Phi; \Gamma \vdash \mathtt{(|}e\mathtt{|)<}\overline{a}\mathtt{>(}x\mathtt{:}t_1\mathtt{,}z\mathtt{)::}pat \overset{pat}{\Longrightarrow} \\ \quad e''[\overline{\alpha}][\![y]\!]::\rho' \dashv \Delta', \overline{a}; \Gamma', x; t_1, z; \mathtt{String}; \Xi, y \mapsto (x, z)\end{array}}\ \textsf{tpat:cons-ann}$$

Figure 2.22: Pattern Translation from AspectML to $\mathbb{F}_A$

Lemma 2.5.1 (Translation commutes with type substitution) *Given* $\Delta \Longrightarrow \Delta'$ *and* $\Delta \vdash t' \stackrel{type}{\Longrightarrow} \tau'$

then

1. *If* $\Delta, a \vdash s \stackrel{type}{\Longrightarrow} \tau$ *then* $\Delta \vdash s[t'/a] \stackrel{type}{\Longrightarrow} \tau[\tau'/\alpha]$.

2. *If* $\Delta, a \vdash t \stackrel{type}{\Longrightarrow} \tau$ *then* $\Delta \vdash t[t'/a] \stackrel{type}{\Longrightarrow} \tau[\tau'/\alpha]$.

3. *If* $\Delta, a; \Phi; \Gamma \vdash e : t \stackrel{term}{\Longrightarrow} e'$ *and* $e[t'/a]$ *is defined then* $\Delta; \Phi; \Gamma[t'/a] \vdash e[t'/a] : t \stackrel{term}{\Longrightarrow} e'[\tau'/\alpha]$.

4. *If* $\Delta, a; \Phi; \Gamma \stackrel{\vdash^{\text{loc}}}{} e : t \stackrel{term}{\Longrightarrow} e'$ *and* $e[t'/a]$ *is defined then* $\Delta; \Phi; \Gamma[t'/a] \stackrel{\vdash^{\text{loc}}}{} e[t'/a] : t[t'/a] \stackrel{term}{\Longrightarrow} e'[\tau'/\alpha]$.

5. *If* $\Delta, a; \Phi; \Gamma \vdash ds\,; e : t \stackrel{decs}{\Longrightarrow} e'$ *and both* $ds[t'/a]$ *and* $e[t'/a]$ *are defined then* $\Delta; \Phi; \Gamma[t'/a] \vdash ds[t'/a]\,; e[t'/a] : t[t'/a] \stackrel{decs}{\Longrightarrow} e'[\tau'/\alpha]$.

6. *If* $\Delta, a; \Phi; \Gamma \vdash pat \stackrel{pat}{\Longrightarrow} \rho \dashv \Delta''; \Gamma''; \Xi$ *and* $pat[t'/a]$ *is defined then* $\Delta; \Phi; \Gamma[t'/a] \vdash pat[t'/a] \stackrel{pat}{\Longrightarrow} \rho[\tau'/\alpha] \dashv \Delta''; \Gamma''[\tau'/\alpha]; \Xi$

Proof: By induction on derivations. □

Lemma 2.5.2 (Pointcut splitting commutes with type substitution)

1. $\pi(tm, pt[\tau/\alpha]) = (\pi(tm, pt))[\tau/\alpha]$.

2. $\pi(tm, e[\tau/\alpha]) = (\pi(tm, e))[\tau/\alpha]$.

Proof: Trivial case analysis. □

Lemma 2.5.3 (Split commutes with type substitution)

$split(\Xi, e[\tau/\alpha]) = (split(\Xi, e))[\tau/\alpha]$.

Proof: Trivial induction. □

Lemma 2.5.4 (Binding type variables preserved under substitution)
If $\Delta \vdash t'$ *and* $\Delta' = \bigcup \overline{\mathrm{FTV}(t)} - (\Delta, \mathrm{a})$
then $\Delta' = \bigcup \overline{\mathrm{FTV}(t[t'/\mathrm{a}])} - \Delta$.

Proof: Trivial. □

Lemma 2.5.5 (Instance equivalence) $\Delta' \vdash \overline{\alpha}.\tau_1 \preceq \overline{\beta}.\tau_2$ *iff* $\Delta' \vdash \overline{\alpha}.(\tau_1 \otimes \tau_3) \preceq \overline{\beta}.(\tau_2 \otimes \tau_3)$ *for any type constructor* $\otimes$ *and* $\Delta' \vdash \tau_3$

Proof: Straightforward. □

Lemma 2.5.6 (Splitting lemma) *If* $\Delta \Longrightarrow \Delta'$ *and* $\Delta; \Phi \vdash \Gamma_1 \Longrightarrow \Gamma_1'$ *and* $\Delta; \Phi \vdash \Gamma_2 \Longrightarrow \Gamma_2'$ *and* $\Delta; \Gamma_2 \vdash \Xi \Longrightarrow \Gamma_3'$ *and* $\Delta'; \Gamma_1', \Gamma_2' \vdash e : \tau$
then $\Delta'; \Gamma_1', \Gamma_3' \vdash \mathit{split}(\Xi, e) : \tau$.

Proof: Straightforward induction over the structure of $\Delta; \Gamma_2 \vdash \Xi \Longrightarrow \Gamma_3'$. □

Lemma 2.5.7 (Context substitution lemma) *If* $\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma'$ *and* $\Delta \vdash t \overset{type}{\Longrightarrow} \tau$
then $\Delta; \Phi \vdash \Gamma\langle t/\mathrm{a}\rangle \Longrightarrow \Gamma''$ *and* $\Gamma''[\tau/\alpha] = \Gamma'[\tau/\alpha]$.

Proof: By induction on the context translation. □

Lemma 2.5.8 (Type translation uniqueness)

1. *If* $\Delta \vdash s \overset{type}{\Longrightarrow} \tau$ *and* $\Delta \vdash s \overset{type}{\Longrightarrow} \tau'$ *then* $\tau = \tau'$.
2. *If* $\Delta \vdash t \overset{type}{\Longrightarrow} \tau$ *and* $\Delta \vdash t \overset{type}{\Longrightarrow} \tau'$ *then* $\tau = \tau'$.

Proof: By induction over the structure of the type. □

Lemma 2.5.9 (Translation soundness) *Given* $\Delta \Longrightarrow \Delta'$ *and* $\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma'$ *then*

1. *if* $\Delta \vdash s \stackrel{type}{\Longrightarrow} \tau$ *then* $\Delta' \vdash \tau$.
2. *if* $\Delta \vdash t \stackrel{type}{\Longrightarrow} \tau$ *then* $\Delta' \vdash \tau$.
3. *If* $\Delta; \Phi; \Gamma \vdash e : t \stackrel{term}{\Longrightarrow} e'$ *then* $\Delta'; \Gamma' \vdash e' : \tau$ *where* $\Delta \vdash t \stackrel{type}{\Longrightarrow} \tau$.
4. *If* $\Delta; \Phi; \Gamma \stackrel{\text{loc}}{\vdash} e : t \stackrel{term}{\Longrightarrow} e'$ *then* $\Delta'; \Gamma' \vdash e' : \tau$ *where* $\Delta \vdash t \stackrel{type}{\Longrightarrow} \tau$.
5. *If* $\Delta; \Phi; \Gamma \vdash ds\,; e : t \stackrel{decs}{\Longrightarrow} e'$ *then* $\Delta'; \Gamma' \vdash e' : \tau$ *where* $\Delta \vdash t \stackrel{type}{\Longrightarrow} \tau$.
6. $\Delta; \Phi; \Gamma \vdash pat \stackrel{pat}{\Longrightarrow} \rho \dashv \Delta_i; \Gamma_i; \Xi$ *then* $\Delta'; \Gamma' \vdash \rho \dashv \Delta_i'; \Gamma^*$ *where* $\Delta_i \Longrightarrow \Delta_i'$ *and* $\Delta, \Delta_i; \Phi\Gamma_i \Longrightarrow \Gamma_i'$ *and* $\Delta, \Delta_i; \Gamma_i \vdash \Xi \Longrightarrow \Gamma^*$.

Proof: By induction on derivations.

- Case gttm:scase uses the Splitting Lemma 2.5.6.

□

Theorem 2.5.10 (Program translation safety) *If* $e : t \stackrel{prog}{\Longrightarrow} e'$ *then* $\cdot; \cdot \vdash e' : \tau$ *where* $\cdot \vdash t \stackrel{type}{\Longrightarrow} \tau$.

Proof: Straightforward use of the Translation Soundness for Declarations Lemma 2.5.9: Part 5. □

Theorem 2.5.11 (AspectML safety) *Suppose* $e : t \stackrel{prog}{\Longrightarrow} e'$ *then either* e' *loops safely but infinitely or there exists a sequence of reductions* $(\cdot;\cdot;e') \mapsto^* (\Sigma; A; e'')$ *to a finished configuration.*

Proof: Composition of the Program Translation Safety Theorem 2.5.10 with the $\mathbb{F}_A$ Progress Theorem 2.4.11 and Preservation Theorem 2.4.23. □ □

Chapter 3

Harmless Advice

3.1 Introduction

Many within the aspect-oriented programming community adhere to the tenet that aspects are most effective when the code they advise is *oblivious* to their presence [18]. In other words, aspects are effective when a programmer is not required to annotate the advised code (henceforth, the *mainline code*) in any particular way. When aspect-oriented languages are oblivious, all control over when advice is applied rests with the aspect programmer as opposed to the mainline programmer. This design choice simplifies after-the-fact customization or analysis of programs using aspects. For example, obliviousness makes it trivial to implement and update an extremely flexible access control infrastructure. To adjust the places where security checks occur, which may be scattered across the code base, one need only make local changes to a single aspect. Obliviousness might be one of the reasons that aspect-oriented programming has caught on with such enthusiasm in recent years,

causing major companies such as IBM and Microsoft to endorse the new paradigm and inspiring academics to create conferences and workshops to study the idea.

On the other hand, obliviousness threatens conventional modularity principles and undermines a programmer's ability to reason locally about the behavior of their code. Consequently, many traditional programming language researchers believe that aspect-oriented programs are ticking time bombs, which, if widely deployed, are bound to cause the software industry irreparable harm. One central problem is that while mainline code may be *syntactically* oblivious to aspects, it is not *semantically* oblivious to aspects. Aspects can reach inside modules, influence the behavior of local routines, and alter local data structures. As a result, to understand the semantics of code in an aspect-oriented language such as AspectJ, programmers will have to examine all external aspects that might modify local data structures or control flow. As the number and invasiveness of aspects grows, understanding and maintaining their programs may become more and more difficult.

In this work, we define a new form of *harmless* aspect-oriented advice that programmers can use to accomplish nontrivial programming tasks yet also allows them to enjoy most of the local reasoning principles they have come to depend upon for program understanding, development, and maintenance. Like ordinary aspect-oriented advice, harmless advice is a computation that executes whenever mainline control reaches a designated control-flow point. Unlike ordinary aspect-oriented advice, harmless advice is constrained to prevent it from *interfering* with the underlying computation. Therefore, harmless advice can be added to a program at any point in the development cycle without fear that important program invariants will be disrupted. In addition, programmers who develop, debug, or enhance mainline code can safely ignore harmless advice, if there is any present.

In principle, one could devise many variants of harmless advice depending upon exactly what it means to *interfere* with the underlying computation. At the most extreme end, changing the timing behavior of a program constitutes interference and consequently, only trivial advice is harmless. A slightly less extreme viewpoint is one taken by secure programming languages such as Jif [41] and Flow Caml [45]. These languages ignore some kinds of interference such as changes to the timing and termination behavior of programs, arguing that these kinds of interference will have a minimal impact on security. However, overall, they continue to place very restrictive constraints on programs, prohibiting I/O in high security contexts, for instance. Allowing unchecked I/O would make it possible to leak too much secret information.

In our case, an appropriate balance point between usability and interference prevention is slightly more relaxed than in secure information-flow systems. We allow aspects to perform simple output I/O (more complex I/O will be discussed in the next chapter) and to alter the termination behavior of the program. We say that computation A does not *interfere with* computation B if A does not influence the final value produced by B. Computation A may change the timing and termination behavior of B (influencing whether or not B does indeed return a value) and it may perform simple output I/O.

Now suppose that A is advice and B is the main program. If we can establish that A does not interfere with B as defined above, programmers working on B can reason completely locally about *partial correctness* properties of their code. For these properties, they do not need to know anything about advice A or whether or not it has been applied to their code. For example, if we are working in a functional language like ML and enjoy equational reasoning, all our favorite (partial

correctness) equations continue to hold. If we are working in an imperative language and reason (perhaps informally) using Hoare logic-style pre- and post-conditions, the standard, commonly-used partial-correctness interpretation of these pre- and post-conditions continues to be valid. In other words, many of our most important local reasoning principles remain intact in the presence of harmless advice.

Every time a programmer writes new advice and can guarantee that advice is harmless, he or she will maintain the local reasoning principles so critical to reliable software development. Hence there is great incentive to write harmless advice whenever possible. Fortunately, our notion of harmless, non-interfering advice continues to support many common aspect-oriented applications, including the following broad application classes.

- Security. Harmless advice can check invariants at run-time, maintain access control tables, perform resource accounting, and terminate programs that disobey dynamic security policies. We will later demonstrate a case study in security, rewriting the security policies from Evans' thesis [16] in our system, and finding that all but one were harmless.

- Profiling. Harmless advice can maintain its own state separate from the mainline computation to gather statistics concerning the number of times different procedures are called. When the program terminates, the harmless advice can print out the profiling statistics.

- Program tracing and monitoring. Harmless advice can print out a variety of debugging information including when procedures are called and what data they are passed.

- Logging and backups. Harmless advice can back up data onto persistent secondary storage or make logs of events that occur during program execution for performance analysis, fault recovery, or postmortem security audits.

We have also accumulated anecdotal evidence indicating that several important applications appear to fall into this category, making it a useful abstraction. For example, IBM experimented with aspects in their middle-ware product line [8], finding them useful for such "harmless" tasks as enforcing consistency in tracing and logging concerns and for encapsulating monitoring and statistics components. We also observed a sequence of emails on the AspectJ users list [24] cataloging uses of aspects with Java projects. Many respondents specified that, in addition to some uncommon uses that they wished to highlight, they certainly used AspectJ for the common aspect-oriented concerns such as security, profiling, and monitoring mentioned above. Finally, we will present the aforementioned security case study on the Naccio execution monitoring system.

In the rest of this chapter, we develop a theory of harmless advice. Section 3.2, we define the core calculus, $\mathbb{F}_{HRM}$. As in the previous chapter, the calculus contains primitive notions of pointcuts and advice. We add a collection of static protection domains, arranged in a partial order. The main contribution of this chapter is to define a type system to guarantee that code, including advice, in a low-protection domain cannot influence execution of code in a high-protection domain. Though our type system is directly inspired by information-flow type systems for security [41, 45, 47], we take advantage of the syntactic separation between advice and code in HarmlessAML to simplify the type system of $\mathbb{F}_{HRM}$. The key technical result for $\mathbb{F}_{HRM}$ is a proof that the core calculus satisfies a noninterference property. The proof

adapts the technique used by Simonet and Pottier in their proof of noninterference for Flow Caml [45] to our aspect-oriented language.

Section 3.3 develops a surface language, HarmlessAML, that is more amenable to programming. HarmlessAML allows programmers to define aspects that are collections of state, objects, and advice. Each aspect operates in a separate static protection domain and does not interfere with the mainline computation or the other aspects. This section defines the syntax of HarmlessAML and establishes its semantics through a translation from source into core. We prove that HarmlessAML aspects are harmless by exploiting properties of the core calculus. In addition, we present example code implemented in our system and a case study in security, rewriting the security policies from Evans' thesis [16] in our system, and finding that all but one were harmless.

This chapter is an expansion of research published at the FOOL 2005 workshop [9] and the POPL 2006 conference [10].

3.2 Noninterfering Core Calculus: $\mathbb{F}_{HRM}$

As in the previous chapter, we extend and modify the WZL core calculus [37] to create the new "harmless" core calculus, $\mathbb{F}_{HRM}$. Research on the harmless $\mathbb{F}_{HRM}$ calculus and the polymorphic $\mathbb{F}_A$ core calculus from the previous chapter proceeded concurrently, so several features from each are not present in the other. Most importantly, the $\mathbb{F}_{HRM}$ calculus is not polymorphically-typed and the labels are not hierarchical. However, the $\mathbb{F}_{HRM}$ calculus contains mutable references, lacking in $\mathbb{F}_A$, to show that the noninterference result persists in the presence of program state. Because much of the common core calculus features were explored thoroughly in

the previous chapter, here we will only detail features that are new in this chapter. The two main features we will examine are labeled control-flow points and advice, both of which are variants of related constructs introduced by WZL.

Labels l, which are drawn from some countably infinite set, mark points in a computation at which advice may be triggered. We had discovered in the examples of Section 2.2 and the security case study of Section 2.3 that the `any` pointcut turned out in practice to be less useful than expected – as a result, we have eliminated the hierarchical nature of labels here for simplicity. Execution of a joinpoint $l[\![e_1]\!]; e_2$ proceeds by first evaluating e_1 until it reduces to a value v and at this point, any advice associated with the label l executes with v as an input. Once all advice associated with l has completed execution, control returns to the marked point and evaluation continues with e_2. Notably, a marked point $l[\![e_1]\!]$ has type unit, and no data are returned from the triggered advice. The advice triggering mechanism is much simpler than that of the previous chapter, in which labels marked control-flow points where data exchange could occur.

Harmless advice $\{pcd(x) \rhd e\}$ is a computation that is triggered whenever execution reaches the control-flow point described by the pointcut designator *pcd*. As before, *pcd*s are simply sets of labels $\{l_1, \ldots, l_k\}$. When advice is triggered, the value at the control-flow point is bound to x, which may be used within the body of the advice e. The advice body may have "harmless" effects (such as I/O), but it does not return any data to the mainline computation and consequently e is expected to have type unit.

The following example shows how harmless advice activation works (assuming that there is no other advice associated with label l in the environment). The code

prints `3: hello world`.

$$\Uparrow \{\{l\}(x) \rhd \texttt{printint x; print "} : \texttt{hello "}\};$$
$$\Uparrow \{\{l\}(y) \rhd \texttt{print "world"}\};$$
$$l[\![3]\!]$$

The expression **new** τ allows programs to generate a fresh label with type τ. Labels are considered first class values, so they may be passed to functions or stored in data structures before being used to mark control-flow points. For example, we might write the following code to allocate a new label and use it in two pieces of advice.

$$\texttt{let } pt \texttt{ = } \textbf{new } \mathsf{int} \texttt{ in}$$
$$\Uparrow \{\{pt\}(x) \rhd \texttt{print "hello "}\};$$
$$\Uparrow \{\{pt\}(y) \rhd \texttt{print "world"}\};$$
$$pt[\![3]\!]$$

3.2.1 Types for Enforcing Harmlessness

In order to protect the mainline computation from interference from advice, we have devised a type and effect system for the calculus we informally introduced in the previous section. The type system operates by ascribing a protection domain p to each expression in the language. These protection domains are organized in a lattice $\mathcal{L} = (\mathit{Protections}, \leq)$ where *Protections* is the set of possible protection domains and $\mathbf{P} \leq \mathbf{Q}$ specifies that $\mathbf{P}$ should not interfere with $\mathbf{Q}$. Alternatively, one might say that data in $\mathbf{Q}$ have higher integrity than data in $\mathbf{P}$. In our examples, we often assume there are **HIGH** and **LOW** protection levels with **LOW** < **HIGH**.

$\mathbf{P} \in$ Protections $\quad s \in$ Strings $\quad x \in$ Identifiers
$l \in$ Labels $\quad r \in$ Reference Locations

(*prot doms*)	p	::=	$\mathbf{P}$
(*types*)	τ	::=	$1 \mid \mathsf{string} \mid \mathsf{bool} \mid \tau_1 \times ... \times \tau_n \mid \tau \rightarrow_p \tau$
		$\mid$	$\mathsf{advice}_p \mid \tau\ \mathsf{label}_p \mid \tau\ \mathsf{ref}_p \mid \tau\ \mathsf{pc}_p$
(*values*)	v	::=	$() \mid s \mid \mathbf{true} \mid \mathbf{false} \mid (\overline{v}) \mid \{\overline{l}\}_p \mid \lambda_p x : \tau.e$
		$\mid$	$\{v(x) \rhd_p e\} \mid l \mid r$
(*expressions*)	e	::=	$v \mid x \mid e_1; e_2 \mid \mathbf{print}\ e \mid \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$
		$\mid$	$(\overline{e}) \mid \mathbf{split}\ (\overline{x}) = e\ \mathbf{in}\ e \mid e\ e \mid \{e(x) \rhd_p e\}$
		$\mid$	$\Uparrow e \mid \mathbf{new}_p\ \tau \mid e[\![e]\!] \mid \mathbf{ref}_p\ e \mid\ !e \mid e := e$
		$\mid$	$\{\overline{e}\}_p \mid e \cup_p e \mid p\langle e\rangle$
(*stores*)	S	::=	$. \mid S, r \rightarrow (v, \mathbf{P}) \mid S, l \rightarrow (\tau, \mathbf{P})$
(*aspect stores*)	A	::=	$. \mid A, v$
(*machine states*)	M	::=	$(S; A; \mathbf{P}; e)$
(*environments*)	Γ	::=	$. \mid \Gamma, x : \tau \mid \Gamma, r : \tau\ \mathsf{ref}_p \mid \Gamma, l : \tau\ \mathsf{label}_p$

Figure 3.1: Syntax of $\mathbb{F}_{HRM}$

Syntax of $\mathbb{F}_{HRM}$ In order to allow programmers to specify protection requirements we have augmented the syntax of the core language described in the previous section with a collection of protection annotations. The formal syntax appears in Figure 3.1.

The values include unit values and string and boolean constants. Programmers may also use n-ary tuples. Functions are annotated with the protection domain p in which they execute. This protection domain also shows up in the type of the function. Advice values $\{v(x) \rhd_p e\}$ are annotated with their protection domain as well. Labels l and reference locations r do not appear in initial programs; they only appear as programs execute and generate new labels and new references.

Most of the expression forms are fairly standard. For instance, in addition to values and variables, we allow ordinary expression forms for sequencing, printing strings, conditionals, tuples, function calls, and method invocations. Expressions

for introducing and eliminating advice were explained in the previous section. The expressions $\mathbf{new}_p\ \tau$ and $\mathbf{ref}_p\ e$ allocate labels that can be placed in protection domain p and references associated with protection domain p respectively. The last command $p\texttt{<}e\texttt{>}$ is a typing coercion that changes the current protection domain to the lower protection domain p.

3.2.2 Typing Judgments

The main typing judgment in our system has the form $\Gamma; p \vdash e : \tau$. It states that in the context Γ, expression e has type τ and may influence computations occurring in protection domains p or lower. A related judgment $\Gamma \vdash v : \tau$ checks that value v has type τ. Since values by themselves do not have effects that influence the computations, this latter judgment is not indexed by a protection domain. The context Γ maps variables, labels and reference locations to their types. We use the notation $\Gamma, x : \tau$ to extend Γ so that it maps x to τ. Whenever we extend Γ in this way, we assume that x does not already appear in the domain of Γ. Since we also treat all terms as equivalent up to alpha-renaming of bound variables, it will always be possible to find a variable x that does not appear in Γ when we need to. Figures 3.2, 3.3, and 3.4 contain the rules for typing expressions and values respectively.

The main goal of the typing relation is to guarantee that no values other than values with unit type (which have no information content) flow from a low protection domain to a high protection domain, although arbitrary data can flow in the other direction. This goal is very similar to, but not exactly the same as in, standard information flow systems such as Jif and Flow Caml. The latter systems actually do allow the flow of values from low contexts to high contexts, but mark all such values

with a low-protection type. Jif and Flow Caml typing rules make it impossible to use these low-protection objects in the high-protection context (without raising the protection of the object). In our system, we simply cut off the flow of low-protection values to high-protection contexts completely (aside from the unit value). We are able to do this in our setting, as there is a greater syntactic separation between high-integrity code (the mainline computation) and low-integrity code (the advice, written elsewhere) than there might be in a standard secure information-flow setting. We believe this is the right design choice for us because it simplifies the type system, as we do not have to annotate basic data such as booleans, strings, or tuples with information flow labels. A value is considered to be in the protection of the code in which the value exists.

Most of the value typing rules in Figure 3.2 are straightforward. For instance, the rule for functions $\lambda_p x : \tau.e$, states that the body of the function must be checked under the assumption that the code operates in protection domain p. The resulting type has the shape $\tau \rightarrow_p \tau'$. Checking our simple objects is similar: the type checker must verify that each method operates correctly in the declared protection domain. Labels and references are given types by the context. In the current calculus, pointcut designators are sets of labels. Unlike the other values, the rules for typing advice are fairly subtle. We will discuss these rules, along with the rules for typing labeled control-flow points, in a moment.

The first few expression typing rules in Figure 3.3 are standard rules for type systems that track information flow. The rule for **if** statements, Rule cet:if deviates slightly from the usual rule for tracking information flow. Normally, types for booleans will contain a security level and the branches of the **if** will be checked at a level equal to the join of the current security level and the level of the boolean.

Well-formed Values $\Gamma \vdash v : \tau$

$$\frac{}{\Gamma \vdash () : \mathsf{1}}\ \mathsf{cvt{:}unit} \qquad \frac{}{\Gamma \vdash s : \mathsf{string}}\ \mathsf{cvt{:}string} \qquad \frac{}{\Gamma \vdash \mathbf{true} : \mathsf{bool}}\ \mathsf{cvt{:}true}$$

$$\frac{}{\Gamma \vdash \mathbf{false} : \mathsf{bool}}\ \mathsf{cvt{:}false} \qquad \frac{(\Gamma \vdash v_i : \tau_i)^{1 \le i \le n}}{\Gamma \vdash (\overline{v}) : \tau_1 \times ... \times \tau_n}\ \mathsf{cvt{:}tup}$$

$$\frac{\Gamma, x : \tau; p \vdash e : \tau'}{\Gamma \vdash \lambda_p x : \tau.e : \tau \to_p \tau'}\ \mathsf{cvt{:}fun}$$

$$\frac{\Gamma \vdash v : \tau\ \mathsf{pc}_p \qquad \Gamma, x : \tau; p' \vdash e : \mathsf{1} \qquad \vdash p' \le p}{\Gamma \vdash \{v(x) \rhd_{p'} e\} : \mathsf{advice}_{p'}}\ \mathsf{cvt{:}adv}$$

$$\frac{\Gamma(l) = \tau\ \mathsf{label}_p}{\Gamma \vdash l : \tau\ \mathsf{label}_p}\ \mathsf{cvt{:}lab} \qquad \frac{\Gamma(r) = \tau\ \mathsf{ref}_p}{\Gamma \vdash r : \tau\ \mathsf{ref}_p}\ \mathsf{cvt{:}ref}$$

$$\frac{(\Gamma \vdash v_i : \tau\ \mathsf{label}_{p_i})^{(1 \le i \le n)} \qquad (\vdash p \le p_i)^{(1 \le i \le n)}}{\Gamma \vdash \{\bar{l}\}_p : \tau\ \mathsf{pc}_p}\ \mathsf{cvt{:}pc}$$

Figure 3.2: Value Typing of $\mathbb{F}_{HRM}$

Well-formed Expressions $\Gamma; p \vdash e : \tau$

$$\frac{\Gamma \vdash v : \tau}{\Gamma; p \vdash v : \tau}\ \textsf{cet:val} \qquad \frac{\Gamma(x) = \tau}{\Gamma; p \vdash x : \tau}\ \textsf{cet:var}$$

$$\frac{\Gamma; p \vdash e_1 : 1 \qquad \Gamma; p \vdash e_2 : \tau}{\Gamma; p \vdash e_1; e_2 : \tau}\ \textsf{cet:seq} \qquad \frac{\Gamma; p \vdash e : \textsf{string}}{\Gamma; p \vdash \textbf{print}\ e : 1}\ \textsf{cet:print}$$

$$\frac{\Gamma; p \vdash e_1 : \textsf{bool} \qquad \Gamma; p \vdash e_2 : \tau \qquad \Gamma; p \vdash e_3 : \tau}{\Gamma; p \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \tau}\ \textsf{cet:if}$$

$$\frac{(\Gamma; p \vdash e_i : \tau_i)^{1 \le i \le n}}{\Gamma; p \vdash (\overline{e}) : \tau_1 \times ... \times \tau_n}\ \textsf{cet:tup}$$

$$\frac{\Gamma; p \vdash e_1 : \tau_1 \times ... \times \tau_n \qquad \Gamma, (\overline{x} : \overline{t}); p \vdash e_2 : \tau}{\Gamma; p \vdash \textbf{split}\ (\overline{x}) = e_1\ \textbf{in}\ e_2 : \tau}\ \textsf{cet:split}$$

$$\frac{\Gamma; p \vdash e_1 : \tau_1 \to_p \tau_2 \qquad \Gamma; p \vdash e_2 : \tau_1}{\Gamma; p \vdash e_1\ e_2 : \tau_2}\ \textsf{cet:app}$$

$$\frac{\Gamma; p \vdash e_1 : \tau\ \textsf{pc}_{p'} \qquad \Gamma, x : \tau; p'' \vdash e_2 : 1 \qquad \vdash p'' \le p'}{\Gamma; p \vdash \{e_1(x) \rhd_{p''} e_2\} : \textsf{advice}_{p''}}\ \textsf{cet:adv}$$

Figure 3.3: Expression Typing of $\mathbb{F}_{HRM}$: Part 1

Well-typed Expressions $\Gamma; p \vdash e : \tau$

$$\frac{\Gamma; p \vdash e : \mathsf{advice}_{p'} \qquad \vdash p' \leq p}{\Gamma; p \vdash \Uparrow e : 1}\ \mathsf{cet{:}advinst} \qquad \frac{\vdash p' \leq p}{\Gamma; p \vdash \mathbf{new}_{p'}\ \tau : \tau\ \mathsf{label}_{p'}}\ \mathsf{cet{:}lab}$$

$$\frac{\Gamma; p \vdash e_1 : \tau\ \mathsf{label}_p \qquad \Gamma; p \vdash e_2 : \tau}{\Gamma; p \vdash e_1[\![e_2]\!] : 1}\ \mathsf{cet{:}jp} \qquad \frac{\Gamma; p \vdash e : \tau \qquad \vdash p' \leq p}{\Gamma; p \vdash \mathbf{ref}_{p'}\ e : \tau\ \mathsf{ref}_{p'}}\ \mathsf{cet{:}ref}$$

$$\frac{\Gamma; p \vdash e : \tau\ \mathsf{ref}_{p'} \qquad \vdash p \leq p'}{\Gamma; p \vdash !e : \tau}\ \mathsf{cet{:}asgn}$$

$$\frac{\Gamma; p \vdash e_1 : \tau\ \mathsf{ref}_{p'} \qquad \Gamma; p \vdash e_2 : \tau \qquad \vdash p' \leq p}{\Gamma; p \vdash e_1\ :=\ e_2 : \tau}\ \mathsf{cet{:}deref}$$

$$\frac{(\Gamma; p \vdash e_i : \tau\ \mathsf{label}_{p_i})^{(1 \leq i \leq n)} \qquad (\vdash p' \leq p_i)^{(1 \leq i \leq n)}}{\Gamma; p \vdash \{\overline{e}\}_{p'} : \tau\ \mathsf{pc}_{p'}}\ \mathsf{cet{:}pc}$$

$$\frac{\Gamma; p \vdash e_1 : \tau\ \mathsf{pc}_{p_1} \qquad \vdash p' \leq p_1 \qquad \Gamma; p \vdash e_2 : \tau\ \mathsf{pc}_{p_2} \qquad \vdash p' \leq p_2}{\Gamma; p \vdash e_1 \cup_{p'} e_2 : \tau\ \mathsf{pc}_{p'}}\ \mathsf{cet{:}union}$$

$$\frac{\Gamma; p' \vdash e : 1 \qquad \vdash p' \leq p}{\Gamma; p \vdash p'\langle e \rangle : 1}\ \mathsf{cet{:}low}$$

Figure 3.4: Expression Typing of $\mathbb{F}_{HRM}$: Part 2

However, in our system, any data, including booleans, manufactured by code at level p contains level p information. Consequently, the branches of the **if** statement may be safely checked at level p. The typing rules for function calls, Rule cet:app, require that the function in question be safe to run at the current protection level p.

The typing rules for references in Figure 3.4 enforce the usual integrity constraint found in information-flow systems. In Rule cet:deref, when in protection domain p, we are allowed to dereference references in protection domain p' when p is less than or equal to p'. In Rule cet:ref and cet:asgn, we are allowed to create and store to references in protection domain p' only if our current domain p is greater than or equal to p'.

Rule cet:low in Figure 3.4 is a typing coercion that changes the protection level. It is legal for the protection level to be lowered from p to p' when no information flows back from the computation e to be executed. We prevent this information flow by constraining the result type of e to be $\mathsf{1}$. One might wonder whether the following dual rule, which allows one to raise the protection level, is sound in our system:

$$\frac{\cdot; p' \vdash e : \tau \qquad \vdash p \leq p'}{\Gamma; p \vdash p' \,\texttt{>}e\texttt{<} : \tau}\ \textsf{cet:high}$$

This rule raises the protection domain for the expression e and allows information to flow out of the expression, but does not allow any information to flow in. In the context of the features we have looked at so far, this rule appears sound, but in combination with the context-sensitive advice we will introduce in Section 3.2.4, it

is not. Fortunately, the rule does not appear useful in our application and we have omitted it.[1]

The last component of our type system involves the rules for typing advice and marking control-flow points. If we want to ensure that low-protection code cannot interfere with high-protection code by manipulating advice and control-flow labels, we must be sure that low-protection code cannot do either of the following:

1. Declare and activate high-protection advice. For instance, assume r is a high-protection reference with type $\mathsf{int\ ref}_{\mathbf{high}}$ and l is a label that has been placed in high-protection code. If we allow the expression

$$\Uparrow \{l(x) \rhd_{\mathbf{high}} r := 3 + x\};\ e$$

   to appear in low-protection code, then this low privilege code can indirectly cause writes to the reference r.

2. Mark a control-flow point with a label that triggers high-protection advice. For instance, assume that

$$\{l(x) \rhd_{\mathbf{high}} r := 3 + x\}$$

   is an active piece of high-protection advice which writes to the high-protection reference r. Placing the label l in low-protection code allows low-protection code to determine via its control-flow, when the high-protection advice will run and write to r.

[1] There may well be some strategy that allows us to add this rule together with the context-sensitive advice of Section 3.2.4. However, the naive approach does not appear to work. Rather then complicating the type structure or operational semantics for something we do not need, we leave it out.

In order to properly protect high-protection code in the face of these potential errors, we do the following.

1. Add protection levels to advice types (e.g., $\mathsf{advice}_{\mathbf{high}}$), which will allow us to prevent advice from being activated in the illegal contexts. (eg. low-protection contexts)

2. Add protection levels to label types (e.g., $\mathsf{string\ label}_{\mathbf{high}}$) which will allow us to prevent labels being placed in illegal spots. (eg. low-protection contexts)

One might hope that it would be possible to simplify the system and add protection levels to only one of the two constructs, but doing so leads to unsoundness.

Several typing rules in the middle of Figures 3.3 and 3.4 give the well-formedness conditions for advice and labels. Notice that in Rule cet:adv, the rule for typing advice introduction ($\{e_1(x) \rhd_p e_2\}$), the protection level of the advice, and therefore the protection level the body of the advice must operate under, is connected to the protection level of the label that triggers it. Second, given a high-protection piece of advice in Rule cet:advinst, this advice cannot be installed ($\Uparrow e$) from low-protection code. Finally, notice that when marking a control-flow point with a label ($e_1[\![e_2]\!]$) in Rule cet:jp, the protection level of the label is connected to the protection level of the expression at that point. The result of these constraints is that when in a low-protection zone, there is no way to cause execution of high-protection advice.

3.2.3 Operational Semantics

The definition of the operational semantics for $\mathbb{F}_{HRM}$ is very similar those of the previous chapter. In particular, we use a context-based semantics where contexts E specify a left-to-right, call-by-value evaluation relation. The top-level operational

$\boxed{\text{Reduction } (\Sigma; A; p; e) \longmapsto (\Sigma'; A'; p; e')}$

$$\frac{(\Sigma; A; p; e) \longmapsto_\beta (\Sigma'; A'; p; e')}{(\Sigma; A; p; e) \longmapsto (\Sigma'; A'; p; e')}\ \textsf{ce:beta}$$

$$\frac{(\Sigma; A; p; e) \longmapsto (\Sigma'; A'; p; e)}{(\Sigma; A; p; E[e]) \longmapsto (\Sigma'; A'; p; E[e'])}\ \textsf{ce:eval}$$

$$\frac{(\Sigma; A; p'; e) \longmapsto (\Sigma'; A'; p'; e')}{(\Sigma; A; p; p'\texttt{<}e\texttt{>}) \longmapsto (\Sigma'; A'; p; p'\texttt{<}e'\texttt{>})}\ \textsf{ce:low}$$

Figure 3.5: Operational Semantics for $\mathbb{F}_{HRM}$

$\boxed{\beta\text{-reduction } (\Sigma; A; p; e) \longmapsto_\beta (\Sigma'; A'; p; e')}$

$$\frac{}{(\Sigma; A; p; (); e) \longmapsto_\beta (\Sigma; A; p; e)}\ \textsf{ceb:seq}$$

$$\frac{}{(\Sigma; A; p; \mathbf{print}\ s) \longmapsto_\beta (\Sigma; A; p; ())}\ \textsf{ceb:print}$$

$$\frac{}{(\Sigma; A; p; \mathbf{if\ true\ then}\ e_1\ \mathbf{else}\ e_2) \longmapsto_\beta (\Sigma; A; p; e_1)}\ \textsf{ceb:ifthen}$$

$$\frac{}{(\Sigma; A; p; \mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2) \longmapsto_\beta (\Sigma; A; p; e_2)}\ \textsf{ceb:ifelse}$$

$$\frac{}{(\Sigma; A; p; \mathbf{split}\ (\overline{x}) = (\overline{v})\ \mathbf{in}\ e) \longmapsto_\beta (\Sigma; A; p; e[\overline{v}/\overline{x}])}\ \textsf{ceb:split}$$

$$\frac{}{(\Sigma; A; p; \lambda_p x : t.e\ v) \longmapsto_\beta (\Sigma; A; p; e[v/x])}\ \textsf{ceb:app}$$

$$\frac{}{(\Sigma; A; p; \Uparrow \{v(x) \rhd_{p'} e_1\}) \longmapsto_\beta (\Sigma; (A, \{v(x) \rhd_{p'} e_1\}); p; ())}\ \textsf{ceb:advinst}$$

Figure 3.6: β-redex Operational Semantics for $\mathbb{F}_{HRM}$: Part 1

$\boxed{\beta\text{-reduction } (\Sigma; A; p; e) \longmapsto_\beta (\Sigma'; A'; p; e')}$

$$\frac{l \notin \Sigma}{(\Sigma; A; p; \mathbf{new}_{p'}\ \tau) \longmapsto_\beta ((\Sigma, l); A; p; l)}\ \textsf{ceb:newlab}$$

$$\frac{l \in \Sigma \qquad \mathcal{A}[\![A]\!]_{l[\![v]\!]} = e}{(\Sigma; A; p; l[\![v]\!]) \longmapsto_\beta (\Sigma; A; p; e)}\ \textsf{ceb:jp}$$

$$\frac{r \notin \Sigma}{(\Sigma; A; p; \mathbf{ref}_{p'}\ v) \longmapsto_\beta ((\Sigma, r = v); A; p; r)}\ \textsf{ceb:newref}$$

$$\frac{}{(\Sigma; A; p; !r) \longmapsto_\beta (\Sigma; A; p; \Sigma(r))}\ \textsf{ceb:deref}$$

$$\frac{}{(\Sigma; A; p; r\ :=\ v) \longmapsto_\beta ((\Sigma, r = v); A; p; v)}\ \textsf{ceb:assign}$$

$$\frac{}{(\Sigma; A; p; \{\overline{l_1}\}_{p_1} \cup_{p'} \{\overline{l_2}\}_{p_2}) \longmapsto_\beta (\Sigma; A; p; \{\overline{l_1}\ \overline{l_2}\}_{p'})}\ \textsf{ceb:union}$$

$$\frac{}{(\Sigma; A; p; p'\texttt{<}()\texttt{>}) \longmapsto_\beta (\Sigma; A; p; ())}\ \textsf{ceb:low}$$

Figure 3.7: β-redex Operational Semantics for $\mathbb{F}_{HRM}$: Part 2

Advice Composition $\mathcal{A}[\![A]\!]_{l[\![v]\!]} = e$

$$\frac{}{\mathcal{A}[\![\cdot]\!]_{l[\![v]\!]} = ()}\ \textsf{cac:end} \qquad \frac{l \in \{\bar{l}\}_{p'} \qquad \mathcal{A}[\![A]\!]_{l[\![v]\!]} = e}{\mathcal{A}[\![\{\{\bar{l}\}_{p'}(x) \rhd_p e'\}, A]\!]_{l[\![v]\!]} = p\langle e'[v/x]\rangle; e}\ \textsf{cac:match}$$

$$\frac{l \notin \{\bar{l}\}_{p'} \qquad \mathcal{A}[\![A]\!]_{l[\![v]\!]} = e}{\mathcal{A}[\![\{\{\bar{l}\}_{p'}(x) \rhd_p e'\}, A]\!]_{l[\![v]\!]} = e}\ \textsf{cac:nomatch}$$

Figure 3.8: Aspect Composition for $\mathbb{F}_{HRM}$

judgment has the form $(\Sigma; A; p; e) \longmapsto (\Sigma'; A'; p; e')$ where Σ collects the labels l that may be used to mark control-flow points and also maps reference locations r to values. The meta-variable A represents an advice store, which is a list of advice. The current protection level of the code is p. The protection level does not influence execution of the code, and could be omitted, but is useful to consider in our noninterference proof. Most of the real work is done by the auxiliary relation $(\Sigma, A, p, e) \longmapsto_\beta (\Sigma'; A'; p; e')$. The syntax of stores and advice stores is given below.

$$\begin{array}{lrcl} (\textit{stores}) & \Sigma & ::= & \cdot \mid \Sigma, r = e \mid \Sigma, l \\ (\textit{asp stores}) & A & ::= & \cdot \mid A, \{v(x) \rhd_p e\} \end{array}$$

The definitions of these relations can be found in Figures 3.6 and 3.7. Notice that Rule ceb:jp for marked control-flow points depends upon an auxiliary function $\mathcal{A}[\![A]\!]_{l[\![v]\!]} = e$. This advice composition function, defined in Figure 3.8, selects all advice in A that is triggered by the label l and combines their bodies to form the expression e. Finally, an abstract machine configuration $(\Sigma; A; p; e)$ is well-typed if it satisfies the judgment $\vdash (\Sigma; XSA; p; e)$ **ok** specified in Figure 3.9.

Well-formed Stores $\vdash \Sigma : \Gamma$

$$\frac{\begin{array}{c} dom(\Sigma) = dom(\Gamma) \\ \forall r \in dom(\Sigma).\ \Gamma(r) = \tau\ \mathsf{ref}_p \quad \Gamma \vdash \Sigma(r) : \tau\ \textbf{for some}\ p, \tau \\ \forall l \in dom(\Sigma).\ \Gamma(l) = \tau\ \mathsf{label}_p\ \textbf{for some}\ p, \tau \end{array}}{\vdash \Sigma : \Gamma}\ \mathsf{camt{:}labrefstore}$$

Well-formed Advice Stores $\Gamma \vdash A\ \mathbf{ok}$

$$\frac{}{\Gamma \vdash \cdot\ \mathbf{ok}}\ \mathsf{camt{:}advstoreend}$$

$$\frac{\Gamma \vdash a : \mathsf{advice}_p\ \textbf{for some}\ p \qquad \Gamma \vdash A\ \mathbf{ok}}{\Gamma \vdash A, a\ \mathbf{ok}}\ \mathsf{camt{:}advstorenext}$$

Well-formed Machine States $\vdash (\Sigma; A; p; e)\ \mathbf{ok}$

$$\frac{\vdash \Sigma : \Gamma \qquad \Gamma \vdash A\ \mathbf{ok} \qquad \Gamma; p \vdash e : \tau\ \textbf{for some}\ \tau}{\vdash (\Sigma; A; p; e)\ \mathbf{ok}}\ \mathsf{camt{:}ms}$$

Figure 3.9: Well-formed Machine States in $\mathbb{F}_{HRM}$

$$
\begin{array}{llll}
(\textit{types}) & \tau & ::= & \ldots \mid \mathsf{stack} \\
(\textit{values}) & v & ::= & \ldots \mid \cdot \mid l[\![v]\!] :: v \\
(\textit{expressions}) & e & ::= & \ldots \mid \textbf{stack}() \mid \textbf{store}\ e[\![e]\!]\ \textbf{in}\ e \\
 & & \mid & \textbf{stkcase}\ e\ (\textit{pat} \Rightarrow e, \_ \Rightarrow e) \\
(\textit{stack patterns}) & \textit{pat} & ::= & [] \mid e[\![x]\!] :: \textit{pat} \mid \_ :: \textit{pat} \mid x \\
(\textit{stack val pats}) & \textit{vpat} & ::= & [] \mid \{\bar{l}\}_p[\![x]\!] :: \textit{vpat} \mid \_ :: \textit{vpat} \mid x \\
(\textit{eval contexts}) & E & ::= & \ldots \mid \textbf{store}\ E[\![e]\!]\ \textbf{in}\ e \mid \textbf{store}\ l[\![E]\!]\ \textbf{in}\ e \\
 & & \mid & \textbf{store}\ l[\![v]\!]\ \textbf{in}\ E \mid \textbf{stkcase}\ E\ (\textit{pat} \Rightarrow e, \_ \Rightarrow e) \\
 & & \mid & \textbf{stkcase}\ v\ (\textit{Epat} \Rightarrow e, \_ \Rightarrow e) \\
(\textit{pat eval ctxts}) & \textit{Epat} & ::= & E[\![x]\!] :: \textit{pat} \mid \{\bar{l}\}_p[\![x]\!] :: \textit{Epat} \mid \_ :: \textit{Epat} \\
(\textit{top eval ctxts}) & F & ::= & \ldots \mid [\![]\!] \mid \textbf{store}\ l[\![v]\!]\ \textbf{in}\ F \mid p\langle F \rangle \\
 & & \mid & E[\![F]\!] \text{ where } E \neq \textbf{store}\ l[\![v]\!]\ \textbf{in}\ F
\end{array}
$$

Figure 3.10: Context Sensitive Advice Syntax of $\mathbb{F}_{HRM}$

3.2.4 Extensions

Context-sensitive Advice The advice defined in previous sections could not analyze the call stack from which it was activated. Programming languages such as AspectJ allow this flexibility via special pointcut designators such as **cflow**. As in the previous chapter, we describe a fully general facility for analysis of information on the current call stack. Figure 3.10 outlines the syntactic extensions to $\mathbb{F}_{HRM}$.

In order to program with context-sensitive advice, programmers grab the current stack using the **stack**() command. Data is explicitly allocated on the stack using the command **store** $e_1[\![e_2]\!]$ **in** e_3, where e_1 is a label and e_2 represents a value associated with the label. e_2 is typically used to store the value passed into the control flow point marked by the label. The **store** command evaluates e_1 to a label l and e_2 to a value v_2, places $l[\![v_2]\!]$ on the stack, evaluates e_3 to a value v_3 and finally removes $l[\![v_2]\!]$ from the stack and returns v_3. The programmer may examine a stack data structure using the **stkcase** e $(\textit{pat} \Rightarrow e, \_ \Rightarrow e)$ command, which matches the stack e against the pattern *pat*. If there is a match, the first branch is executed;

otherwise, the second branch is executed. There are patterns that match the empty stack ($\cdot$), patterns that match a stack starting with any label in a particular set ($\{\bar{l}\}_p[\![x]\!] :: pat$) where x is bound to the value associated with the label on the top of the stack if it is in the label set, patterns that match a stack starting with anything at all ($\_ :: pat$), and patterns involving stack variables (x).

The typing rules for these extensions appear in Figure 3.11. There are three sets of rules in this figure. The first two sets extend the value typing and expression typing relations respectively. The last set of rules gives types to patterns where the type of a pattern is a context Γ that describes the types of the variables bound within the pattern.

The rules for evaluating these new expressions appear in Figure 3.12. Again, there are three sets of rules. The first set defines a new set of top-level evaluation rules, and the second set adds additional β-evaluation rules. Notice that the top-level rule for evaluating the stack primitive uses an auxiliary function $\mathcal{S}(F)$ that extracts the current stack of values from F contexts, which contains evaluation context E's, and $p\langle F\rangle$ contexts. Here, we use the notation $st@X$ to append the object X to the bottom of the stack st. Also, notice that there are three levels of operational semantics (β, normal, and top) as opposed to the two of the previous chapter. This is an artifact of the concurrent development of harmless core calculus and the polymorphic $\mathbb{F}_A$—the context-sensitive advice should behave similarly in both systems. The last set of rules conclude in judgments with the form $st \models vpat \Rightarrow sub$. These rules describe the circumstances under which a stack st matches an (evaluated) pattern $vpat$ and generates a substitution of values for variables sub.

For the most part, it is relatively straightforward to reassure oneself that these extensions will not disrupt the noninterference properties that our language pos-

Well-formed Values $\Gamma \vdash v : \tau$

$$\frac{}{\Gamma \vdash \cdot : \mathsf{stack}}\ \mathsf{cvt{:}stackend}$$

$$\frac{\Gamma \vdash l : \tau\ \mathsf{label}_p \qquad \Gamma \vdash v_1 : \tau \qquad \Gamma \vdash v_2 : \mathsf{stack}}{\Gamma \vdash l[\![v_1]\!] :: v_2 : \mathsf{stack}}\ \mathsf{cvt{:}stacknext}$$

Well-formed Expressions $\Gamma; p \vdash e : \tau$

$$\frac{}{\Gamma; p \vdash \mathbf{stack}() : \mathsf{stack}}\ \mathsf{cet{:}stack}$$

$$\frac{\Gamma; p \vdash e_1 : \tau'\ \mathsf{label}_{p'} \qquad \Gamma; p \vdash e_2 : \tau' \qquad \Gamma; p \vdash e_3 : \tau}{\Gamma; p \vdash \mathbf{store}\ e_1[\![e_2]\!]\ \mathbf{in}\ e_3 : \tau}\ \mathsf{cet{:}store}$$

$$\frac{\Gamma; p \vdash e_1 : \mathsf{stack} \qquad \Gamma; p \vdash pat \Rightarrow \Gamma' \qquad \Gamma, \Gamma'; p \vdash e_2 : \tau \qquad \Gamma; p \vdash e_3 : \tau}{\Gamma; p \vdash \mathbf{stkcase}\ e_1\ (pat \Rightarrow e_2, \_ \Rightarrow e_3) : \tau}\ \mathsf{cet{:}scase}$$

Well-formed Patterns $\Gamma; p \vdash pat \Rightarrow \Gamma$

$$\frac{}{\Gamma; p \vdash [\,] \Rightarrow \cdot}\ \mathsf{cpt{:}nil} \qquad \frac{\Gamma; p \vdash e : \tau\ \mathsf{pc}_{p'} \qquad \Gamma; p \vdash pat \Rightarrow \Gamma'}{\Gamma; p \vdash e[\![x]\!] :: pat \Rightarrow \Gamma', x : \tau}\ \mathsf{cpt{:}jp}$$

$$\frac{\Gamma; p \vdash pat \Rightarrow \Gamma'}{\Gamma; p \vdash \_ :: pat \Rightarrow \Gamma'}\ \mathsf{cpt{:}any} \qquad \frac{}{\Gamma; p \vdash x \Rightarrow \cdot, x : \mathsf{stack}}\ \mathsf{cpt{:}var}$$

Figure 3.11: Stack Typing of $\mathbb{F}_{HRM}$

Stack Reification $\mathcal{S}(e)$

$$\mathcal{S}([\![]\!]) = \cdot \qquad \mathcal{S}(\mathbf{store}\ l[\![v]\!]\ \mathbf{in}\ F) = \mathcal{S}(F)\ @\ (l[\![v]\!]) \qquad \mathcal{S}(p\langle F\rangle) = \mathcal{S}(F)$$

$$\mathcal{S}(E[F]) = \mathcal{S}(F) \text{ when } E \neq \mathbf{store}\ l[\![v]\!]\ \mathbf{in}\ F$$

Top Reduction $(S; A; p; e) \longmapsto_{top} (S'; A'; p; e')$

$$\frac{(\Sigma; A; p; e) \longmapsto (\Sigma'; A'; p; e')}{(\Sigma; A; p; e) \longmapsto_{top} (\Sigma'; A'; p; e')}\ \textsf{cevt:ce}$$

$$\frac{}{(\Sigma; A; p; F[\mathbf{stack}()]) \longmapsto_{top} (\Sigma; A; p; F[\mathcal{S}(F)])}\ \textsf{cevt:stack}$$

β-reduction $(S; A; p; e) \longmapsto_{\beta} (S; A; p; e)$

$$\frac{}{(\Sigma; A; p; \mathbf{store}\ v_1[\![v_2]\!]\ \mathbf{in}\ v_3) \longmapsto_{\beta} (\Sigma; A; p; v_3)}\ \textsf{ceb:store}$$

$$\frac{v \models vpat \Rightarrow sub}{(\Sigma; A; p; \mathbf{stkcase}\ v\ (vpat \Rightarrow e_1, \_ \Rightarrow e_2)) \longmapsto_{\beta} (\Sigma; A; p; sub(e_1))}\ \textsf{ceb:scaseyes}$$

$$\frac{v \not\models vpat \Rightarrow sub}{(\Sigma; A; p; \mathbf{stkcase}\ v\ (vpat \Rightarrow e_1, \_ \Rightarrow e_2)) \longmapsto_{\beta} (\Sigma; A; p; e_2)}\ \textsf{ceb:scaseno}$$

Stack Pattern Matching $v \models vpat \Rightarrow sub$

$$\frac{}{\cdot \models [] \Rightarrow \cdot}\ \textsf{csm:nil} \qquad \frac{l \in \{\bar{l}\}_p \qquad v_2 \models vpat \Rightarrow sub}{l[\![v_1]\!] :: v_2 \models \{\bar{l}\}_p[\![x]\!] :: vpat \Rightarrow sub; [v_1/x]}\ \textsf{csm:jp}$$

$$\frac{v_2 \models vpat \Rightarrow sub}{l[\![v_1]\!] :: v_2 \models \_ :: vpat \Rightarrow sub}\ \textsf{csm:any} \qquad \frac{}{v \models x \Rightarrow [v/x]}\ \textsf{csm:var}$$

Figure 3.12: Stack Matching for $\mathbb{F}_{HRM}$

sesses. However, there is one major subtlety to consider: the **stack**() primitive. In order for this primitive to be safe, it must be the case that whenever it is activated in a high-level context, there is no low-level data on the stack, which could influence execution in that high-level context. Fortunately, this is indeed the case. The only way to switch protection levels from one evaluation context to the next is via the context $p\texttt{<}E\texttt{>}$, which lowers the protection level. Consequently, any use of the **stack**() command is done in the context that looks like $p_1\texttt{<}E_1[p_2\texttt{<}E_2[p_3\texttt{<}E_3\texttt{>}]\texttt{>}]\texttt{>}$ where $p_3 \leq p_2 \leq p_1$. So while a low-level expression can read high-level data via the **stack**() command and subsequent `case` expressions, the opposite is not possible. We are safe.

Polymorphic Protection Domains In $\mathbb{F}_{HRM}$, functions, labels, pointcuts, and advice exist in a particular protection domain. As a result, libraries cannot be built whose functions can be used by both the main program and aspects. The library must be redeclared in the main program and each aspect.

We have explored adding polymorphic protection domains to our system. The most obvious method adds a polymorphic protection domain type $\forall\rho.\tau$ to $\mathbb{F}_{HRM}$. We would also add the protection domain function abstraction value $\Lambda\rho.v$ and the protection domain function application $e[p]$.

In addition, as in the previous chapter, labels that are embedded in polymorphic functions, pointcuts that are triggered by those labels, and advice that contain those pointcuts must also indicate that they are valid for any protection domain ρ. As such, polymorphism would need to be added to the protection domain annotations on the types and syntax of labels, pointcuts, and advice. Figure 3.2.4 presents

$$\begin{array}{llll}
\multicolumn{4}{l}{\mathbf{P} \in \text{Protections} \qquad x, \rho \in \text{Identifiers} \qquad l \in \text{Labels}} \\
(\textit{protdoms}) & p & ::= & \rho \mid \mathbf{P} \\
(\textit{types}) & \tau & ::= & \ldots \mid \forall \rho.p \mid \mathsf{advice}_{\overline{\rho}.p} \mid \tau\ \mathsf{label}_{\overline{\rho}.p} \mid \tau\ \mathsf{pc}_{\overline{\rho}.p} \\
(\textit{values}) & v & ::= & \ldots \mid \{\overline{l}\}_{\overline{\rho}.p} \mid \Lambda\rho.v \mid \{v(x) \rhd_{\overline{\rho}.p} e\} \\
(\textit{expressions}) & e & ::= & \ldots \mid e[e] \mid \{e(x) \rhd_{\overline{\rho}.p} e\} \mid \mathbf{new}_{\overline{\rho}.p}\ \tau \\
 & & \mid & e[\![\overline{\rho}]\!][[e]] \mid \{\overline{e}\}_{\overline{\rho}.p} \mid e \cup_{\overline{\rho}.p} e \\
(\textit{protdomenvs}) & R & ::= & \cdot \mid R, \rho \\
(\textit{environments}) & \Gamma & ::= & \ldots \mid \Gamma, l : \tau\ \mathsf{label}_{\overline{\rho}.p}
\end{array}$$

Figure 3.13: Postulated Polymorphic Protection Domain Grammar for $\mathbb{F}_{HRM}$

a polymorphic protection domain extension to $\mathbb{F}_{HRM}$ in a manner similar to the addition of polymorphic types to advice in Chapter 2.

However, we have discovered that due to complexities in the HarmlessAML noninterference proof that will not be fully explored until Section 3.3, there are several difficulties with adding such polymorphic protection domains to the language. The design and properties of HarmlessAML depends on a particular division of mainline code from aspect code, where all aspect code exists in a strictly lower protection domain than top-protection, mainline code. This particular structuring of protection domains by the translation from source to core will allow us to use properties of the core language to prove a harmlessness theorem about the source language.

As we will see in Section 3.3, the translation of HarmlessAML function declarations explicitly declares *before* and *after* labels that can be used to trigger primitive $\mathbb{F}_{HRM}$ advice. Because of the protection domain ordering that is required by Rule cet:lab, such label creation expressions could only be used by top-protection, mainline code. Therefore, polymorphic library functions could only be defined in the main program. As such, aspects added after the fact could not add harmless libraries to the system.

More importantly, due to the way advice stores are implemented in $\mathbb{F}_{HRM}$ and due to Rule cet:advinst for advice installation typing, advice with a polymorphic protection domain annotation could only be installed into the advice store from the top protection domain in the lattice. Due again to the particular structuring of protection domains by the translation from source to core, adding polymorphic advice that must be installed from the mainline code protection domain would disallow such a translation, violating the careful syntactic separation we have maintained between advice code and mainline program code.

As such, the restrictions that would be added by polymorphic system libraries seem as prohibitive as the current requirements that libraries must be duplicated. We will continue our exploration of how to use system libraries with advice in Chapter 4.

3.2.5 Meta-theory

To prove noninterference, we use the technique developed by Simonet and Pottier [45]. In order to do so, we initially assume the collection of protection domains has been divided into two groups, the high protection domains (H) and the low protection domains (L). The low-protection group is a downward-closed subset of protection domains and the high-protection group contains all other protection domains. The goal is to prove that low-protection code cannot interfere with the behavior of high-protection code, no matter how aspects, references or labels are used. Overall, our proof may be broken down into four main steps (See also Figure 3.14):

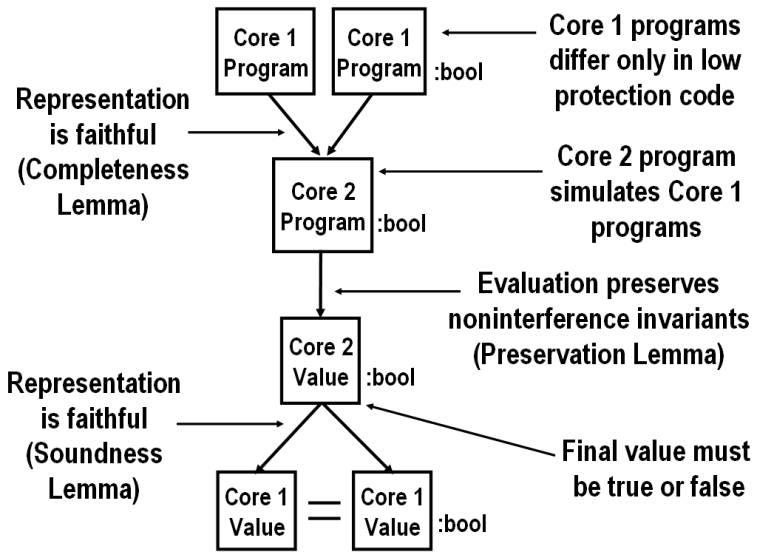

Figure 3.14: Noninterference Proof Diagram for $\mathbb{F}_{HRM}$

- Define a new language $\mathbb{F}_{HRM2}$ whose programs simulate execution of two $\mathbb{F}_{HRM}$ programs.
- Show $\mathbb{F}_{HRM2}$ is a correct simulation of $\mathbb{F}_{HRM}$ programs via Soundness and Completeness theorems.
- Prove $\mathbb{F}_{HRM2}$ is a safe language and preserves the noninterference invariants via the standard Progress and Preservation theorems.
- Put the theorems above together to prove the noninterference result for $\mathbb{F}_{HRM}$

Defining $\mathbb{F}_{HRM2}$

We begin by defining a new calculus $\mathbb{F}_{HRM2}$ that simulates the simultaneous execution of two of $\mathbb{F}_{HRM}$ expressions. The $\mathbb{F}_{HRM2}$ grammar is described in Figure 3.15. The main syntactic difference between $\mathbb{F}_{HRM}$ expressions and $\mathbb{F}_{HRM2}$ expressions

$$
\begin{array}{llll}
(\textit{simult exprs}) & e & ::= & \ldots \mid p\texttt{<}e\texttt{|}e\texttt{>} \\
(\textit{simult values}) & v^2 & ::= & v \mid \texttt{<}v\texttt{|}v\texttt{>} \mid \texttt{<}v\texttt{|}\textit{void}\texttt{>} \mid \texttt{<}\textit{void}\texttt{|}v\texttt{>} \\
(\textit{simult types}) & \tau^2 & ::= & \tau \mid \texttt{<}\tau\texttt{|}\textit{void}\texttt{>} \mid \texttt{<}\textit{void}\texttt{|}\tau\texttt{>} \\
(\textit{simult aspects}) & a^2 & ::= & v \mid \texttt{<}v\texttt{|}\textit{void}\texttt{>} \mid \texttt{<}\textit{void}\texttt{|}v\texttt{>} \\
(\textit{stores}) & \Sigma & ::= & \cdot \mid \Sigma, r = v^2 \mid \Sigma, l \rightarrow \tau^2 \\
(\textit{aspect stores}) & A & ::= & \cdot \mid A, a^2 \\
(\textit{top eval ctxts}) & F & ::= & [] \mid \textbf{store}\ E[\![e]\!]\ \textbf{in}\ e \mid p\texttt{<}F\texttt{>} \mid p\texttt{<}F\texttt{|}e\texttt{>} \\
 & & \mid & p\texttt{<}e\texttt{|}F\texttt{>} \mid E[F] \text{ where } E \neq \textbf{store}\ l[\![v]\!]\ \textbf{in}\ F
\end{array}
$$

Figure 3.15: Syntax of $\mathbb{F}_{HRM2}$

$\boxed{\Gamma; p \vdash_2 e : 1}$

$$\frac{\Gamma; p' \vdash_2 e_1 : 1 \qquad \Gamma; p' \vdash_2 e_2 : 1 \qquad p \in H \qquad p' \in L \qquad \vdash_2 p' \leq p}{\Gamma; p \vdash_2 p'\texttt{<}e_1\texttt{|}e_2\texttt{>} : 1}\ \textsf{c2et:highlow}$$

Figure 3.16: Expression Typing in $\mathbb{F}_{HRM2}$

is the "brackets expression", $p\texttt{<}e_1\texttt{|}e_2\texttt{>}$. Here p is a low-protection label and the e_i are $\mathbb{F}_{HRM}$ expressions. These brackets expressions encapsulate all differences between the two $\mathbb{F}_{HRM}$ expressions that are being simulated. For instance, the $\mathbb{F}_{HRM2}$ expression

```
p<print ‘‘hi’’ | print ‘‘bi’’>;x+3
```

represents the two $\mathbb{F}_{HRM}$ programs

```
p<print ‘‘hi’’>;x+3
```

```
p<print ‘‘bi’’>;x+3
```

The typing rule for the brackets expression in Figure 3.16 requires that the two subexpressions have low protection and release no information into the surrounding high-protection context.

To express the operational semantics of $\mathbb{F}_{HRM2}$ we need to add similar bracket constructs to the $\mathbb{F}_{HRM2}$ grammar in Figure 3.15 for the contents of the reference/label store S and the aspect store A. Simultaneous values stored in references are signified by v^2. In these bracket constructions, the *void* marker indicates that the appropriate element is not present in that half of the program. For example, if advice a is activated in only the left instance but not the right instance of simultaneously executing $\mathbb{F}_{HRM}$ programs, the aspect store of the $\mathbb{F}_{HRM2}$ program that simulates them will contain $\langle a | void \rangle$.

Simultaneous values stored in references are signified by v^2. Labels may only have been created by one of the simultaneously executing expressions, so they are given the type τ^2 in the label store. Finally, advice may only have been instantiated in one of the simultaneously executing expressions, so simultaneous advice a^2 are placed in the aspect store of $\mathbb{F}_{HRM2}$ expressions.

The typing judgments for the modified stores is described in Figure 3.17. Of crucial importance to our noninterference proof is that if a reference, label, or advice has been created or modified by only one of the $\mathbb{F}_{HRM}$ expressions, then the protection domain annotation on that reference, label, or advice must be in the set of low protection domains.

To relate $\mathbb{F}_{HRM}$ to $\mathbb{F}_{HRM2}$, we define the projection function $|\ |_i$ where $i \in \{1, 2\}$ in Figure 3.20. $|p\langle e_1|e_2\rangle|_i$ is $p\langle e_i\rangle$ and $|\ |_i$ is a homomorphism on all other expressions. Since $p\langle e_1|e_2\rangle$ in $\mathbb{F}_{HRM2}$ simulates the simultaneous execution of two low-protection original $\mathbb{F}_{HRM}$ expressions, the projection function extracts one of these two executions.

The $\mathbb{F}_{HRM2}$ machine state $(\Sigma; A; p; e)$ symbolizes the current state of the two simultaneously executing $\mathbb{F}_{HRM}$ programs where the i-th projection is the state of

$\boxed{\Gamma \vdash_2 v \Rightarrow \tau\ \mathsf{ref}_p}$

$$\frac{\Gamma \vdash_2 v : \tau}{\Gamma \vdash_2 v \Rightarrow \tau\ \mathsf{ref}_p}\ \textsf{c2rst:v} \qquad \frac{(\Gamma \vdash_2 v_i : \tau)^{1 \le i \le 2} \qquad p \in L}{\Gamma \vdash_2 \texttt{<}v_1|v_2\texttt{>} \Rightarrow \tau\ \mathsf{ref}_p}\ \textsf{c2rst:vv}$$

$$\frac{\Gamma \vdash_2 v : \tau \qquad p \in L}{\Gamma \vdash_2 \texttt{<}v|void\texttt{>} \Rightarrow \tau\ \mathsf{ref}_p}\ \textsf{c2rst:vvoid} \qquad \frac{\Gamma \vdash_2 v : \tau \qquad p \in L}{\Gamma \vdash_2 \texttt{<}void|v\texttt{>} \Rightarrow \tau\ \mathsf{ref}_p}\ \textsf{c2rst:voidv}$$

$\boxed{\vdash_2 \tau \Rightarrow \tau\ \mathsf{label}_p}$

$$\frac{}{\vdash_2 \tau \Rightarrow \tau\ \mathsf{label}_p}\ \textsf{c2lst:t} \qquad \frac{p \in L}{\vdash_2 \texttt{<}\tau|void\texttt{>} \Rightarrow \tau\ \mathsf{label}_p}\ \textsf{c2lst:tvoid}$$

$$\frac{p \in L}{\vdash_2 \texttt{<}void|\tau\texttt{>} \Rightarrow \tau\ \mathsf{label}_p}\ \textsf{c2lst:voidt}$$

$\boxed{\vdash_2 \Sigma : \Gamma}$

$$\frac{dom(\Sigma) = dom(\Gamma) \qquad \forall r \in dom(\Sigma).\ \Gamma \vdash_2 \Sigma(r) \Rightarrow \Gamma(r) \qquad \forall l \in dom(\Sigma).\ \vdash_2 \Sigma(l) \Rightarrow \Gamma(l)}{\vdash_2 \Sigma : \Gamma}\ \textsf{c2amt:labrefstore}$$

Figure 3.17: Store Typing in $\mathbb{F}_{HRM2}$

$\boxed{\Gamma \vdash_2 a \Rightarrow \mathsf{advice}_p}$

$$\frac{\Gamma \vdash_2 a : \mathsf{advice}_p}{\Gamma \vdash_2 a \Rightarrow \mathsf{advice}_p}\ \mathsf{c2at{:}a} \qquad \frac{\Gamma \vdash_2 a : \mathsf{advice}_p \qquad p \in L}{\Gamma \vdash_2 \texttt{<}a|void\texttt{>} \Rightarrow \mathsf{advice}_p}\ \mathsf{c2at{:}avoid}$$

$$\frac{\Gamma \vdash_2 a : \mathsf{advice}_p \qquad p \in L}{\Gamma \vdash_2 \texttt{<}void|a\texttt{>} \Rightarrow \mathsf{advice}_p}\ \mathsf{c2at{:}voida}$$

$\boxed{\Gamma \vdash_2 A\ \mathbf{ok}}$

$$\frac{}{\Gamma \vdash_2 \cdot\ \mathbf{ok}}\ \mathsf{c2amt{:}aspstoreend}$$

$$\frac{\Gamma \vdash_2 A\ \mathbf{ok} \qquad \Gamma \vdash_2 a^2 \Rightarrow \mathsf{advice}_p}{\Gamma \vdash_2 A, a^2\ \mathbf{ok}}\ \mathsf{c2amt{:}aspstorenext}$$

Figure 3.18: Well-formed Aspect Stores in $\mathbb{F}_{HRM2}$

$\boxed{\vdash_2 (\Sigma; A; p; e)_i\ \mathbf{ok}}$

$$\frac{\begin{array}{c}\vdash_2 \Sigma : \Gamma \qquad \Gamma \vdash_2 A\ \mathbf{ok} \\ \Gamma; p \vdash_2 e : \tau\ \mathbf{for\ some}\ \tau \qquad i \in \{0\} \Rightarrow p \in H \qquad i \in \{1,2\} \Rightarrow p \in L\end{array}}{\vdash_2 (\Sigma; A; p; e)_i\ \mathbf{ok}}\ \mathsf{c2amt{:}ms}$$

Figure 3.19: Well-formed Machine States in $\mathbb{F}_{HRM2}$

$\boxed{|e|_i}$

$|p\texttt{<}e_1 \texttt{|} e_2\texttt{>}|_i = p\texttt{<}e_i\texttt{>}$ $\quad$ $|e|_i$ is a homomorphism on all other expressions

$\boxed{|S|_i}$

$$|\cdot|_i = \cdot$$

$$|\Sigma, r = v|_i = |\Sigma|_i, r = |v|_i \qquad |\Sigma, r = \texttt{<}v_1 \texttt{|} v_2\texttt{>}|_i = |\Sigma|_i, r = v_i$$

$$|\Sigma, r = \texttt{<}v \texttt{|} \mathit{void}\texttt{>}|_1 = |\Sigma|_1, r = v \qquad |\Sigma, r = \texttt{<}v \texttt{|} \mathit{void}\texttt{>}|_2 = |\Sigma|_2$$

$$|\Sigma, r = \texttt{<}\mathit{void} \texttt{|} v\texttt{>}|_1 = |\Sigma|_1 \qquad |\Sigma, r = \texttt{<}\mathit{void} \texttt{|} v\texttt{>}|_2 = |\Sigma|_2, r = v$$

$$|\Sigma, l \to \tau|_i = |\Sigma|_i, l \to \tau \qquad |\Sigma, l \to \texttt{<}\tau \texttt{|} \mathit{void}\texttt{>}|_1 = |\Sigma|_1, l \to \tau$$

$$|\Sigma, l \to \texttt{<}\tau \texttt{|} \mathit{void}\texttt{>}|_2 = |\Sigma|_2 \qquad |\Sigma, l \to \texttt{<}\mathit{void} \texttt{|} \tau\texttt{>}|_1 = |\Sigma|_1$$

$$|\Sigma, l \to \texttt{<}\mathit{void} \texttt{|} \tau\texttt{>}|_2 = |\Sigma|_2, l \to \tau$$

$\boxed{|A|_i}$

$$|\cdot|_i = \cdot \qquad |A, v|_i = |A|_i, |v|_i \qquad |A, \texttt{<}v \texttt{|} \mathit{void}\texttt{>}|_1 = |A|_1, v \qquad |A, \texttt{<}v \texttt{|} \mathit{void}\texttt{>}|_2 = |A|_2$$

$$|A, \texttt{<}\mathit{void} \texttt{|} v\texttt{>}|_1 = |A|_1 \qquad |A, \texttt{<}\mathit{void} \texttt{|} v\texttt{>}|_2 = |A|_2, v$$

Figure 3.20: Projection Functions in $\mathbb{F}_{HRM2}$

the i-th $\mathbb{F}_{HRM}$ program:

$$|(\Sigma; A; p; e)|_i = (|\Sigma|_i; |A|_i; p; |e|_i)$$

The projection function for the reference/label store and the aspect store, also defined in Figure 3.20, is similar to the one for expressions.

Intuitively, the main ideas of the operational semantics of $\mathbb{F}_{HRM2}$ are as follows:

- Ordinary $\mathbb{F}_{HRM}$ expressions embedded within $\mathbb{F}_{HRM2}$ expressions operate as $\mathbb{F}_{HRM}$ expressions normally do. However, the label/reference store and aspect store are accessed through helper functions, defined in Figure 3.21. The new_i function is used to create items in the label/reference store and aspect store. The i annotation is o if the $\mathbb{F}_{HRM}$ expression is on the top level, 1 if the embedded $\mathbb{F}_{HRM}$ is the "left" expression of the simultaneously executing expressions, and 2 if the "right" expression. The $read_i$ and $update_i$ work similarly to access and change, respectively, the label/reference store and aspect store. Finally, Figure 3.22 displays how the helper functions are used by the $\mathbb{F}_{HRM2}$ evaluation rules.

- To evaluate inside the brackets expression $p\langle e_1|e_2\rangle$ in Figure 3.23, the semantics nondeterministically chooses one of e_1 or e_2 to execute. Operations on references or advice executed in e_1 use the "left-hand" component of the reference store or aspect store; operations on references or aspects executed in e_2 use the "right-hand" component of the reference store or aspect store. Finally, to evaluate the expression $p\langle()|()\rangle$, we simply throw away the brackets, returning the unit value (). Since () carries no information, no information is transmitted between low- and high-protection contexts. Whenever values

$\boxed{new_i}$

$$new_o\ v = v \qquad new_1\ v = \texttt{<}v \mid void\texttt{>} \qquad new_2\ v = \texttt{<}void \mid v\texttt{>}$$

$\boxed{update_i}$

$$update_o\ v\ v' = v' \qquad update_1\ v\ v' = \texttt{<}v' \mid |v|_2\texttt{>} \qquad update_2\ v\ v' = \texttt{<}|v|_1 \mid v'\texttt{>}$$

$\boxed{read_i}$

$$read_o\ v = v \qquad read_1\ v = |v|_1 \qquad read_2\ v = |v|_2$$

$\boxed{\mathcal{A}[\![A]\!]_{l[\![v]\!]} = e}$

$$\frac{}{\mathcal{A}[\![\cdot]\!]_{l[\![v]\!]} = ()}\ \textsf{c2ac:end} \qquad \frac{l \in \{\bar{l}\}_{p'} \qquad \mathcal{A}[\![A]\!]_{l[\![v]\!]} = e}{\mathcal{A}[\![\{\{\bar{l}\}_{p'}(x) \rhd_p e'\}, A]\!]_{l[\![v]\!]} = p\texttt{<}e'\texttt{>}[v/x]; e}\ \textsf{c2ac:match}$$

$$\frac{l \in \{\bar{l}\}_{p'} \qquad \mathcal{A}[\![A]\!]_{l[\![v]\!]} = e}{\mathcal{A}[\![\texttt{<}\{\{\bar{l}\}_{p'}(x) \rhd_p e'\} \mid void\texttt{>}, A]\!]_{l[\![v]\!]} = p\texttt{<}e' \mid ()\texttt{>}[v/x]; e}\ \textsf{c2ac:match1}$$

$$\frac{l \in \{\bar{l}\}_{p'} \qquad \mathcal{A}[\![A]\!]_{l[\![v]\!]} = e}{\mathcal{A}[\![\texttt{<}void \mid \{\{\bar{l}\}_{p'}(x) \rhd_p e'\}\texttt{>}, A]\!]_{l[\![v]\!]} = p\texttt{<}() \mid e'\texttt{>}[v/x]; e}\ \textsf{c2ac:match2}$$

$$\frac{l \notin \{\bar{l}\}_{p'} \qquad \mathcal{A}[\![A]\!]_{l[\![v]\!]} = e}{\mathcal{A}[\![\{\{\bar{l}\}_{p'}(x) \rhd_p e'\}, A]\!]_{l[\![v]\!]} = e}\ \textsf{c2ac:nomatch}$$

$$\frac{l \notin \{\bar{l}\}_{p'} \qquad \mathcal{A}[\![A]\!]_{l[\![v]\!]} = e}{\mathcal{A}[\![\texttt{<}\{\{\bar{l}\}_{p'}(x) \rhd_p e'\} \mid void\texttt{>}, A]\!]_{l[\![v]\!]} = e}\ \textsf{c2ac:nomatch1}$$

$$\frac{l \notin \{\bar{l}\}_{p'} \qquad \mathcal{A}[\![A]\!]_{l[\![v]\!]} = e}{\mathcal{A}[\![\texttt{<}void \mid \{\{\bar{l}\}_{p'}(x) \rhd_p e'\}\texttt{>}, A]\!]_{l[\![v]\!]} = e}\ \textsf{c2ac:nomatch2}$$

Figure 3.21: Helper Functions for $\mathbb{F}_{HRM2}$

$\boxed{(\Sigma; A; p; e)_i \longmapsto_{2,\beta} (\Sigma'; A'; p; e')_i}$

$$\frac{\vdash_2 p' \leq p}{(\Sigma; A; p; \Uparrow \{v(x) \rhd_{p'} e_1\})_i \longmapsto_{2,\beta} (\Sigma; (A, new_i\ \{v(x) \rhd_{p'} e_1\}); p; ())_i}\ \text{c2eb:advinst}$$

$$\frac{l \notin \Sigma \qquad \vdash_2 p' \leq p}{(\Sigma; A; p; \mathbf{new}_{p'}\ \tau)_i \longmapsto_{2,\beta} ((\Sigma, l = (new_i\ \tau, p')); A; p; l)_i}\ \text{c2eb:new}$$

$$\frac{read_i\ \Sigma(l) \neq void \qquad \mathcal{A}[\![|A|_i]\!]_{l[\![v]\!]} = e}{(\Sigma; A; p; l[\![v]\!])_i \longmapsto_{\beta} (\Sigma; A; p; e)_i}\ \text{c2eb:jp}$$

$$\frac{r \notin \Sigma \qquad \vdash_2 p' \leq p}{(\Sigma; A; p; \mathbf{ref}_{p'}\ v)_i \longmapsto_{2,\beta} ((\Sigma, r = (new_i\ v, p')); A; p; r)_i}\ \text{c2eb:ref}$$

$$\frac{v' = read_i\ \Sigma(r) \qquad v' \neq void}{(\Sigma; A; p; !r)_i \longmapsto_{2,\beta} (\Sigma; A; p; v')_i}\ \text{c2eb:deref}$$

$$\frac{\Sigma(r) = (v^2, p') \qquad \vdash_2 p' \leq p}{(\Sigma; A; p; r\ :=\ v)_i \longmapsto_{2,\beta} ((\Sigma, r = (update_i\ v^2\ v, p')); A; p; v)_i}\ \text{c2eb:assign}$$

$$\frac{\vdash_2 p' \leq p}{(\Sigma; A; p; p'\texttt{<}()\,\texttt{|}\,()\texttt{>})_i \longmapsto_{2,\beta} (\Sigma; A; p; ())_i}\ \text{c2eb:highlow}$$

Figure 3.22: β-redex Operational Semantics for $\mathbb{F}_{HRM2}$

$\boxed{(\Sigma; A; p; e) \longmapsto_2 (\Sigma'; A'; p; e')}$

$$\frac{(\Sigma; A; P'; e)_i \longmapsto_2 (\Sigma'; A'; P'; e')_i \qquad \vdash_2 P' \leq P \qquad P \in H \equiv P' \in H}{(\Sigma; A; P; P'\texttt{<}e\texttt{>})_i \longmapsto_2 (\Sigma'; A'; P; P'\texttt{<}e'\texttt{>})_i}\ \text{c2e:low}$$

$$\frac{(\Sigma; A; P'; e_i)_i \longmapsto_2 (\Sigma'; A'; P'; e'_i)_i \qquad e_j = e'_j \qquad \{i, j\} = \{1, 2\}}{(\Sigma; A; P; P'\texttt{<}e_1\,\texttt{|}\,e_2\texttt{>}) \longmapsto_2 (\Sigma'; A'; P; P'\texttt{<}e'_1\,\texttt{|}\,e'_2\texttt{>})}\ \text{c2e:brack}$$

$$\frac{P \in H \qquad P' \in L}{(\Sigma; A; P; P'\texttt{<}e\texttt{>}) \longmapsto_2 (\Sigma; A; P; P'\texttt{<}|e|_1\,\texttt{|}\,|e|_2\texttt{>})}\ \text{c2e:highlow}$$

Figure 3.23: Operational Semantics for $\mathbb{F}_{HRM2}$

$\boxed{\mathcal{S}(F)}$

$$\ldots \qquad \mathcal{S}(p\langle F \mid e\rangle) = \mathcal{S}(F) \qquad\qquad \mathcal{S}(p\langle e \mid F\rangle) = \mathcal{S}(F)$$

$\boxed{(\Sigma; A; p; e) \longmapsto_{top2} (\Sigma'; A'; p; e')}$

$$\frac{(\Sigma; A; p; e) \longmapsto (\Sigma'; A'; p; e')}{(\Sigma; A; p; e) \longmapsto_{top,2} (\Sigma'; A'; p; e')}\ \textsf{c2evt:ce}$$

$$\frac{}{(\Sigma; A; p; F[\mathbf{stack}()]) \longmapsto_{top,2} (\Sigma; A; p; F[\mathcal{S}(F)])}\ \textsf{c2evt:stack}$$

Figure 3.24: Top Operational Semantics for $\mathbb{F}_{HRM2}$

v_1 and v_2 are not both (), the expression $p\langle v_1|v_2\rangle$ is stuck. Fortunately, such an expression is ill-typed and never arises from evaluation of a well-typed program.

- The judgment form for execution of top-level $\mathbb{F}_{HRM2}$ expressions in Figure 3.24 has the same shape as the judgment form for $\mathbb{F}_{HRM}$ expressions. The stack function $\mathcal{S}(F)$ behaves similarly for the brackets expression, $p\langle e_1 \mid e_2\rangle$, as it does for $p\langle e\rangle$.

Relating $\mathbb{F}_{HRM}$ and $\mathbb{F}_{HRM2}$

Once $\mathbb{F}_{HRM2}$ has been defined, it is necessary to show that it accurately simulates two $\mathbb{F}_{HRM}$ programs. Two theorems, one concerning the *soundness* of $\mathbb{F}_{HRM2}$ execution relative to $\mathbb{F}_{HRM}$ and the other concerning the *completeness* of $\mathbb{F}_{HRM2}$ relative to $\mathbb{F}_{HRM}$ help to establish the proper correspondence.

The soundness theorem states that if a $\mathbb{F}_{HRM2}$ expression takes a step, then the two corresponding $\mathbb{F}_{HRM}$ programs (the projections of the $\mathbb{F}_{HRM2}$ expression) must

each take the same respective steps. The proof of this theorem requires, among other things, an auxiliary lemma that establishes a soundness result for aspect composition.

Lemma 3.2.1 (Expression Soundness Lemma) *For* $i \in \{1, 2\}$,
if $(\Sigma; A; p; e)_i \longmapsto_{2,top} (\Sigma'; A'; p; e')_i$,
then $(|\Sigma|_i; |A|_i; p; e) \longmapsto_{top} (|\Sigma'|_i; |A'|_i; p; e')$.

Proof: By induction on the structure of $(\Sigma; A; p'; e)_i \longmapsto_{2,top} (\Sigma'; A'; p; e')_i$. □

Lemma 3.2.2 (Stack Soundness Lemma) *For* $i \in \{1, 2\}$, $\mathcal{S}(|F|_i) = |\mathcal{S}(F)|_i$.

Proof: By induction on the structure of $\mathcal{S}(F)$. □

Lemma 3.2.3 (Aspect Composition Soundness Lemma) *For* $i \in \{1, 2\}$, *if* $\mathcal{A}[\![A]\!]_{l[\![v]\!]} = e$,
then $\mathcal{A}[\![|A|_i]\!]_{l[\![|v|_i]\!]} = |e|_i$.

Proof: By induction on the structure of the definition of $\mathcal{A}[\![A]\!]_{l[\![v]\!]} = e$. □

Theorem 3.2.4 (Soundness) *For* $i \in \{1, 2\}$, *if* $(\Sigma; A; p; e) \longmapsto^*_{2,top} (\Sigma'; A'; p; e')$
then $|(\Sigma; A; p; e)|_i \longmapsto^*_{top} |(\Sigma'; A'; p; e')|_i$

Proof: By induction on the structure of the operational judgment $(\Sigma; A; p; e) \longmapsto^*_{2,top} (\Sigma'; A'; p; e')$, with use of the Soundness Lemma 3.2.1.

- Case c2evt:stack uses the Stack Soundness Lemma 3.2.2.
- Case c2eb:jp uses the Aspect Composition Soundness Lemma 3.2.3.

□

The completeness theorem states that if two $\mathbb{F}_{HRM}$ programs step to values, then the representation in $\mathbb{F}_{HRM2}$ that simulates them simultaneously must step to a value. The completeness theorem requires an auxiliary lemma stating that a $\mathbb{F}_{HRM2}$ program is only stuck when one of its corresponding $\mathbb{F}_{HRM}$ programs are stuck.

Lemma 3.2.5 (Completeness Stuck Lemma) *Assume* $(\Sigma; A; p; e)$ *is stuck. Then* $|(\Sigma; A; p; e)|_i$ *is stuck for some* $i \in \{1, 2\}$.

Proof: Proof by induction on the structure of e. □

Theorem 3.2.6 (Completeness) *Assume* $|(\Sigma; A; p; e)|_i \longmapsto^*_{top} (\Sigma'_i; A'_i; p; v_i)$ *for all* $i \in 1, 2$ *then there exists* $(\Sigma'; A'; p; v)$ *such that* $(\Sigma; A; p; e) \longmapsto^*_{2,top} (\Sigma'; A'; p; v)$

Proof: If (Σ, A, p, e) yields an infinite reduction sequence, then $|(\Sigma, A, p, e)|_i$ yields an infinite reduction sequence by the Soundness Theorem 3.2.4.
If $(\Sigma; A; p; e)$ is stuck, then $|(\Sigma; A; p; e)|_i$ is stuck by the Completeness Stuck Lemma 3.2.5.
Therefore, $(\Sigma; A; p; e)$ reduces to a successful configuration. □

Safety of $\mathbb{F}_{HRM2}$

To continue we prove that the type system of $\mathbb{F}_{HRM2}$ is sound with respect to our operational semantics using Progress and Preservation theorems. This strategy requires that we extend the typing relation to cover all of the run-time terms in the language as well as the other elements of the abstract machine (*i.e.,* the code store

and aspect store). A $\mathbb{F}_{HRM2}$ configuration $(\Sigma; A; p; e)$ is well-typed if it satisfies the judgment $\vdash_2 (\Sigma; A; p; e)$ **ok** in Figure 3.19. If the stores and the expression contain brackets, the protection domains associated with the brackets must be low.

Part of the proof of Progress involves defining the canonical forms of each type. It is important to note that the brackets expression is not a value and therefore the only values with type bool, for instance, are **true** and **false**. This fact comes into play later in the noninterference proof.

The following lemma gives the rest of canonical forms.

Lemma 3.2.7 (Canonical Forms) *Suppose* $\cdot \vdash_2 \ v : \tau$ *is a closed, well-formed value.*

- *If* $\tau = 1$, *then* $v = ()$.
- *If* $\tau = \mathsf{string}$, *then* $v = \sigma$.
- *If* $\tau = \mathsf{bool}$, *then* $v =$ **true** *or* **false**.
- *If* $\tau = \tau_1 \times ... \times \tau_n$, *then* $v = (\overline{v})$.
- *If* $\tau = \tau_1 \rightarrow_p \tau_2$, *then* $v = \lambda_p x : \tau_1.e$.
- *If* $\tau = \mathsf{advice}_p$, *then* $v = \{v(x) \rhd_p e\}$.
- *If* $\tau = \tau\ \mathsf{label}_p$, *then* $v = l$.
- *If* $\tau = \tau\ \mathsf{ref}_p$, *then* $v = r$.
- *If* $\tau = \tau\ \mathsf{pc}_p$, *then* $v = \{\bar{l}\}_p$.
- *If* $\tau = \mathsf{stack}$, *then* $v = \cdot$ *or* $l[\![v_1]\!] :: v_2$.

Proof: By induction on the structure of $\Gamma \vdash_2 v : \tau$, using the fact that v is a value. □

We now state the standard Progress and Preservation lemmas.

Lemma 3.2.8 (Γ to Σ) • *If* $\vdash_2 \Sigma : \Gamma$ *and* $\Gamma(l) = \tau$ label_p, *then* $\Sigma(l) = \tau^2$.

• *If* $\vdash_2 \Sigma : \Gamma$ *and* $\Gamma(r) = \tau$ ref_p, *then* $\Sigma(r) = v^2$ *for some* v^2.

Proof: Trivial using Rule c2amt:labrefstore. □

Lemma 3.2.9 (Reverse Evaluation Contexts) *If* $\Gamma; p \vdash_2 E[e] : \tau$, *then* $\Gamma; p \vdash_2 e : \tau'$ *for some* τ'.

Proof: By induction on the structure of evaluation contexts E. □

Lemma 3.2.10 (Total Aspect Store) *If* $\vdash_2 \Sigma : \Gamma$ *and* $\Gamma \vdash_2 A$ *ok*, *then* $\mathcal{A}[\![A]\!]_{l[\![v]\!]} = e$ *is a total function.*

Proof: By induction on the structure of the judgment $\mathcal{A}[\![A]\!]_{l[\![v]\!]} = e$. □

Lemma 3.2.11 (Progress Lemma) *If* $\vdash_2 \Sigma : \Gamma$ *and* $\Gamma \vdash_2 A$ *ok and* $\Gamma; p \vdash_2 e : \tau$ *and*

• *If* $i \in \{o\}$, *then* $p \in H$.

• *If* $i \in \{1, 2\}$, *then* $p \in L$.

then either e *is a value or* $(\Sigma'; A'; p; e')$ *such that* $(\Sigma; A; p; e) \longmapsto_{2,top} (\Sigma'; A'; p; e')$.

Proof: By induction on the structure of the typing judgment $\Gamma; p \vdash_2 e : \tau$ with the Canonical Forms Lemma 3.2.7 and the Reverse Evaluation Context Lemma 3.2.9.

- Cases cet:jp, cet:asgn, and cet:deref use the Γ to Σ Lemma 3.2.8.
- Case cet:jp uses the Total Aspect Store Lemma 3.2.10.

□

Theorem 3.2.12 (Progress) *If* $\vdash_2 (\Sigma; A; p; e)$ ok
then either e *is a value, or there exists* $(\Sigma'; A'; p; e')$ *such that* $(\Sigma; A; p; e) \longmapsto_{2,top} (\Sigma'; A'; p; e')$.

Proof: Straightforward use of the Progress Lemma 3.2.11. □

Lemma 3.2.13 (Value Substitution) *If* $\Gamma, x : \tau_2 \vdash_2 v_1 : \tau_1$ *and* $\Gamma \vdash_2 v_2 : \tau_2$,
then $\Gamma \vdash_2 v_1[v_2/x] : \tau_1$.

Proof: By induction on the structure of typing rule $\Gamma \vdash_2 v : \tau$ with the Expression Substitution Lemma 3.2.14. □

Lemma 3.2.14 (Expression Substitution) *If* $\Gamma, x : \tau_2; p \vdash_2 e : \tau_1$ *and* $\Gamma \vdash_2 v : \tau_2$,
then $\Gamma; p \vdash_2 e[v/x] : \tau_1$.

Proof: By induction on the structure of typing rule $\Gamma; p \vdash_2 e : \tau$ with the Value Substitution Lemma 3.2.13. □

Lemma 3.2.15 (Aspect Store Weakening) *If* $\Gamma \vdash_2 A$ *ok, then*

- $\Gamma, l : \tau\ \mathsf{label}_p \vdash_2 A\ ok$ *for some* l, τ, p
- $\Gamma, r : \tau\ \mathsf{ref}_p \vdash_2 A\ ok$ *for some* r, τ, p

Proof: By induction on the structure of typing rule $\Gamma \vdash_2 Aok$. □

Lemma 3.2.16 (New Advice Preservation) *If* $\Gamma \vdash \{v.x \Rightarrow_{p'} e\} : \mathsf{advice}_p$ *and (if* $i \in \{1,2\}$ *then* $p \in L$*) and* $\vdash p' \leq p$
then $\Gamma \vdash new_i\{v.x \Rightarrow_{p'} e\} \Rightarrow \mathsf{advice}_{p'}$.

Proof: Straightforward using the typing for the judgment $\Gamma \vdash new_i\{v.x \Rightarrow_{p'} e\} \Rightarrow \mathsf{advice}_{p'}$. □

Lemma 3.2.17 (New Label Preservation) *If* $\vdash p' \leq p$ *and (if* $i \in \{1,2\}$ *then* $p \in L$*) then* $\vdash (new_i\tau, p') \Rightarrow \tau\ \mathsf{label}_{p'}$.

Proof: Straightforward using the typing rules for the judgment $\vdash (new_i\tau, p') \Rightarrow \tau\ \mathsf{label}_{p'}$. □

Lemma 3.2.18 (New Reference Preservation) *If* $\Gamma \vdash v : \tau$ *and* $\vdash p' \leq p$ *and (if* $i \in \{1,2\}$ *then* $p \in L$*)*
then $\vdash (new_i v, p') \Rightarrow \tau\ \mathsf{ref}_{p'}$.

Proof: Straightforward using the typing rules for the judgment $\vdash (v^2, p') \Rightarrow \tau\ \mathsf{ref}_{p'}$. □

Lemma 3.2.19 (Read Reference Preservation) *If* $\vdash (v^2, p') \Rightarrow \tau\ \mathsf{ref}_{p'}$ *and* $\vdash p \leq p'$ *and (if* $i \in \{1,2\}$ *then* $p \in L$*),*
then $\Gamma \vdash read_i v^2 : \tau$.

Proof: Straightforward use of the typing rules for the judgment $\vdash (v^2, p') \Rightarrow \tau\ \mathsf{ref}_{p'}$. □

Lemma 3.2.20 (Assign Reference Preservation) *If* $\Gamma \vdash v : \tau$ *and* $\vdash p' \leq p$ *and (if* $i \in \{1,2\}$ *then* $p \in L$*) and* $\vdash (v^2, p') \Rightarrow \tau\ \mathsf{ref}_{p'}$
then $\Gamma \vdash (update_i v^2 v, p') \Rightarrow \tau\ \mathsf{ref}_{p'}$.

Proof: Straightforward using the typing rules for the judgment $\vdash (v^2, p') \Rightarrow \tau\ \mathsf{ref}_{p'}$. □

Lemma 3.2.21 (Stack Case Pattern Preservation Lemma)
If $\Gamma; p \vdash v : stack$ *and* $\Gamma \dashv pat \vdash \Gamma'$ *and* $G, G'; p \vdash e_1 : \tau$ *and* $v \vdash pat \Rightarrow sub$
then $\Gamma; p \vdash sub(e_1) : \tau$.

Proof: By induction on the typing rule $\Gamma \dashv pat \vdash \Gamma'$. □

Lemma 3.2.22 (Stack Case Preservation) *If* $\Gamma; p \vdash v : stack$ *and* $v \vdash pat \Rightarrow \Gamma'$ *and* $G, G'; p \vdash e_1 : \tau$ *and* $v \vdash pat \Rightarrow sub$
then $\Gamma; p \vdash sub(e_1) : \tau$.

Proof: Straightforward use of the Stack Case Pattern Preservation Lemma 3.2.21. □

Lemma 3.2.23 (Advice Composition Preservation) *For* $i \in \{o, 1, 2\}$, *if* $\vdash_2 \Sigma : \Gamma$ *and* $\Gamma \vdash_2 A\ \mathsf{ok}$ *and* $\mathcal{A}[\![|A|_i]\!]_{l[\![v]\!]} = e$ *and* $\Gamma \vdash_2 l : \tau\ \mathsf{label}_{p'}$ *and* $\Gamma \vdash_2 v : \tau$ *and* $p' \leq p$,
then $\Gamma; p \vdash_2 e : \tau$.

Proof: By induction on the structure of the typing judgment $\mathcal{A}[\![|A|_i]\!]_{l[\![v]\!]} = e$.

□

Lemma 3.2.24 (β-redex Preservation) *If* $\vdash_2 \Sigma : \Gamma$ *and* $\Gamma \vdash_2 A\ ok$ *and* $\Gamma; p \vdash_2 e : \tau$ *and* $(\Sigma; A; p; e) \longmapsto_{2,\beta} (\Sigma'; A'; p; e')$,

then

- $\vdash_2 \Sigma' : \Gamma'$
- $\Gamma' \vdash_2 A' ok$
- $\Gamma'; p \vdash_2 e' : \tau$.

Proof: By induction on the structure of the operational rules $(\Sigma; A; p; e) \longmapsto_{2,\beta} (\Sigma'; A'; p; e')$ with the Value Substitution Lemma 3.2.13, the Expression Substitution Lemma 3.2.14, the Aspect Store Weakening Lemma 3.2.15.

- Case c2eb:advinst uses the New Advice Preservation Lemma 3.2.16.
- Case c2eb:new uses the New Label Preservation Lemma 3.2.17.
- Case c2eb:ref uses New Reference Preservation Lemma 3.2.18.
- Case c2eb:deref uses the Read Reference Preservation Lemma 3.2.19.
- Case c2eb:assign uses the Assign Reference Preservation Lemma 3.2.20.
- Case c2eb:jp uses the Advice Composition Preservation Lemma 3.2.23.
- Case ceb:scaseyes uses the Stack Case Preservation Lemma 3.2.22.

□

Lemma 3.2.25 (Evaluation Context Preservation) *If* $\Gamma; p \vdash_2 E[e] : \tau$ *and* $\Gamma; p \vdash_2 e : \tau'$ *and* $\Gamma; p \vdash_2 e' : \tau'$,
then $\Gamma; p \vdash_2 E[e'] : \tau$

Proof: By induction on the structure of evaluation contexts E. □

Lemma 3.2.26 (Projection Typing) *If* $\Gamma; p \vdash_2 e : \tau$,
then $\Gamma; p \vdash_2 |e|_i : \tau$ *for all* $i \in \{1, 2\}$

Proof: By induction on the structure of the typing rules $\Gamma; p \vdash_2 e : \tau$. □

Lemma 3.2.27 (Preservation Lemma) *If* $\vdash_2 \Sigma : \Gamma$ *and* $\Gamma \vdash_2 A\ ok$ *and* $\Gamma; p \vdash_2 e : \tau$ *and* $(\Sigma; A; p; e) \longmapsto_2 (\Sigma'; A'; p; e')$, *then*

- $\vdash_2 \Sigma' : \Gamma'$
- $\Gamma' \vdash_2 A' ok$
- $\Gamma'; p \vdash_2 e' : \tau$.

Proof: By induction on the structure of the operational rules $(\Sigma; A; p; e) \longmapsto_2 (\Sigma'; A'; p; e')$.

- Case ce:beta uses the β-redex Preservation Lemma 3.2.24.
- Case ce:eval uses the Reverse Evaluation Context Lemma 3.2.9 and the Evaluation Context Preservation 3.2.25.
- Case c2e:highlow uses Projection Typing Lemma 3.2.26.

□

Lemma 3.2.28 (Top Preservation Lemma) *If* $\vdash_2 \Sigma : \Gamma$ *and* $\Gamma \vdash_2 A\ ok$ *and* $\Gamma; p \vdash_2 e : \tau$ *and* $(\Sigma; A; p; e) \longmapsto_{2,top} (\Sigma'; A'; p; e')$, *then*

- $F' \vdash_2 \Sigma' : \Gamma'$
- $\Gamma' \vdash_2 A'ok$
- $\Gamma'; p \vdash_2 e' : \tau$.

Proof: By induction on the structure of the operational rule $(\Sigma; A; p; e) \longmapsto_{2,top} (\Sigma'; A'; p; e')$.

- Case cevt:ce uses the Preservation Lemma 3.2.27.

□

Theorem 3.2.29 (Preservation) *If* $\vdash_2 (\Sigma; A; p; e)$ ok *and* $(\Sigma; A; p; e) \longmapsto_{2,top} (\Sigma'; A'; p; e')$
then $\vdash_2 (\Sigma'; A'; p; e')$ ok.

Proof: Straightforward use of the Top Preservation Lemma 3.2.27. □

Well-typed $\mathbb{F}_{HRM2}$ programs produce indistinguishable $\mathbb{F}_{HRM}$ results

Most of the difficult work has now been done. We merely need to apply lemmas and theorems we have already proven to get our first powerful result: If a high-protection $\mathbb{F}_{HRM2}$ expression steps to a boolean value, then the corresponding $\mathbb{F}_{HRM}$ projections (which differ only in low protection code) step to equal values. In other words, no low-production code (be it aspect-oriented features or otherwise) has influenced execution of high-protection expressions.

Lemma 3.2.30 (Equivalent Execution in $\mathbb{F}_{HRM2}$)
If $\mathbf{HIGH} \in H$ *and* $\cdot; \mathbf{HIGH} \vdash_2 e : \mathsf{bool}$ *and*
$(\cdot; \cdot; \mathbf{HIGH}; e) \longmapsto^*_{2,top} (\Sigma; A; \mathbf{HIGH}; v)$
then $|v|_1 = |v|_2$.

Proof: By the Preservation Theorem 3.2.29, $\vdash_2 \Sigma : \Gamma$ and $\Gamma \vdash_2 v : \mathsf{bool}$.
By the Canonical Forms Lemma 3.2.7, v is either **true** or **false**.
$|\mathbf{true}|_1 = |\mathbf{true}|_2$ and $|\mathbf{false}|_1 = |\mathbf{false}|_2$. □

Putting it all together: Noninterference

Finally, for the noninterference proof, we start with a high-protection $\mathbb{F}_{HRM}$ expression e that has a free variable x. We add a low-protection expression **LOW**<e'> where $\mathbf{LOW} \in L$ so that e with the low-protection code and e alone are executed simultaneously and their resulting values compared. This is achieved by constructing the $\mathbb{F}_{HRM2}$ expression $e[\mathbf{LOW}\langle e' | () \rangle / x]$ where the right projection steps to e with () substituted for x and the left projection is the low-protection code e' substituted for x in e. Using the soundness, completeness, and preservation theorems, we show that both e alone and e with the added low-protection code step to the same value. Therefore the low-protection code, even if it introduced advice, did not interfere with execution.

Theorem 3.2.31 (Noninterference) *If* $\mathbf{HIGH} \in H$ *and* $\mathbf{LOW} \in L$ *and* $\vdash \mathbf{LOW} \leq \mathbf{HIGH}$ *and* e *is a core language expression where* $x; \mathbf{HIGH} \vdash e : \mathsf{bool}$ *and* $\cdot; \mathbf{LOW} \vdash e' : 1$ *and* $(\cdot; \cdot; \mathbf{HIGH}; e[\mathbf{LOW}\langle e' \rangle / x]) \longmapsto^*_{top} (\Sigma_1; A_1; \mathbf{HIGH}; v_1)$ *and* $(\cdot; \cdot; \mathbf{HIGH}; e[\mathbf{LOW}\langle () \rangle / x]) \longmapsto^*_{top} (\Sigma_2; A_2; \mathbf{HIGH}; v_2)$
then $v_1 = v_2$.

Proof: Construct the $\mathbb{F}_{HRM2}$ expression $\mathbf{LOW}\langle e' \mid ()\rangle; e$, such that $|\mathbf{LOW}\langle e' \mid ()\rangle; e|_1 = \mathbf{LOW}\langle e'\rangle; e$ and $|\mathbf{LOW}\langle e' \mid ()\rangle; e|_2 = \mathbf{LOW}\langle()\rangle; e$.

Therefore, $|(\cdot;\cdot;\mathbf{HIGH};\mathbf{LOW}\langle e' \mid ()\rangle; e)|_1 \longmapsto^*_{2,top} (\Sigma_1; A_1; \mathbf{HIGH}; v_1)$ and $|(\cdot;\cdot;\mathbf{HIGH};\mathbf{LOW}\langle e' \mid ()\rangle; e)|_2 \longmapsto^*_{2,top} (\Sigma_2; A_2; \mathbf{HIGH}; v_2)$.

By the Completeness Theorem 3.2.6, $(\cdot;\cdot;\mathbf{HIGH};\mathbf{LOW}\langle e' \mid ()\rangle; e) \longmapsto^*_{2,top} (\Sigma; A; \mathbf{HIGH}; v)$ for some Σ, A, and v.

By the Soundness Theorem 3.2.4, for $i \in \{1, 2\}$, $|(\cdot;\cdot;\mathbf{HIGH};\mathbf{LOW}\langle e' \mid ()\rangle; e)|_i \longmapsto^*_{2,top} |(\Sigma; A; \mathbf{HIGH}; v)|_i$.

Therefore, $|v|_1 = v_1$ and $|v|_2 = v_2$.

By the Equivalent Execution Lemma 3.2.30, $|v|_1 = |v|_2$.

Therefore, $v_1 = v_2$. □

3.3 Harmless Source Language: HarmlessAML

Our core calculus will not serve as an effective source-level aspect-oriented programming language – it is far too low level for convenient programming. However, the intended purpose of the core calculus is not to serve as a user-friendly programming language itself, but rather to serve as a semantic intermediate language to which we compile a more practical source language.

As we have already shown, it is possible to prove deep properties of the core, including our powerful noninterference result. The primary reason for this is that $\mathbb{F}_{HRM}$ consists of a relatively simple, orthogonal collection of primitive operators. In a more convenient source language, these simple operators are combined together to form complex, higher-level primitives. To obtain properties of a source language, we give a type-preserving translation into the core and then exploit properties of

$$
\begin{array}{llll}
(\textit{types}) & t & ::= & \mathsf{1} \mid \mathsf{string} \mid \mathsf{bool} \mid t \rightarrow_p t \mid t\ \mathsf{ref}_p \mid \mathsf{stack} \\
(\textit{values}) & v & ::= & () \mid s \mid \texttt{true} \mid \texttt{false} \\
(\textit{expressions}) & e & ::= & v \mid x \mid e; e \mid \texttt{print}\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & & \mid & \texttt{let}\ ds\ \texttt{in}\ e \mid e\ e \mid \texttt{!}e \mid e\texttt{:=}e \\
 & & \mid & \texttt{stkcase}\ e\ (pat\ \texttt{=>}\ e \mid \texttt{\_ =>}\ e) \\
(\textit{frame patterns}) & \texttt{fpat} & ::= & \texttt{\_} \mid \texttt{(|\#}\bar{f}\texttt{\#|)}\ (x, n) \\
(\textit{stack patterns}) & pat & ::= & \texttt{[]} \mid \texttt{fpat} :: pat \mid x \\
(\textit{main decl}) & d & ::= & \texttt{string}\ x\ \texttt{=}\ e \mid \texttt{bool}\ x\ \texttt{=}\ e \mid \texttt{ref}\ x\ \texttt{=}\ e \\
 & & \mid & \texttt{fun}\ f\ (x\texttt{:}\, t_1) : t_2\ \texttt{=}\ e \\
(\textit{main decls}) & ds & ::= & . \mid d\ ds \\
(\textit{time}) & tm & ::= & \texttt{before} \mid \texttt{after} \\
(\textit{advice decl}) & a & ::= & \texttt{advice}\ tm\ \texttt{(|\#}\bar{f}\texttt{\#|)}\ (x, s, n)\ \texttt{=}\ e \\
(\textit{advice decls}) & as & ::= & . \mid d\ as \mid a\ as \\
(\textit{aspects}) & asps & ::= & . \mid p : \{as\}\ asps \\
(\textit{programs}) & prog & ::= & ds\ asps\ e
\end{array}
$$

Figure 3.25: Syntax of HarmlessAML

type-correct core calculus terms. This strategy effectively modularizes proofs of properties about the source and greatly simplifies the overall proof.

Hence, in this section, we proceed to define an aspect-oriented source language, HarmlessAML, with harmless advice. We have implemented the language in Standard ML and explored the extent to which we can use harmless aspects to implement dynamic security policies.

3.3.1 Syntax

Figure 3.25 presents the formal syntax of the source language, HarmlessAML.

Most of HarmlessAML expressions and values mimic $\mathbb{F}_{HRM}$ expressions and values, although there are a few differences. For instance, none of the run-time-only values such as labels, reference locations, or stack values need appear in the collection of source values, as HarmlessAML is not executed directly. Also, for

convenience, we allow a local let declaration in expressions, which programmers can use to allocate values with basic types, reference type, or function type. Note that we use the meta-variable f to stand for program variables bound to functions. We use the meta-variable x to stand for any kind of program variable.

HarmlessAML `stkcase` expressions analyze stack values in a similar way to the target, only the patterns are slightly different, reflecting a particular compilation strategy. More specifically, when compiling a function, we will automatically allocate the following items on the stack: a label corresponding to the function and a tuple containing a pointer to the function argument and a string corresponding to the name of the function that was called. Consequently, the patterns that match stack frames have the form `(|#`$\bar{f}$`#|)` (x, n), where $\bar{f}$ is checked against the label, and x and n are bound to the argument and the function name respectively. The function name can be used when printing out debugging information, profiling information, etc.

Advice in HarmlessAML (`advice` tm `(|#`$\bar{f}$`#|)` (x, s, n) `=` e) is either before advice that runs before a function call or after advice that runs after a function call. Around advice, which replaces the function body, is not harmless, and as such is not represented in HarmlessAML. Similar to the HarmlessAML stack patterns, when the advice is triggered, x is bound to the function argument, and n is bound to a string corresponding to the function name. The variable s is bound to the stack at the point the advice is triggered. In the source language, programmers do not explicitly allocate their own data on the stack, nor do they explicitly grab the current stack. Code for performing these actions is emitted at specific points during the translation from HarmlessAML into $\mathbb{F}_{HRM2}$.

```
fun openA (x : string) : file = ...
fun openR (x : string) : file = ...
fun openO (x : string) : file = ...
fun openC (x : string) : file = ...
fun exists (x : string) : bool = ...
fun openW (x:string):file =
  if exists x
  then openO x
  else openC x
fun read (x : file * int)  : string = ...
fun write (x : file * string) : int = ...
...
```

Figure 3.26: File I/O Library

The main syntactic difference between HarmlessAML and AspectML is that HarmlessAML programs (*ds asps e*) preserve a distinct syntactic separation between main program declarations *ds*, aspect declarations *asps*, and the main program expression *e*. In Section 3.3.4, we will show how this syntactic separation, combined with the noninterference properties of $\mathbb{F}_{HRM}$, can be used to prove that programs written in the HarmlessAML are harmless.

3.3.2 Assorted Security Examples

Figure 3.26 and 3.27 display HarmlessAML example code.[2] Figure 3.26 presents the (partial) definition of a file I/O library, which implements a number of file operations. We have shown some of the file operations here.

[2]The examples in this section use a number of additional standard operations, such as string and integer manipulations, and file and network I/O operations, that we have not formalized, but we have implemented in our system.

Figure 3.27 presents three simple security policies we have implemented to exhibit the basic language features. Programmers would use these aspects to sandbox untrusted code [4, 15, 17, 32].

The first policy, `limitdirectories`, disallows programs from opening files in directories other than the `tests` directory. This is achieved using `before` advice that is triggered by execution of any of the open-file calls in the file I/O library. The before advice checks the method call argument to determine the file to be opened. If the file is not in the right directory the advice prints out a message and `aborts`, terminating the program. This advice calls a number of pure string functions as well as the `print` and `abort` functions, which have I/O and termination effects. Our type system correctly verifies it is harmless.

The second example, `limitcreate`, limits the number of new files a program can create. To do so, it allocates a local reference `filescreated` to keep track of the number of files created so far. By default, such references take on the protection level of the aspect that creates them, in this case `limitcreate`. Once again, the type system verifies that the aspect is harmless. Notice that even though `limitcreate` and `limitdirectories` can be invoked at some of the same control flow points (*e.g.,* `openC`), they are guaranteed not to interfere. Hence, these two policies could have being created by independent programmers and they would still work properly when composed.

The last example, `opencheck`, shows how to use stack patterns to implement a highly simplified stack-inspection-like policy. In this example, we have assumed that `openC` and `openO` are "helper functions" that should only be called by `openW`, which checks to see whether or not the file in question exists before determining which method to call. In this aspect, before advice analyzes the control stack

(advice argument `s`) and only allows execution to proceed if the immediate caller was `openW`. Real stack inspection policies examine the whole control stack, not just the immediate caller. Such policies may be implemented in our system using a recursive method that runs down the stack.

3.3.3 Naccio Security Case Study

To study the usefulness of harmless advice somewhat more broadly in the security domain, we examined the suite of security policies that Evans implemented as part of the Naccio system for his thesis [16]. At the time, Evans thought of Naccio as a domain specific language for implementing security policies, and he argued effectively (as did Erlingsson and Schneider in concurrent research [15]) that his language, which completely separates security code from mainline program, was an effective means of developing reliable security policies. It is now clear that Naccio is form of aspect-oriented programming language, though at the time Naccio was developed, aspects had not yet gained much attention.

We lifted the security policies Evans wrote for Java from his thesis and rewrote them in HarmlessAML, testing them for harmlessness. We omitted those elements of the policies that were particularly Java-specific. For example, in our system, we implemented SML-style file and network I/O system calls, rather than emulating the Java API. Of the group, there was one policy that was not harmless. It was a networking policy called **SoftSendLimit** that divided up the data to be sent on the network into small chunks and limited the sending rate to some preset limit. A straightforward implementation of the policy would require "harmful" around advice that calls the *proceed* function several times with the smaller segments of data to be sent. This policy is not a typical "permit or deny" access-control security policy,

```
limitdirectories:{
  string alloweddirectory = "tests/"
  advice before (| #openR,openW,openA,openC,openO# |) (arg,_,_) =
    let
      string directory = substring(arg,0,(lastindexof(arg,"/")+1))
    in
      if directory == alloweddirectory
      then ()
      else ((print "Forbidden directory.\n");
            (abort ()))
}

limitcreate:{
  ref filescreated : int = 0
  int createlimit = 10
  advice before (| #openC# |) (_,_,_) =
    if (!filescreated + 1) > createlimit
    then ((print "Too many files created.\n");
          (abort ()))
    else ()
  advice after (| #openC# |) (_,_,_) =
    ((filescreated := (!filescreated) + 1); ())
}

opencheck:{
  advice before (| #openC,openO# |) (_,stk,_) =
    case stk
    ( _ :: (| #openW# |)(_,_) :: tail => ()
    | _ => (print ("Invalid openW call.\n");
            abort ()))
}
```

Figure 3.27: Simple Security Aspects

but modifies the behavior of the network I/O implementation. As such, it is harmful by our definition. We would expect (and indeed do find in the rest of the policies) that most security policies would not require such a modification to the behavior of the system.

Some of the other policies we investigated include the following:

- **NoBashingExceptTmp** allows modification of files in the "tmp" directory but no others.
- **LimitWrite** prevents the modification of existing files and limits the number of characters that can be written to the file system.
- **NetLimit** restricts the network send rate to a designated limit. When the rate is being exceeded, our implementation uses `sleep` system calls to slow the send process. This is allowed in our definition of harmlessness.
- **JavaApplet** only allows access to a specified file and only allows connections to a specified host. Our harmless implementation contains aspects on the relevant read, write, observe file system calls and on the relevant connect and accept network system calls. We did not implement the Java-specific stack-inspection security model that the corresponding Naccio policy also enforces. Recall that we had performed such a case study in Section 2.3.
- **Paranoid** implements file read, write, create and observe limits, directory restrictions, and network usage prevention.
- **TarCustom** is a modified **NoBashingFiles** policy except ".tar" files can be overwritten. It only allows read access to specified files, does not allow more bytes written than read, and prohibits all network use.

3.3.4 Meta-theory

To gives a semantics to HarmlessAML, we define a type-preserving transformation into the core language. However, unlike AspectML in the previous chapter, we define a type-preserving transformation into $\mathbb{F}_{HRM2}$, not $\mathbb{F}_{HRM}$. This gives us not one, but two semantics for the source. The first semantics is the *reference semantics* in which every source language aspect is translated into the unit value. In this semantics, aspects cannot possibly be anything but harmless. The second semantics is the *implementation semantics*. In this latter case, every HarmlessAML aspect is implemented as the appropriate core calculus state, objects and advice. Clearly, one implements the second (implementation) semantics not the reference semantics. However, using the key properties of the core — *soundness*, *completeness* and *preservation* — we prove that if the two semantics both terminate and produce results, then the results they produce are equal. Consequently, we obtain our central result: the implementation semantics of HarmlessAML aspects is harmless.

The translation from HarmlessAML to $\mathbb{F}_{HRM2}$ places the state and code for the mainline program and for each aspect into their own protection domains. Recall that the main program and the aspects are syntactically separate due to the structure of the grammar. During translation, $\mathbb{F}_{HRM2}$ expressions generated by translating the mainline program code are assigned the protection domain **MAIN**. The $\mathbb{F}_{HRM2}$ expressions generate by translating an aspect named $ASPECT$ are given the protection domain **ASPECT**.

We then assume that the translation operates in the presence of a security lattice in which **ASPECT** $<$ **MAIN** for all aspects $ASPECT$ defined by the program. Consequently, the translation specifies the noninterference policy that we wish to

Value Translation $\Phi; \Gamma \vdash v : \tau \overset{\mathbf{val}}{\Longrightarrow} v'$

$$\frac{}{\Phi; \Gamma \vdash () : 1 \overset{\mathbf{val}}{\Longrightarrow} ()}\ \text{svt:unit} \qquad \frac{}{\Phi; \Gamma \vdash s : \mathsf{string} \overset{\mathbf{val}}{\Longrightarrow} s}\ \text{svt:string}$$

$$\frac{}{\Phi; \Gamma \vdash \mathtt{true} : \mathsf{bool} \overset{\mathbf{val}}{\Longrightarrow} \mathbf{true}}\ \text{svt:true} \qquad \frac{}{\Phi; \Gamma \vdash \mathtt{false} : \mathsf{bool} \overset{\mathbf{val}}{\Longrightarrow} \mathbf{false}}\ \text{svt:false}$$

Figure 3.28: Value Translation from HarmlessAML to $\mathbb{F}_{HRM2}$

enforce, namely that no aspect interferes with any other aspect and that no aspect interferes with the mainline computation.

The translation from HarmlessAML into the core calculus is defined by a series of five mutually recursive judgments. The translation judgments are generally parameterized by a typing context involving a pointcut context (Φ), which contains a collection of declarations that can be used in source-level pointcuts, a standard type context (Γ), which maps source variables to types, and a protection level/aspect name (p). The point-cut context Φ contains declarations of the form $f : (\tau_{arg}, \tau_{res}, p)$. These declarations say that an function named f has been declared and may be advised. The function takes an argument with type τ_{arg} and returns a result with type τ_{res}. The function inhabits protection domain p.

The form of the translation judgments are as follows.

- The judgment $\Phi; \Gamma \vdash v : \tau \overset{\mathbf{val}}{\Longrightarrow} v'$ in Figure 3.28 describes the translation from HarmlessAML values v with type t to $\mathbb{F}_{HRM2}$ values v' with type t.
- The judgment $\Phi; \Gamma; p \vdash e : \tau \overset{\mathbf{exp}}{\Longrightarrow} e'$ in Figure 3.29 describes the translation from HarmlessAML expressions e with type t to $\mathbb{F}_{HRM2}$ expressions e' with type t.

Expression Translation $\Phi; \Gamma; p \vdash e : \tau \Longrightarrow^{\mathbf{exp}} e'$

$$\frac{\Phi; \Gamma \vdash v : \tau \Longrightarrow^{\mathbf{val}} v'}{\Phi; \Gamma; p \vdash v : \tau \Longrightarrow^{\mathbf{exp}} v'}\ \textsf{set:val} \qquad \frac{\Gamma(x) = \tau}{\Phi; \Gamma; p \vdash x : \tau \Longrightarrow^{\mathbf{exp}} x}\ \textsf{set:var}$$

$$\frac{\Phi; \Gamma; p \vdash e_1 : 1 \Longrightarrow^{\mathbf{exp}} e_1' \qquad \Phi; \Gamma; p \vdash e_2 : \tau \Longrightarrow^{\mathbf{exp}} e_2'}{\Phi; \Gamma; p \vdash e_1; e_2 : \tau \Longrightarrow^{\mathbf{exp}} e_1'; e_2'}\ \textsf{set:seq}$$

$$\frac{\Phi; \Gamma; p \vdash e : \mathsf{string} \Longrightarrow^{\mathbf{exp}} e'}{\Phi; \Gamma; p \vdash \texttt{print}\ e : 1 \Longrightarrow^{\mathbf{exp}} \mathbf{print}\ e'}\ \textsf{set:print}$$

$$\frac{\Phi; \Gamma; p \vdash e_1 : \mathsf{bool} \Longrightarrow^{\mathbf{exp}} e_1' \qquad \Phi; \Gamma; p \vdash e_2 : \tau \Longrightarrow^{\mathbf{exp}} e_2' \qquad \Phi; \Gamma; p \vdash e_3 : \tau \Longrightarrow^{\mathbf{exp}} e_3'}{\Phi; \Gamma; p \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \Longrightarrow^{\mathbf{exp}} \mathbf{if}\ e_1'\ \mathbf{then}\ e_2'\ \mathbf{else}\ e_3'}\ \textsf{set:if}$$

$$\frac{\Phi; \Gamma; p \vdash ds; .; e : \tau \Longrightarrow^{\mathbf{dec}} e'}{\Phi; \Gamma; p \vdash \texttt{let}\ ds\ \texttt{in}\ e : \tau \Longrightarrow^{\mathbf{exp}} e'}\ \textsf{set:let}$$

$$\frac{\Phi; \Gamma; p \vdash e_1 : \tau_1 \to_{p'} \tau_2 \Longrightarrow^{\mathbf{exp}} e_1' \qquad \Phi; \Gamma; p \vdash e_2 : \tau_1 \Longrightarrow^{\mathbf{exp}} e_2' \qquad \vdash p = p'}{\Phi; \Gamma; p \vdash e_1\ e_2 : \tau_2 \Longrightarrow^{\mathbf{exp}} e_1'\ e_2'}\ \textsf{set:app}$$

$$\frac{\Phi; \Gamma; p \vdash e : \tau\ \mathsf{ref}_{p'} \Longrightarrow^{\mathbf{exp}} e' \qquad \vdash p \leq p'}{\Phi; \Gamma; p \vdash\ !e : \tau \Longrightarrow^{\mathbf{exp}} !e'}\ \textsf{set:deref}$$

$$\frac{\Phi; \Gamma; p \vdash e_1 : \tau\ \mathsf{ref}_{p'} \Longrightarrow^{\mathbf{exp}} e_1' \qquad \Phi; \Gamma; p \vdash e_2 : \tau \Longrightarrow^{\mathbf{exp}} e_2' \qquad \vdash p' \leq p}{\Phi; \Gamma; p \vdash e_1 :\!= e_2 : 1 \Longrightarrow^{\mathbf{exp}} e_1' \ :=\ e_2'}\ \textsf{set:asgn}$$

$$\frac{\begin{array}{c}\Phi; \Gamma; p \vdash e_1 : \mathsf{stack} \Longrightarrow^{\mathbf{exp}} e_1' \qquad \Phi; \Gamma; p \vdash pat \Longrightarrow^{\mathbf{pat}} pat' \dashv \Gamma'; \Theta \\ P; \Gamma, \Gamma'; p \vdash e_2 : \tau \Longrightarrow^{\mathbf{exp}} e_2' \qquad \Phi; \Gamma; p \vdash e_3 : \tau \Longrightarrow^{\mathbf{exp}} e_3'\end{array}}{\begin{array}{l}\Phi; \Gamma; p \vdash \texttt{stkcase}\ e_1\ (pat\ \texttt{=>}\ e_2\ \mid\ \_\ \texttt{=>}\ e_3) : \tau \Longrightarrow^{\mathbf{exp}} \\ \quad \mathbf{stkcase}\ e_1'\ (pat' \Rightarrow \mathbf{split}(\Theta, e_2'), \_ \Rightarrow e_3')\end{array}}\ \textsf{set:scase}$$

Figure 3.29: Expression Translation from HarmlessAML to $\mathbb{F}_{HRM2}$

Stack-case Splitting Function $\texttt{split}(\Theta)$

$$\texttt{split}(\cdot, e) = e \qquad \texttt{split}(z \to (x, y), \Theta) = \texttt{split}(\Theta, \textbf{split}\ (x, y) = z\ \textbf{in}\ e)$$

Stack-case Context Translation $\Gamma \dashv \Theta \Rightarrow \Gamma'$

$$\frac{}{\Gamma \dashv \cdot \Rightarrow \Gamma}\ \textsf{stt:end} \qquad \frac{\Gamma \dashv \Theta \Rightarrow \Gamma'}{\Gamma, x : \tau, y : \tau' \dashv \Theta, z \to (x, y) \Rightarrow \Gamma', z : (\tau \times \tau')}\ \textsf{stt:next}$$

Pointcut Context Translation $\mathcal{T}(Phi)$

$$\mathcal{T}(()\cdot) = \cdot$$

$$\mathcal{T}(\Phi, f : (\tau_{arg}, \tau_{res}, p)) = f_{\texttt{bef}} : (\tau_{arg} \times \textsf{string})\ \textsf{label}_p, f_{\texttt{aft}} : (\tau_{res} \times \textsf{string})\ \textsf{label}_p$$

Figure 3.30: Auxiliary Definitions for Translation from HarmlessAML to $\mathbb{F}_{HRM2}$

Pattern Translation $\Phi; \Gamma; p \vdash pat \overset{\textbf{pat}}{\Longrightarrow} pat' \dashv \Gamma'; \Theta$

$$\frac{(f_i \in \Phi)^{(1 \le i \le n)} \qquad (\Gamma(f_i) = \tau_{arg}\ \texttt{->}p_i\ \tau_{res})^{(1 \le i \le n)} \qquad \Phi; \Gamma; p \vdash pat \overset{\textbf{pat}}{\Longrightarrow} pat' \dashv \Gamma'; \Theta}{\begin{array}{c}\Phi; \Gamma; p \vdash \texttt{(|\#}\bar{f}\texttt{\#|)}\ (x\texttt{,}n) :: pat \overset{\textbf{pat}}{\Longrightarrow} \{\overline{f_{\texttt{bef}}}\}_p[\![y]\!] :: pat' : \\ (\Gamma', x : \tau_{arg}, n : \textsf{string}; \Theta, y \to (x, n))\end{array}}\ \textsf{spt:pc}$$

$$\frac{}{\Phi; \Gamma; p \vdash \texttt{[]} \overset{\textbf{pat}}{\Longrightarrow} nil \dashv \cdot; \cdot}\ \textsf{spt:nil} \qquad \frac{\Phi; \Gamma; p \vdash pat \overset{\textbf{pat}}{\Longrightarrow} pat' \dashv \Gamma'; \Theta}{\Phi; \Gamma; p \vdash \_ :: pat \overset{\textbf{pat}}{\Longrightarrow} \_ :: pat' \dashv \Gamma'; \Theta}\ \textsf{spt:any}$$

$$\frac{}{\Phi; \Gamma; p \vdash x \overset{\textbf{pat}}{\Longrightarrow} x \dashv (x : \textsf{stack}); \cdot}\ \textsf{spt:var}$$

Figure 3.31: Pattern Translation from HarmlessAML to $\mathbb{F}_{HRM2}$

Declaration Translation $\Phi; \Gamma; p \vdash as; asps; e : \tau \overset{\mathbf{dec}}{\Longrightarrow} e'$

$$\frac{P; \Gamma; P' \vdash as; .; () : 1 \overset{\mathbf{as}}{\Longrightarrow} e' \qquad P; \Gamma; \mathbf{MAIN} \vdash .; asps; e : \tau \overset{\mathbf{dec}}{\Longrightarrow} e'' \qquad \vdash P' \langle \mathbf{MAIN}}{P; \Gamma; \mathbf{MAIN} \vdash .; P' : \{as\}\ asps; e : \tau \overset{\mathbf{dec}}{\Longrightarrow} P' \texttt{<}()|e' \texttt{>}; e''} \text{ sdt:asp}$$

$$\frac{\Phi; \Gamma; p \vdash e : \tau \overset{\mathbf{exp}}{\Longrightarrow} e'}{\Phi; \Gamma; p \vdash .; .; e : \tau \overset{\mathbf{dec}}{\Longrightarrow} e'} \text{ sdt:end}$$

$$\frac{\Phi; \Gamma; p \vdash e_1 : \mathsf{string} \overset{\mathbf{exp}}{\Longrightarrow} e_1' \qquad P; \Gamma, x : \mathsf{string}; p \vdash as; asps; e_2 : \tau \overset{\mathbf{dec}}{\Longrightarrow} e_2'}{\Phi; \Gamma; p \vdash \texttt{string}\ x\ \texttt{=}\ e_1\ as; asps; e_2 : \tau \overset{\mathbf{dec}}{\Longrightarrow} \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2'} \text{ sdt:string}$$

$$\frac{\Phi; \Gamma; p \vdash e_1 : \mathsf{bool} \overset{\mathbf{exp}}{\Longrightarrow} e_1' \qquad P; \Gamma, x : \mathsf{bool}; p \vdash as; asps; e_2 : \tau \overset{\mathbf{dec}}{\Longrightarrow} e_2'}{\Phi; \Gamma; p \vdash \texttt{bool}\ x\ \texttt{=}\ e_1\ as; asps; e_2 : \tau \overset{\mathbf{dec}}{\Longrightarrow} \mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2'} \text{ sdt:bool}$$

$$\frac{\begin{array}{c}\Phi; \Gamma, x : \tau_1; p \vdash e_1 : \tau_2 \overset{\mathbf{exp}}{\Longrightarrow} e_1' \\ \Phi, f; \Gamma, f : \tau_1 \to_p \tau_2; p \vdash as; asps; e_2 : \tau \overset{\mathbf{dec}}{\Longrightarrow} e_2'\end{array}}{\begin{array}{l}\Phi; \Gamma; p \vdash \texttt{fun}\ f\ (x\texttt{:}\ t_1) : t_2\ \texttt{=}\ e_1\ as; asps; e_2 : t_2 \overset{\mathbf{dec}}{\Longrightarrow} \\ \quad \mathbf{let}\ f_{\texttt{bef}} = \mathbf{new}_p\ (\tau_1, \mathsf{string})\ \mathbf{in}\ \mathbf{let}\ f_{\texttt{aft}} = \mathbf{new}_p\ (\tau_2, \mathsf{string})\ \mathbf{in} \\ \quad \mathbf{let}\ f = \lambda_p x : \tau_1.(\mathbf{store}\ f_{\texttt{bef}}[\![(x, \text{“}f''\text{)}]\!]\ \mathbf{in}\ f_{\texttt{bef}}[\![(x, \text{“}f''\text{)}]\!]; \\ \quad\quad \mathbf{let}\ res = e_1'\ \mathbf{in}\ f_{\texttt{aft}}[\![(res, \text{“}f''\text{)}]\!]; res)\ \mathbf{in}\ e_2'\end{array}} \text{ sdts:fun}$$

$$\frac{\Phi; \Gamma; p \vdash e_1 : \tau \overset{\mathbf{exp}}{\Longrightarrow} e_1' \qquad P; \Gamma, x : \tau\ \mathsf{ref}_p; p \vdash as; asps; e_2 : \tau' \overset{\mathbf{dec}}{\Longrightarrow} e_2'}{\Phi; \Gamma; p \vdash \texttt{ref}\ x\ \texttt{=}\ e_1\ as; asps; e_2 : \tau' \overset{\mathbf{dec}}{\Longrightarrow} \mathbf{let}\ x = \mathbf{ref}_p\ e_1'\ \mathbf{in}\ e_2'} \text{ sdt:ref}$$

Figure 3.32: Declaration Translation from HarmlessAML to $\mathbb{F}_{HRM2}$: Part 1

Declaration Translation $\Phi; \Gamma; p \vdash as; asps; e : \tau \overset{\mathbf{dec}}{\Longrightarrow} e'$

$$\frac{\Phi; \Gamma, x : \tau_1; p \vdash e_1 : \tau_2 \overset{\mathbf{exp}}{\Longrightarrow} e_1' \qquad \Phi, f; \Gamma, f : \tau_1 \to_p \tau_2; p \vdash as; asps; e_2 : \tau \overset{\mathbf{dec}}{\Longrightarrow} e_2'}{\begin{array}{l}\Phi; \Gamma; p \vdash \texttt{fun}\ f\,(x\texttt{:}\,t_1) : t_2\ \texttt{=}\ e_1\ as; asps; e_2 : t_2 \overset{\mathbf{dec}}{\Longrightarrow} \\ \quad \mathbf{let}\ f_{\texttt{bef}} = \mathbf{new}_p\ (\tau_1, \mathsf{string})\ \mathbf{in}\ \mathbf{let}\ f_{\texttt{aft}} = \mathbf{new}_p\ (\tau_2, \mathsf{string})\ \mathbf{in} \\ \quad \mathbf{let}\ f = \lambda_p x : \tau_1.(\mathbf{store}\ f_{\texttt{bef}}[\![(x, \text{“}f''\text{)}]\!]\ \mathbf{in}\ f_{\texttt{bef}}[\![(x, \text{“}f''\text{)}]\!]; \\ \quad\quad \mathbf{let}\ res = e_1'\ \mathbf{in}\ f_{\texttt{aft}}[\![(res, \text{“}f''\text{)}]\!]; res)\ \mathbf{in}\ e_2'\end{array}}\ \textsf{sdts:fun}$$

$$\frac{\begin{array}{c}(f_i \in \Phi)^{(1 \le i \le n)} \qquad (\Gamma(f_i) = \tau_{arg} \to_{p_i} \tau_{res})^{(1 \le i \le n)} \\ \Phi; (\Gamma, x : \tau_{arg}, s : \mathsf{stack}, n : \mathsf{string}); p \vdash e_1 : 1 \overset{\mathbf{exp}}{\Longrightarrow} e_1' \\ \vdash p \le p_i \qquad \Phi; \Gamma; p \vdash as; asps; e_2 : \tau \overset{\mathbf{dec}}{\Longrightarrow} e_2'\end{array}}{\begin{array}{l}\Phi; \Gamma; p \vdash \texttt{advice before (|\#}\bar{f}\texttt{\#|)}\ (x\texttt{,}s\texttt{,}n)\ \texttt{=}\ e_1\ as; asps; e_2 : \tau \overset{\mathbf{dec}}{\Longrightarrow} \\ \quad \Uparrow \{\{\overline{f_{\texttt{bef}}}\}_p(y) \rhd_p \mathbf{split}\ (x, n) = y\ \mathbf{in}\ \mathbf{let}\ s = \mathbf{stack}()\ \mathbf{in}\ e_1'\}; e_2'\end{array}}\ \textsf{sdts:before}$$

$$\frac{\begin{array}{c}(f_i \in \Phi)^{(1 \le i \le n)} \qquad (\Gamma(f_i) = \tau_{arg} \to_{p_i} \tau_{res})^{(1 \le i \le n)} \\ \Phi; (\Gamma, x : \tau_{res}, s : \mathsf{stack}, n : \mathsf{string}); p \vdash e_1 : 1 \overset{\mathbf{exp}}{\Longrightarrow} e_1' \\ \vdash p \le p_i \qquad \Phi; \Gamma; p \vdash as; asps; e_2 : \tau \overset{\mathbf{dec}}{\Longrightarrow} e_2'\end{array}}{\begin{array}{l}\Phi; \Gamma; p \vdash \texttt{advice after (|\#}\bar{f}\texttt{\#|)}\ (x\texttt{,}s\texttt{,}n)\ \texttt{=}\ e_1\ as; asps; e_2 : \tau \overset{\mathbf{dec}}{\Longrightarrow} \\ \quad \Uparrow \{\{\overline{f_{\texttt{aft}}}\}_p(y) \rhd_p \mathbf{split}\ (x, n) = y\ \mathbf{in}\ \mathbf{let}\ s = \mathbf{stack}()\ \mathbf{in}\ e_1'\}; e_2'\end{array}}\ \textsf{sdts:after}$$

Figure 3.33: Declaration Translation from HarmlessAML to $\mathbb{F}_{HRM2}$: Part 2

$\vdash ds\ asps\ e : \tau \overset{prog}{\Longrightarrow} e'$

$$\frac{.; .; \mathbf{MAIN} \vdash ds; asps; e : \tau \overset{\mathbf{dec}}{\Longrightarrow} e'}{\vdash ds\ asps\ e : \tau \overset{\mathbf{prog}}{\Longrightarrow} e'}\ \textsf{spt:prog}$$

Figure 3.34: Program Translation from HarmlessAML to $\mathbb{F}_{HRM2}$

- The judgment $\texttt{split}(\Theta, e)$ in Figure 3.30 is used by the stack case operation. What is extracted from the $\mathbb{F}_{HRM2}$ stack is a tuple containing the argument of the function, and the name of the function. The `split` function extracts the individual elements from these tuples.

- The judgment $\Gamma \dashv \Theta \Rightarrow \Gamma'$ in Figure 3.30 takes a context for individual elements pulled from the stack–the argument of the function, and the name of the function and returns a context containing a tuple of those individual elements. This new context with tuples is what is actually generated by the pattern translation described in the next section. This judgment is used in the proof of translation type safety.

- The judgment $\Phi; \Gamma; p \vdash pat \overset{\textbf{pat}}{\Longrightarrow} pat' \dashv \Gamma; \Theta$ in Figure 3.31 describes the translation from HarmlessAML patterns pat to $\mathbb{F}_{HRM2}$ patterns pat' binding variables described by Γ. Notice that the context Γ returned describes individual elements–the argument to the function, and the name of the function. It is modified by Θ by the judgment $\Gamma \dashv \Theta \Rightarrow \Gamma'$ to generate the new context containing tuples that the $\mathbb{F}_{HRM2}$ pattern pat' actually generates. Later, the `split` command in the stack case translation will be used to extract the individual elements from the tuples.

- The judgment $\Phi; \Gamma; p \vdash as; asps; e : \tau \overset{\textbf{dec}}{\Longrightarrow} e'$ in Figures 3.32 and 3.33 describes the translation of declarations as, aspects $asps$ and mainline code e. The scope of the declarations as includes both $asps$ and e. Mainline code e has type τ and the expression e' that results from the translation has type τ as well.

- The judgment $\vdash$ *ds asps* $e \overset{\textbf{prog}}{\Longrightarrow} e'$ in Figure 3.34 translates a whole HarmlessAML program (*ds asps* e) with a mainline computation producing values of type τ into a $\mathbb{F}_{HRM2}$ expression e' with type τ.

Apart from the addition of protection domains, much of the translation is similar to that in the previous chapter. Throughout the translation we use the abbreviation $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ to stand for $(\lambda_p x{:}\tau.e_2)\ e_1$ for some appropriate type τ and protection p, which can be determined from the context. The interesting cases involve function declarations and advice in Figure 3.33. Function declarations are translated in Rule sdts:fun by first allocating two sets of labels, one set for the control flow points at the beginning of functions, and one for the control flow points at the end of functions. Recall that HarmlessAML programs cannot use around advice. This allows a simpler translation of functions with a before and an after label, rather than the complex translation using an around label in AspectML of the previous chapter. In a rather severe abuse of notation, we bind these new before and after labels to variables with the names "$f_{\texttt{bef}}$" and "$f_{\texttt{aft}}$." During the translation, we maintain the invariant that whenever $f : (\tau_{arg}, \tau_{res}, p)$ appears in the context P, the translated term is well typed in a context including the variables $f_{\texttt{bef}}$ with type $(\tau_{arg}, \mathsf{string})\ \mathsf{label}_p$ and $f_{\texttt{post}}$ with type $(\tau_{res}, \mathsf{string})\ \mathsf{label}_p$. The translation $\mathcal{T}(\Phi)$ that generates a $\mathbb{F}_{HRM2}$ context Γ from a pointcut context Φ is defined in Figure 3.30. In the body of each function, the translation first allocates onto the stack the $f_{\texttt{bef}}$ label with a tuple containing the argument of the function, and a string corresponding to the function name[3]. Then we mark the following control-flow point with the $f_{\texttt{bef}}$ label for the function, passing a tuple including the argument and the function

[3]Again, there is an abuse of notation here. We assume that we may write "f" for the string equivalent of the function name.

name to the advice. Next comes the body of the function and finally, the $f_{\texttt{aft}}$ label including the result and the function name.

Before and after advice are translated in Rules sdts:before and sdts:after respectively. Before advice is triggered by the $f_{\texttt{bef}}$ label, while after advice is triggered by $f_{\texttt{aft}}$ label. In both cases, the first action inside the advice body involves extracting the components (function argument or result, and string name) from the advice argument z. Next, the translated advice reifies the current stack and binds it to the variable s. Finally, the advice executes the translated body. After declaring the advice, the translated code immediately activates it, placing it after any previously encountered advice. The relatively simple translation of HarmlessAML before and after advice stands in contrast to AspectML advice, which must use a more complex translation to handle around advice.

Finally, Rule sdt:asp gives HarmlessAML aspects two interpretations in the core. This rule translates one aspect (p':{as}) in a sequence into a $\mathbb{F}_{HRM2}$ expression. Assuming e' is the code that results from translating as, the declarations that make up the aspect, then the resulting $\mathbb{F}_{HRM2}$ expression is $p'\texttt{<}()|e'\texttt{>}$. In this case, the reference semantics for the aspect is () and the implementation semantics is e'.

The first important property of the translation is that it only produces well-typed $\mathbb{F}_{HRM2}$ expressions.

Lemma 3.3.1 (Split Translation Lemma) *If* $\Gamma, \Gamma'; p \vdash e : \tau$ *and* $\Gamma' \dashv \Theta \Rightarrow \Gamma''$, *then* $\Gamma, \Gamma''; p \vdash \texttt{split}(\Theta, e) : \tau$

Proof: By induction on the structure of $\Gamma \dashv \Theta \Rightarrow \Gamma'$. □

Lemma 3.3.2 (Translation Type Safety Lemmas)

- *If* $\Phi; \Gamma; p \vdash pat \stackrel{\mathbf{pat}}{\Longrightarrow} pat' \dashv \Gamma; \Theta$ *and* $\Gamma \dashv \Theta \Rightarrow \Gamma'$ *then* $\mathcal{T}(\Phi); p \vdash pat' \Rightarrow \Gamma'$.
- *If* $\Phi; \Gamma \vdash v : \tau \stackrel{\mathbf{val}}{\Longrightarrow} v'$, *then* $\mathcal{T}(\Phi), \Gamma \vdash v' : \tau$.
- *If* $\Phi; \Gamma; p \vdash e : \tau \stackrel{\mathbf{exp}}{\Longrightarrow} e'$, *then* $\mathcal{T}(\Phi), \Gamma; p \vdash e' : \tau$.
- *If* $\Phi; \Gamma; p \vdash as; asps; e : \tau \stackrel{\mathbf{dec}}{\Longrightarrow} e'$, *then* $\mathcal{T}(\Phi), \Gamma; p \vdash e' : \tau$.

Proof: By induction on the structure of $\Phi; \Gamma; p \vdash pat \stackrel{\mathbf{pat}}{\Longrightarrow} pat' \dashv \Gamma; \Theta$, $\Phi; \Gamma \vdash v : \tau \stackrel{\mathbf{val}}{\Longrightarrow} v'$, $\Phi; \Gamma; p \vdash e : \tau \stackrel{\mathbf{exp}}{\Longrightarrow} e'$, and $\Phi; \Gamma; p \vdash as; asps; e : \tau \stackrel{\mathbf{dec}}{\Longrightarrow} e'$.

- Case set:scase uses the Split Translation Lemma 3.3.1.

□

Using these lemmas, we can now prove a translation type safety theorem.

Theorem 3.3.3 (Translation Type Safety)
If $\vdash prog : \mathsf{bool} \stackrel{\mathbf{prog}}{\Longrightarrow} e'$, *then* $.; main \vdash_2 e' : \mathsf{bool}$.

Proof: Straightforward use of Translation Type Safety Lemma 3.3.2. □

Since the translation places each HarmlessAML aspect into its own protection domain, which sits below the main program protection domain, a corollary of the Translation Type Safety and Noninterference theorems is that no well-typed HarmlessAML aspect interferes with the main program or other well-typed HarmlessAML aspects. Our final theorem establishes that the reference and implementation semantics of HarmlessAML coincide and therefore that aspects are harmless. The proof has a similar structure to the proof of noninterference for $\mathbb{F}_{HRM}$ covered in Section 3.2.5.

Theorem 3.3.4 (Source Language Aspects are Harmless)

If $\vdash prog : \mathsf{bool} \stackrel{\mathbf{prog}}{\Longrightarrow} e'$

and $(\cdot, \cdot, main, |e'|_1) \longmapsto^*_{top} (S_1, A_1, main, v_1)$

and $(\cdot, \cdot, main, |e'|_2) \longmapsto^*_{top} (S_2, A_2, main, v_2)$

then $v_1 = v_2$.

Proof: Straightforward use of Theorem 3.3.3 and the proof technique and lemmas of Theorem 3.2.31. □

Chapter 4

Interference Policies

4.1 Introduction

In the previous chapter, we laid out some important basic principles for modular programming with aspects, but in an idealized setting where the only effects involved mutable references and simple "print" statements. Real programs access a variety of system resources through various libraries. Moreover, many of these libraries provide a means through which program components may, perhaps unknowingly, interfere with one another. Perhaps the most obvious communication channel occurs through the file system – when advice and mainline code read, write, move, or create files, they can easily interfere with one another. Many other libraries provide access to similar shared resources.

One option is to continue to "bake-in" protection domain checks into the translation rules from HarmlessAML to $\mathbb{F}_{HRM2}$ as new system resources and libraries are added. The problem takes on a new dimension, however, when we consider that our definition of "harmlessness" in the last chapter, where simple "print" statements,

altered termination behavior, and altered timing were allowed, was only one of many possible alternatives. Indeed, a programmer may even want the option to write completely "harmful" advice in certain situations. "Baked-in" protection domain checks for libraries are not enough to satisfy a programmer's need for flexibility in specifying how advice can interact using those libraries.

To provide the flexibility and generality required in realistic programming environments, we have developed and implemented a new mechanism that allows programmers to create custom policies for a library that specify how to manage interference between aspects and mainline code that use that library. More specifically, a programmer creates one or two policy specification files, depending on the library. The first *interference policy* file specifies the static and dynamic checks on the operations in the library to ensure that aspects and the main program do not interact in a "harmful" manner. What the programmer considers "harmful" is specified by the policy. The second *resource policy* file segments an underlying resource of the library, for example, the file system, into regions belonging to the main program and regions belonging to particular aspects. These resource regions can then be used by the dynamic checks of the interference policy to prevent certain actions. For example, the programmer may wish to prevent the main program from reading from a region of the file system that belongs to an aspect.

There are four main contributions in this chapter.

1. We introduce customizable interference and resource policies for system libraries in aspect-oriented programs in Section 4.2. These policies enforce static and dynamic checks to determine how aspects and the main program may interact when they use the system library.

2. In Section 4.3, we extend the core calculus $\mathbb{F}_{HRM}$ to support the static and dynamic checks enforced by the policy system. These extensions to $\mathbb{F}_{HRM}$ include run-time protection domains, existential protection domain types, and dynamic error handling.

3. We describe an algorithm in Section 4.4 that uses an interference policy to generate a translation of HarmlessAML system library function calls into $\mathbb{F}_{HRM}$ expressions. The resulting translation will enforce the static and dynamic checks required by the interference policy during library function calls.

4. To test our interference policy mechanism on a system library, we perform a case study in Section 4.5. We demonstrate that, in addition to many other possibilities, interference policies can certainly be used to enforce the previous chapter's definition of "harmlessness" on a file I/O library. We formalize an underlying noninterfering file I/O library in the core calculus, extending the $\mathbb{F}_{HRM}$ and $\mathbb{F}_{HRM2}$ syntax, operational semantics, and type system with a idealized file system. We prove that the new core file system is type safe and supports strong noninterference properties. We then show that we can define an interference and resource policy for the source-level file I/O library that defines a type safe translation to the noninterfering core-level file I/O library. This allows us to prove that aspects that use the source-level file system are harmless. In this manner, we show that, if the user so desires, interference policies can be useful to enforce our previous definition of harmlessness on system libraries. However, this should not be understood as providing a general result that all interference policies enforce the harmlessness of advice. Recall that in the general case, the purpose of interference policies is to allow

$$
\begin{array}{llll}
(\textit{types}) & t & ::= & \ldots \mid \texttt{infile}\ \mathbf{P} \mid \texttt{outfile}\ \mathbf{P} \mid \texttt{fname} \\
(\textit{vals}) & v & ::= & \ldots \mid \textit{'f'} \\
(\textit{exprs}) & e & ::= & \ldots \mid \texttt{openr}\ \mathbf{P}\ e \mid \texttt{openw}\ \mathbf{P}\ e \\
 & & \mid & \texttt{read}\ e \mid \texttt{write}\ e \\
(\textit{main decl}) & d & ::= & \ldots \mid \texttt{infile}\ x\ \texttt{=}\ e \mid \texttt{outfile}\ x\ \texttt{=}\ e
\end{array}
$$

Figure 4.1: File I/O Library in HarmlessAML

the user to enforce the level of "harmlessness" or "harmfulness" on advice that they desire, not just to enforce the particular harmlessness result that we defined in the previous chapter.

4.2 Policies

In this section, we introduce customizable interference and resource policies for system libraries in aspect-oriented programs. These policies enforce static and dynamic checks to determine how aspects and the main program may interact when they use the system library.

4.2.1 Idealized File I/O Library in HarmlessAML

Before we describe an interference policy for a library, we will first define an idealized file I/O library to be used as an ongoing example. Our source-level file I/O library, whose syntax is displayed in Figure 4.1, is based on a simplified version of the file I/O in Standard ML. First, filenames *'f'* are delineated with quote characters in the syntax. The type system will mark them with the `fname` type. The corresponding files can then be opened for reading or writing by passing the filename to the

```
LOGGER : {
  outfile outfd = openw LOG 'function_results.log'
  advice after (| #f,g,h# |) (res, _, name) =
    (write (outfd, ("leaving "^name^"=>"^int_to_string res^"\n")))
}
```

Figure 4.2: File I/O Logging Aspect

`openr` **P** e or `openw` **P** e library operations. The resulting input and output file descriptors will be given types `infile` **P** and `outfile` **P** respectively.

The annotation **P** on the type of file descriptors describes the integrity level of the file descriptor. Recall that integrity levels in HarmlessAML correspond to either the mainline code **MAIN** or an aspect $\mathbf{ASPECT_i}$. Newly created descriptors take on the annotation of the `openr` or `openw` command with which they were created. For example, an input file descriptor that was created with a `openr` **MAIN** command will have the type `infile` **MAIN**, while an output file descriptor that was created with a `openw` **LOGGER** operation will have type `infile` **LOGGER**.

The `read` e operation reads as many characters as possible from an input file descriptor and returns a string containing those characters. The Standard ML file read operation takes an extra integer that specifies how many characters to read from the file, but we have chosen to eliminate this option to make the operation easier to formalize later. The `write` e operation is passed a tuple (e_1, e_2), writing the string e_2 to the output file descriptor e_1. To simplify the formalization of file writing, our `write` operation overwrites the file pointed to by the output file descriptor. The more complex Standard ML file read and write operations could be written as combinations of our simple `read` and `write` operations and string manipulation functions.

In Figure 4.2, we demonstrate a simple file I/O logging aspect. The second line opens the '`functionresults.log`' file for writing and stores the resulting output file descriptor in the *outfd* variable. The advice in the third and fourth lines writes a log message to the file after the *f*, *g*, or *h* functions are run.

We can imagine several different interference policies that a user may wish to enforce on our file I/O library. In the examples in Section 3.3.2 in the previous chapter, we treated all I/O as harmless. We could create an interference policy for the file I/O library that would similarly enforce no static or dynamic checks on the library.

Another policy might result from recalling that in the previous chapter, we enforced a strict separation between the main program and aspects to create clear and simple translation rules, even when a slightly less restrictive policy would still preserve harmlessness. For example, output file descriptors created by the main program were placed in the **MAIN** protection domain, even though they could have safely been placed in an aspect protection domain.

A less restrictive policy might loosen the separation to allow code to create file descriptors with protection domains above and below the code, as long as the resulting operations were harmless. This gives more flexibility to the programmer but has the disadvantage of making the protection domain checks in the code more difficult to understand. The resulting interference policy will still ensure that aspects cannot interfere with the main program through the use of the file system. For example, such a policy would prevent the main program from writing to output file descriptors defined by an aspect, but allow the main program to read from an aspect's input file descriptors. Similarly, the user may determine that aspects

(*iotypes*)	*it*	::=	iotype t
(*dynamic checks*)	*dyn*	::=	`dynamic n` | `dynamic r` **P**
		|	`dynamic w` **P** | `dynamic rw` **P**
(*static checks*)	*sta*	::=	`static n` | `static r` **P**
		|	`static w` **P**p | `static rw` **P**
(*operations*)	*o*	::=	fun $f : t \rightsquigarrow t$ [$\overline{sta}$] [dyn]
(*policy*)	*pol*	::=	$it : \{\overline{o}\};$
(*policy file*	*pol f*	::=	$\overline{pol}$

Figure 4.3: Policy File Syntax

should be able to read from input file descriptors but not to write to output file descriptors defined in the main program.

However, when we attempted to formalize the above policy (to be discussed in Section 4.5), we discovered that the main program could not be allowed to write to output file descriptors defined by an aspect. The act of looking up the file that the output file descriptor is pointing to is a "read" from the file descriptor in our formalization, even if the actual data in the file is not read. This is an artifact of how we formalize file I/O. A similar issue in the previous chapter was encountered in the formalization of the label store in Rule cet:jp. Therefore, to preserve harmlessness, we disallow the main program from writing to output file descriptors defined by aspects. We will use this modified policy as an example in the rest of this chapter.

4.2.2 Interference Policies

We allow the user to determine an *interference policy*, whose syntax is shown in Figure 4.3, that specifies how the aspects and main program interact when they use a library. That is, rather than having the language designer manually specify once and for all how the protection domains in the types used by a library's operations will interact, we will allow this power to the user for their particular application

requirements. In the following sections, we use a mutable reference library, defined in the previous chapter, and the file I/O library, defined in Section 4.2.1, in our examples in this section. They contains operations that both read and write from state, albeit in different ways. The file I/O library, with its file descriptors and underlying file system resource will provide a base to allow us to demonstrate the full utility of our policies.

First, types used by library operations can be divided into three categories: regular types, *descriptor types*, and *resource types*. Regular types, such as `Unit`, `Bool`, or `String`, point to immutable values. *Descriptor types* represent a descriptor created during program execution to point to some mutable data. Descriptor types include reference types τ `ref` $\mathbf{P}$, file descriptor types `infile` $\mathbf{P}$ and `outfile` $\mathbf{P}$, and network socket types. Descriptor types must be annotated with an integrity level $\mathbf{P}$ that describes the integrity level of the value that they point to. *Resource types* are assigned to values that are addresses into an underlying resource structure. Resource types include filename types `fname` and network address types.

Three example interference policies are shown in Figure 4.4. The first two sections of the figure contain example interference policies for a mutable reference library. Recall that in the previous chapter, the static protection domain checks were baked into Rules set:deref, set:asgn, and sdt:ref of the translation in Figures 3.29 and 3.32. The difference between the two policies is that the first policy enforces a strict separation between the main program and aspects to create clear and simple translation rules. For example, references created by the main program are placed in the **MAIN** protection domain, even though they can safely be placed in an aspect protection domain. The second interference policy loosens the syntactic separation to allow code to create references with protection domains below the code, while still

Strict Example Interference Policy for Reference Library

```
{          (* argtype ~> restype   [statchecks]  [dynchecks]*)
  fun ref P : t ~> t ref P         [static rw P] [dynamic n]
  fun !     : t ref P ~> t         [static r P]  [dynamic n]
  fun :=    : t ref P * t ~> unit  [static w P]  [dynamic n]
};
```

Loose Example Interference Policy for Reference Library

```
{          (* argtype ~> restype   [statchecks]  [dynchecks]*)
  fun ref P : t ~> t ref P         [static w P]  [dynamic n]
  fun !     : t ref P ~> t         [static r P]  [dynamic n]
  fun :=    : t ref P * t ~> unit  [static w P]  [dynamic n]
};
```

Example Interference Policy for File I/O Library

```
fname : {   (* argtype ~> restype         [statchecks]  [dynchecks]*)
 fun read  : infile P ~> string           [static r P]  [dynamic n]
 fun write : (outfile P * string) ~> unit [static rw P] [dynamic n]
 fun openr P : fname ~> infile P          [static w P]  [dynamic r P]
 fun openw P : fname ~> outfile P         [static w P]  [dynamic w P]
};
```

Figure 4.4: Example Interference Policies

preserving the harmlessness of advice. However, some of the separation between the main program and aspects has been lost, making the program more confusing to understand. Allowing the programmer to specify these trade-offs is an example of the flexibility allowed by the interference policy system. The last section of the figure contains an example interference policy for the file I/O library from Section 4.2.1.

Each interference policy first declares the resource type, if any, of the library. Descriptor types do not need to be declared as they can be deduced from the operation specifications in the rest of the policy. In the reference example policies, the reference type ref is a descriptor type, not a resource type. In the file I/O example policy, the resource type `fname` is declared at the start of the policy.

Each subsequent line in the policy consists of an operation, the argument and result type of the operation, the static interference checks that will be placed on that operation, and the dynamic interference checks that will be placed on that operation. The example interference policies for mutable references will apply to three operations: reference creation, dereference, and assignment. The example interference policy for file I/O will apply to four operations, `read`, `write`, `openr` and `openw`. Notice that in Figure 4.4, the `ref`, `openr`, and `openw` operations create reference and file descriptors respectively and, as such, are marked with an integrity level annotation, indicating the protection domain of the reference or file descriptor they create. The descriptor types ref, infile, and outfile are similarly marked with an integrity level annotation. As we will see in the next section, the static checks will use these annotations to enforce an interference policy on the library operations.

Interference Policies: Static Checks

In each line of a policy, static checks are listed first, followed by the dynamic checks. Static checks enforce a policy on the descriptor types. To aid the programmer in creating and understanding interference policies, we avoid requiring the user to declare static checks in the form of potentially confusing explicit inequalities on protection domains ($\vdash \mathbf{P_1} \leq \mathbf{P_2}$), as was done in the "baked-in" checks of the previous chapter.

Instead, we observed that the common use of static checks is in the framework of a traditional "read-up, write-down" integrity policy. As such, our static checks fall into four categories: *none* (`n`), *read-up* (`r`), *write-down* (`w`), and *read/write* (`rw`). This classification reflects the observation that operations can either ignore descriptors, read from descriptors, modify descriptors, or can both read and modify descriptors. These categories are similar to the notion of nonvariance, covariance, contravariance, and invariance in subtyping mechanisms.

An operation that does not refer to a descriptor does not require (`n`) a static check. An operation that reads from a descriptor can be assigned a *read-up* (`static r` $\mathbf{P}$) static check, which enforces a "read-up" integrity policy that forbids high integrity code from using the operation on a low integrity descriptor. For example, the user may want to prevent the mainline program from reading from the state of an aspect. The protection domain inequality that is generated by the policy mechanism and checked during compilation is $\mathbf{P_{desc}} \leq \mathbf{P_{curr}}$. Similarly, a *write-down* (`static w` $\mathbf{P}$) static check can be used to implement a "write-down" integrity policy on operations that write to a descriptor in order to forbid low-integrity code from using the operation on a high-integrity descriptor. The user may wish to prevent an aspect from modifying the state of the mainline code. The

corresponding protection domain inequality generated by the policy mechanism and enforced during compilation is $\mathbf{P_{curr}} \leq \mathbf{P_{desc}}$. Finally, a *read/write* (`static rw` **P**) static check ensures that both $\mathbf{P_{desc}} \leq \mathbf{P_{curr}}$ and $\mathbf{P_{curr}} \leq \mathbf{P_{desc}}$.

All of the static checks except *none* are annotated with an integrity level **P**. These integrity level annotations on the static checks are matched with the integrity level annotations in the argument and result types of the operation in the policy to determine which descriptor the static checks will be enforced on. This matching of annotations is trivial for our simple mutable reference and file I/O libraries, which only manipulate one descriptor at a time. However, we found the complete annotation matching mechanism useful when attempting to write an interference policy for a complex network I/O library, where multiple annotations are necessary. One of the network I/O operations listens to a passive network socket (with one annotation) and then creates an active network socket (with a second annotation).

For our mutable reference library, the example policy in the top half of Figure 4.4 enforces the same definition of harmlessness as the "baked-in" checks in the translation of reference operations in the previous chapter. The reference assignment operation modifies a descriptor and thus modify the state of the system. Therefore, it is given a *write* static check. The reference dereference operation reads from a descriptor and thus reads from the state of the system. Therefore, it is given a *read* static check. Notice that in the first, more-restrictive policy, a *read/write* static check is enforced on the reference creation operation to preserve a syntactic separation between the main program and aspects. In the second, less-restrictive policy, a *write* static check is used to preserve the harmlessness of advice while allowing mainline code to use this mutable reference library to create references in the protection domain of an aspect.

For our file I/O library, the user may attempt to enforce the less restrictive policy that we described in Section 4.2.1. This policy is displayed in the bottom half of Figure 4.4. We must identify how each file system library operation will use its file descriptors. The *read* operation will read from an input file descriptor and requires a `static r` **P** check in the library policy, while the *write* operation will both read (to determine what file the descriptor is pointing to) and write (the data to the file) to the output file descriptor and requires a `static rw` **P** check in the library policy. The *openr* and *openw* operations create (write) file descriptors and require `static w` **P** checks in the file system library policy.

Once the static checks required by the file system library policy are in place, the main program to read from input file descriptors and write to output file descriptors belonging to the main program, but not to read from input file descriptors or write to output file descriptors defined by an aspect. Less restrictively, aspects should be able to read from input file descriptors but not to write to output file descriptors defined in the main program. This is the read-up, write-down integrity policy that our example user attempted to enforce.

Interference Policies: Dynamic Checks

Static checks can be used to protect integrity when reading from and writing to an existing descriptor, and are enough to enforce a full interference policy on our mutable reference library. However, static checks are not enough when creating a descriptor that points to an underlying I/O resource. In our example policy for the file I/O library, static checks are used to ensure that, if the main program has a file open for writing, that same file descriptor cannot then be used by an aspect to write to the file. However, static checks cannot prevent the following scenario: an aspect

opens a file and writing to the resulting file descriptor; then, the main program opens that same file and reads from its resulting file descriptor. In this situation, the static checks on file descriptors required by our example file I/O interference policy have not been violated, but the aspect has interfered with the behavior of the main program.

To protect against such manipulation of the underlying resource, a user can require that the interference policy enforce checks on the use of resource types. Recall that resource types are assigned to values that are addresses into an underlying resource structure. The filename type `fname` is a resource type, as is a network address type. These checks on the use of resource types cannot occur at compile time, because a static check cannot determine the particular resource that will be used.

Instead, we use a system of dynamic checks. First, the underlying resource is segmented using a *resource policy* into regions belonging to the main program and regions belonging to particular aspects. This has the effect of associating an integrity level at run time with each resource. This resource policy system will be described fully in the next section.

Next, the dynamic checks in the interference policy can be placed on operations that create a descriptor by accessing an underlying resource. They describe the relationship between the integrity level associated with the accessed resource (described by the resource policy) and the integrity level associated with the created descriptor. Like static checks, dynamic checks are divided into the same four categories: none, read-up, write-down, and read/write. All of the dynamic checks except none are annotated with the integrity level of the descriptor that the resource will be dynamically checked against. If no underlying resource is accessed to create

a descriptor, then there will be no (`dynamic n`) dynamic checks. If the underlying resource will be read by the created descriptor, then a read-up (`dynamic r` **P**) dynamic check can be used. If the underlying resource will be written by the created descriptor, then a write-down (`dynamic w` **P**) dynamic check can be used. Finally, if the underlying resource is opened for reading and writing by the created descriptor, then a read/write (`dynamic rw` **P**) dynamic check can be enforced by the interference policy.

In our example interference policy for a file I/O library in Figure 4.4, the `read` and `write` operations do not require dynamic checks (`dynamic n`) as they do not manipulate filenames. As such, static checks will be enough to ensure that their use of file descriptors does not cause aspects to interfere with the main program. However, we have inserted dynamic checks on the `openr` and `openw` operations to ensure that the files they open are not in a region of the file system that would cause interference if read or modified. The `openr` operation has a dynamic read check (`dynamic r` **P**) that ensures that the high-integrity main program will not be able to open a file belonging to an aspect's region of the file system. Similarly, the `openw` operation has a dynamic write check (`static w` **P**) that ensures that low-integrity aspects cannot open a file that belongs to the main program's region of the file system.

When a dynamic check fails, there are many possible actions that may be desired. The most restrictive option, and that chosen for the proof-of-concept Standard ML implementation of the interference policy mechanism, is to enforce the interference policy at all times. Upon failing a dynamic check, the program immediately terminates and a warning message is printed to the screen. Another option inspired by web browser design would be to inform the user through a pop-up

```
segment sourceCodeSpecifier {
  region regionType1 regionSpecifier1 regionIntegrityLevel1;
  region regionType2 regionSpecifier2 regionIntegrityLevel2;
  ...
};
...
```

Figure 4.5: Syntax of Resource Policies

message that the program has violated a dynamic check. The user could then choose to temporarily override their interference policy and continue program execution, or could choose to terminate the program.

4.2.3 Resource Policies

As described in the previous section, the dynamic checks specified in an interference policy require the underlying resource to be segmented into regions corresponding to different integrity levels. This mapping of regions to integrity levels is specified using a resource policy file. Loosely modeled on the policy specification files of the Java security mechanism, the resource policy specification can be found in Figure 4.5. A resource policy consists of a set of `segment` declarations, each of which specify a segment of source code and the resource segmentation that applied to programs in that source code. Region segmentation is specified by the name of the region type to be segmented, a region specifier (in a different format for each region type) that defines the segment of the underlying resource, and the integrity level of the region.

In the specification, the *sourceCodeSpecifier* selects the source code to which the region segmentation applies. In Java policy files, the policy file writer is allowed to select code by specifying a `codebase` (where the code is located in the file system), a `signer` (the key that has cryptographically signed the code), and a `principal`

```
segment codebase "code/-" {
  region fname "home/-", "MAIN";
  region fname "logfile/-", "LOG";
};
```

Figure 4.6: Example Resource Policy for a File System

(the entity that is executing the code). For the purposes of our resource policies, we have chosen to allow only the `codebase` specifier in resource policies.

When a dynamic check specified by the interference policy is run, the region (and corresponding integrity level) is determined by extracting the `segment` block that corresponds to the location of the running program. Then, the first `region` line that applies to the resource is located, and the integrity level of the region returned to be used in the dynamic check. In a `region` line, the *resourceType* specifies the resource type that the region belongs to. The *regionSpecifies* defines the region and is specific to the particular resource type–a file system path for a file I/O library or a URL for a network I/O library. Finally, the *resourceIntegrityLevel* specifies the integrity level that the region is mapped to by the resource policy.

Figure 4.6 presents an example resource policy for the file I/O library presented in Section 4.2.1. Recall that the *codebase* specifies to which source language programs this resource policy applies. In this example policy, a source language program that exists in the "*code/*" directory will have the specified resource policy.

Next, recall that the lines marked "*region*" associate a region of the file system with a particular integrity level. In this example policy, the "*home/*" directory is associated with the mainline program, **MAIN**. The "*logfile/*" directory is associated with the logging aspect, **LOG**.

Again, taking the mainline program as the highest-integrity code, aspects as lower integrity code, we continue to enforce a read-up, write-down integrity policy on opening files for reading and writing. Files in the "*home/*" directory can be opened for reading but not for writing by aspects. Files in the "*logfiles/*" directory can be opened for writing but not reading by the mainline program. Finally, files in the "*config/*" directory can be opened for reading but not writing by both the mainline program and aspects.

Finally, we have provided functions in our implementation that allow a user to add new region types to the resource policy parsing and analysis mechanism. The user must specify the name of the region type (used in region policy file parsing), an *addRegion* function of type `String` $\times$ `String` $\rightarrow$ *region* that parses the region specifier and integrity level from the policy file and returns the resulting region, and an `insideRegion` function of type *region* $\times$ *region* $\rightarrow$ *integrityLevel option*. The *insideRegion* function takes in a region that represents a resource used in a dynamic check and a region specified by the resource policy file and a region that represents a resource used in a dynamic check and, if the first is inside the second, returns the resulting integrity level that the region is mapped to.

Accordingly, when we added a file I/O region type to our implementation, we specified the name *FileRegion* and an *addFileRegion* function which parses the file paths in the *regionSpecifier* into an internal representation. Finally, we specified an *insideFileRegion* function which takes a filename specified during a dynamic check and tests to see whether the file is inside a file system region specified in the resource policy. If so, it returns the integrity level mapped to that file system region.

4.3 Core Calculus: Extensions to $\mathbb{F}_{HRM}$

As stated earlier, we define our language at two levels of abstraction: a user-friendly, high-level source language for programmers and a lower-level core calculus. This core calculus consists of a collection of simple, orthogonal constructs amenable to formal analysis. We will now introduce the new core calculus features required implement the interference policy mechanism: run-time protection domains, protection domain existentials, and error handling. The first two features are inspired by Tse and Zdancewic's design of run-time principals for information flow type systems [50].

4.3.1 Run-time Protection Domains

Figure 4.7 presents the semantics of run-time protection domains. For every static protection domain $\mathbf{P}$, there exists a corresponding value $\mathbb{P}$. When an interference policy requires a dynamic check on a library operation, the resource policy file is queried to determine what region and thus integrity level the resource in question belongs to. This resource policy query returns a value $\mathbb{P}$ that represents the protection domain $\mathbf{P}$. The run-time protection domain $\mathbb{P}$ is given the singleton type $\mathsf{S}_{\mathbf{P}}$ in Rule cvt:sing.

Run-time decisions concerning protection domain $\mathbf{P}$ can be made by querying $\mathbb{P}$. Run-time protection domains created during program evaluation can be compared with one another with an **ifpd** expression. The operational semantics for the comparison can be seen in Rules ceb:ifpdthen and ceb:ifpdelse. The expression $\mathbf{ifpd}(\mathbb{P} \leq \mathbb{P}')\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2$ evaluates to e_1 if the protection domain $\mathbf{P}$ is below or equal to the protection domain $\mathbf{P}'$ in the static protection domain lattice $\mathcal{L}$.

Extension to $\mathbb{F}_{HRM}$ Syntax

$$\begin{array}{lrcl} (\textit{types}) & \tau & ::= & \ldots \mid \mathsf{S}_p \\ (\textit{values}) & v & ::= & \ldots \mid \mathbb{P} \\ (\textit{expressions}) & e & ::= & \ldots \mid \mathbf{ifpd}(e \leq e)\,\mathbf{then}\,e\,\mathbf{else}\,e \\ (\textit{lattice envs}) & \Pi & ::= & \mathcal{L} \mid \Pi, p \leq p' \end{array}$$

Well-formed Values $\Delta; \Pi; \Gamma \vdash v : \tau$

$$\frac{}{\Delta; \Pi; \Gamma \vdash \mathbb{P} : \mathsf{S}_{\mathbf{P}}}\ \textsf{cvt:sing}$$

Well-formed Expressions $\Delta; \Pi; \Gamma; p \vdash e : \tau$

$$\frac{\Delta; \Pi; \Gamma; p \vdash e_1 : \mathsf{S}_{p'} \qquad \Delta; \Pi; \Gamma; p \vdash e_2 : \mathsf{S}_{p''} \qquad \Delta; \Pi, p' \leq p''; \Gamma; p \vdash e_3 : \tau \qquad \Delta; \Pi; \Gamma; p \vdash e_4 : \tau}{\Delta; \Pi; \Gamma; p \vdash \mathbf{ifpd}(e_1 \leq e_2)\,\mathbf{then}\,e_3\,\mathbf{else}\,e_4 : \tau}\ \textsf{cet:ifpd}$$

β-reduction $(\Sigma; A; \mathbf{P}, e) \longmapsto_\beta (\Sigma'; A'; \mathbf{P}; e')$

$$\frac{\mathcal{L} \vdash \mathbf{P}' \leq \mathbf{P}''}{(\Sigma; A; \mathbf{P}; \mathbf{ifpd}(\mathbf{P}' \leq \mathbf{P}'')\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2) \longmapsto_\beta (\Sigma; A; \mathbf{P}; e_1)}\ \textsf{ceb:ifpdthen}$$

$$\frac{\mathcal{L} \vdash \mathbf{P}' \not\leq \mathbf{P}''}{(\Sigma; A; \mathbf{P}; \mathbf{ifpd}(\mathbf{P}' \leq \mathbf{P}'')\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2) \longmapsto_\beta (\Sigma; A; \mathbf{P}; e_2)}\ \textsf{ceb:ifpdelse}$$

Protection Domain Inequality $\Pi \vdash p \leq p'$

$$\frac{}{\Pi \vdash p \leq p}\ \textsf{clatt:symm} \qquad \frac{(p \leq p'') \in \Pi \qquad \Pi \vdash p'' \leq p' \qquad p \neq p'}{\Pi \vdash p \leq p'}\ \textsf{clatt:trans}$$

Figure 4.7: Run-time Protection Domains Extension to $\mathbb{F}_{HRM}$

As before, the protection domain lattice, $\mathcal{L} = (\overline{\mathbf{P}}, \leq)$, contains a list of concrete protection domains, and a partial order on them. With the addition of **ifpd** expressions, that hierarchy must be updated at run time to reflect the assumptions made while typing the "true" branch of the expression. As seen in typing rule cet:ifpd of Figure 4.7, the **then** branch of an $\mathbf{ifpd}(\mathbb{P} \leq \mathbb{P}')\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2$ expression adds the assumption $\mathbf{P} \leq \mathbf{P}'$ to the protection domain lattice. Therefore, the typing judgments for values and expressions include a protection domain hierarchy environment, Π, which contains the partial-order of protection domains. The initial environment of Π (before any **ifpd** assumptions are added) is the original static protection domain lattice $\mathcal{L}$.

Rules clatt:symm and clatt:trans determine whether one protection domain is *below* another in this new protection domain hierarchy. Rule clatt:symm says that the resulting hierarchy is symmetric. Rule clatt:trans show that if there is a path from protection domain p to p' in the hierarchy Π, then $\Pi \vdash p \leq p'$.

4.3.2 Existential Protection Domain Types

Figure 4.8 presents the semantics of protection domain existentials. As stated above, during evaluation, when the resource policy file is queried as required by a dynamic check of the interference policy, a run-time protection domain $\mathbb{P}$ with singleton type $\mathsf{S}_{\mathbf{P}}$ is created. However, until that moment, it cannot be determined what protection domain is to be returned. Since the type of a run-time protection domain $\mathbb{P}$ includes the protection domain $\mathbf{P}$, we cannot determine at compilation time what the type of the value returned by the resource policy query will be. Therefore, the resource policy queries must return a run-time protection domain *for some protection domain*.

Extension to $\mathbb{F}_{HRM}$ Syntax

$$
\begin{array}{llcl}
(\textit{prot doms}) & p & ::= & \rho \mid \mathbf{P} \\
(\textit{types}) & \tau & ::= & \ldots \mid \exists\rho.\tau \\
(\textit{values}) & v & ::= & \ldots \mid \mathbf{pack}(p, v)\ \mathbf{as}\ \exists\rho.\tau \\
(\textit{exprs}) & e & ::= & \ldots \mid \mathbf{pack}(p, e)\ \mathbf{as}\ \exists\rho.\tau \mid \mathbf{open}(\rho, x) = e\,\mathbf{in}\,e \\
(\textit{pdv env}) & \Delta & ::= & . \mid \Delta, \rho
\end{array}
$$

Well-formed Values $\Delta; \Pi; \Gamma \vdash v : \tau$

$$\frac{\Delta \vdash p \qquad \Delta; \Pi; \Gamma \vdash v : \tau[p/\rho]}{\Delta; \Pi; \Gamma \vdash \mathbf{pack}(p, v)\ \mathbf{as}\ \exists\rho.\tau : \exists\rho.\tau}\ \textsf{cvt:pack}$$

Well-formed Expressions$\Delta; \Pi; \Gamma; p \vdash e : \tau$

$$\frac{\Delta \vdash p' \qquad \Delta; \Pi; \Gamma; p \vdash v : \tau[p'/\rho] \qquad e\ \mathbf{is\ not\ a\ value}}{\Delta; \Pi; \Gamma; p \vdash \mathbf{pack}(p', e)\ \mathbf{as}\ \exists\rho.\tau : \exists\rho.\tau}\ \textsf{cet:pack}$$

$$\frac{\begin{array}{c}\Delta; \Pi; \Gamma; p \vdash e_1 : \exists\rho.\tau_1 \\ \Delta, \rho; \Pi; \Gamma, x : \tau_1; p \vdash e_2 : \tau_2 \qquad \tau_2\ \mathbf{does\ not\ contain}\ \rho\end{array}}{\Delta; \Pi; \Gamma; p \vdash \mathbf{open}(\rho, x) = e_1\,\mathbf{in}\,e_2 : \tau_2}\ \textsf{cet:open}$$

β-reduction $(\Sigma; A; \mathbf{P}, e) \longmapsto_\beta (\Sigma'; A'; \mathbf{P}; e')$

$$\frac{}{\begin{array}{l}(\Sigma; A; \mathbf{P}; \mathbf{open}(\rho, x) = (\mathbf{pack}(\mathbf{P}', v)\ \mathbf{as}\ \exists\rho.\tau)\,\mathbf{in}\,e) \longmapsto_\beta \\ \quad (\Sigma; A; \mathbf{P}; e[\mathbf{P}'/\rho][v/x])\end{array}}\ \textsf{ceb:o}$$

Well-formed Protection Domains $\Delta \vdash p$

$$\frac{\rho \in \Delta}{\Delta \vdash \rho}\ \textsf{cpd:var} \qquad\qquad \frac{}{\Delta \vdash \mathbf{P}}\ \textsf{cpd:conc}$$

Figure 4.8: Existential Protection Domain Types Extension to $\mathbb{F}_{HRM}$

To represent the idea of an expression that is well-typed *for some protection domain* in the core calculus, we use existential types. The type, $\exists\rho.\tau$, is assigned to an expression that has the type τ for some protection domain ρ. In our example above where τ is a singleton type, the type of the value returned by a resource policy query is $\exists\rho.\mathsf{S}_\rho$.

Existentials are created with the **pack** expression, $\mathbf{pack}(p, v)$ **as** $\exists\rho.\tau$. As seen in Rule cvt:pack, this expression takes a value v of type $\tau[p/\rho]$ and wraps it into an existential of type $\exists\rho.\tau$.

To unwrap an existential, the **open** expression, $\mathbf{open}(\rho, x) = e_1\,\mathbf{in}\,e_2$, is used. This expression unwraps an existential e_1 of type $\exists\rho.\tau_1$ into a protection domain p and a value v. As seen in evaluation rule ceb:open, the unwrapped protection domain p is substituted for ρ, and the unwrapped value v is substituted for x in e_2. Typing rule cet:open describes the typing of the **open** expression—notice that ρ is only bound inside e_2 and cannot escape as a free variable into the rest of the program.

With the introduction of existentials, protection domains, p, can be either concrete protection domains, $\mathbf{P}$, or protection domain variables, ρ. Therefore, we have to account for the presence of protection domain variables in protection domains, types, and environments. The protection domain variable environment, Δ, contains the bound protection variables. Rules cpd:var and cpd:conc demonstrate that protection domains are well-formed if they are a concrete protection domain, $\mathbf{P}$, or if they are a protection domain variable ρ bound in Δ. Rule ct:t shows that for a type to be well-formed in the presence of a context Δ, all of the protection domains in the type must be well-formed. Rule cg:g states that, for a context Γ to be well-formed, all of the types in the context must be well-formed. Rule camt:amt

Extension to $\mathbb{F}_{HRM}$ Syntax

$$(exprs) \quad e \quad ::= \quad ... \quad | \quad \mathbf{abort}_{\tau}$$

Well-formed Expressions $\Delta; \Pi; \Gamma; p \vdash e : \tau$

$$\frac{\Delta \vdash \tau}{\Delta; \Pi; \Gamma; p \vdash \mathbf{abort}_{\tau} : \tau}\ \mathsf{cet{:}abort}$$

β-reduction $(\Sigma; A; \mathbf{P}, e) \longmapsto_{\beta} (\Sigma'; A'; \mathbf{P}; e')$

$$\frac{}{(\Sigma; A; \mathbf{P}; \mathbf{abort}_{\tau}) \longmapsto_{\beta} (\Sigma; A; \mathbf{P}; \mathbf{abort}_{\tau})}\ \mathsf{ceb{:}abort}$$

Figure 4.9: Error Handling Extension to $\mathbb{F}_{HRM}$

demonstrates the rules for well-formedness of the store S that contains the input and output file descriptors. Finally, Rule camt:nms shows well-formed machine states, and is used to later type safety theorems.

They are not shown here to avoid needless repetition, but many core calculus typing rules in Section 3.2.2 have been trivially updated as necessary to reflect the addition of protection domain variables. For example, the function abstraction typing rule now includes a $\Delta \vdash p$ in its premises to ensure that the protection domain annotation on the function is well-formed.

4.3.3 Error Handling

The **abort** expression was not formalized in the previous core calculus because it was not generated by the translation from source to core and thus not used in the Translation Type Soundness proof. However, the dynamic checks generated by interference policies will insert core calculus **abort** expressions. As such, to prove

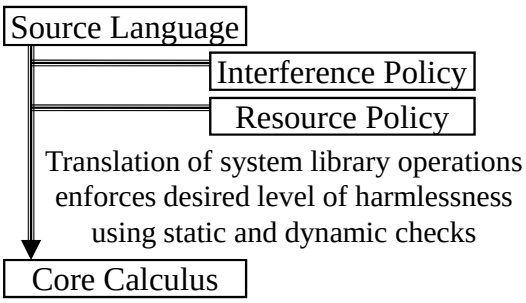

Figure 4.10: Interference Policy Compilation Strategy

a Translation Type Soundness proof in Section 4.5, we formalize the static and operational semantics of **abort** expressions in Figure 4.9.

The typing rule cet:abort assigns to the **abort** expression the type that the expression is annotated with. The evaluation rule ceb:abort simulates aborting by looping endlessly. In our actual Standard ML implementation, the **abort** primitive terminates the program with an error message. We can also envision less restrictive implementations that would simply warn the user or allow the user to choose whether to continue or not.

4.4 Translation from HarmlessAML to $\mathbb{F}_{HRM2}$ Generated by Interference Policies

The compilation strategy for harmless advice with interference policies is displayed in Figure 4.10. As before, source language programs are translated into core calculus expressions. However, the translation rules now depend on the interference policy and the resource policy.

Source Grammar Requirements

$$
\begin{array}{llcl}
 & & & addrval \in \text{Resource Addresses} \\
(types) & t & ::= & ... \mid addrtyp \mid desctyp_{\mathbf{P}} \\
(vals) & v & ::= & ... \mid addrval \\
(exprs) & e & ::= & ... \mid librfun1_{\mathbf{P}}\ e \mid librfun2\ e
\end{array}
$$

Core Calculus Requirements

$$
\begin{array}{llcl}
 & addrval \in \text{Resource Addresses} & & descval \in \text{Resource Descriptors} \\
(types) & \tau & ::= & ... \mid addrtyp_p \mid desctyp_p \\
(values) & v & ::= & ... \mid addrval \mid descval \\
(exprs) & e & ::= & ... \mid librfun1_{\mathbf{P}}\ e \mid librfun2_{\mathbf{P}}\ e \\
(store) & \Sigma & ::= & ... \mid \Sigma, descval \rightarrow (v, \mathbf{P}) \\
(resource) & IO & ::= & . \mid IO, addrval \rightarrow (v, \mathbf{P}) \\
(mach\ sta) & M & ::= & (S; A; IO; \mathbf{P}; e) \\
(var\ env) & \Gamma & ::= & ... \mid \Gamma, addrval : addrtyp_p \mid \Gamma, descval : desctyp_p
\end{array}
$$

Figure 4.11: Requirements when Adding a Library to Source and Core Grammars

System Library Assumptions

Figure 4.11 demonstrates the requirements we assume are true when new libraries are added to the language. The top half of the figure describes the source language grammar, while the bottom half describes the core calculus grammar.

We will first talk about the source language. Addresses *addrval* are a collection of new constants that refer to an underlying resource, like a file system, and are associated with resource types *addrtyp*. We also require that the library creator define any necessary descriptor types with integrity level annotations. No constants of descriptor type should be added as they are not manipulated directly in the source language. Finally, we require that any library functions *librfun1* that create descriptors be annotated with the protection domain that the newly created resource

descriptors will have. Library functions *librfun2* that do not create descriptor require no such annotation.

Now we describe adding a library to the core calculus. In the core calculus, the resource type *addrtyp* has a protection domain annotation, because, where address values are used, the resource policy has been used to determine in which integrity level an address exists. The store Σ has been expanded to include descriptors, which are mapped to a value and the protection domain of the descriptor. For a mutable state library, the value in the store can be the contents of the reference. For a file I/O library, the value can be the name of the file that the descriptor points to.

Finally, we assume that any underlying resource, such as a file system, will be modeled in the core calculus as a store IO that maps address values to pairs $(\mathbb{P}, addrval)$. The core calculus value v is the contents of the resource at that address. The resource policy determines the protection domain $\mathbf{P}$ to which the address belongs. Note that though the resource store contains both the contents and the protection domain of an address, only the protection domain will be used by the translation algorithm. The contents, which are core calculus values, are only examined during core calculus execution.

From Interference Policy to Translation Rules

Figure 4.12 shows the new translation algorithm. First, any regular types, descriptor types, or descriptor values introduced by the library are translated in a straightforward way into their core language equivalents with no influence from the interference policy.

Next, the first two rules translate resource addresses and their associated resource types. A resource address *addrval* is translated in Rule pvt:io into a packed

Generated Type Translation $\vdash t \overset{\mathbf{typ}}{\Longrightarrow} t$

$$\frac{\mathsf{iotype}\ addrtyp \in pol}{\vdash addrtyp \overset{\mathbf{typ}}{\Longrightarrow} \exists r.\mathsf{S}_\rho \times addrtyp_{\mathbf{P}}}\ \mathsf{ptt{:}io}$$

Generated Value Translation $\Phi;\Gamma \vdash v : t \overset{\mathbf{val}}{\Longrightarrow} v'$

$$\frac{\mathcal{IO}(addrval) = (\_,\mathbf{P}) \qquad \mathsf{iotype}\ addrtyp \in pol}{\Phi;\Gamma \vdash addrval : addrtyp \overset{\mathbf{val}}{\Longrightarrow} \mathbf{pack}(\mathbf{P},(\mathbb{P},addrval))\ \mathbf{as}\ \exists\rho.\mathsf{S}_\rho \times addrtyp_\rho}\ \mathsf{pvt{:}io}$$

Generated Expression Translation $\Phi;\Gamma \vdash e : t \overset{\mathbf{exp}}{\Longrightarrow} e'$

$$\frac{\begin{array}{c}\mathsf{fun}\,f : t_1 \rightsquigarrow t_2\ [sta]\ [dyn] \in pol \qquad \Phi;\Gamma;\mathbf{P}_{curr} \vdash e : t_1 \overset{\mathbf{exp}}{\Longrightarrow} e' \\ \mathcal{STA}(sta) \qquad \mathcal{SPLIT}(t_1) = (\_,NONE) \qquad \mathcal{SPLIT}(t_2) = (\_,NONE)\end{array}}{\Phi;\Gamma;\mathbf{P}_{curr} \vdash f\ e : t_2 \overset{\mathbf{exp}}{\Longrightarrow} f\ e'}\ \mathsf{pet{:}none}$$

$$\frac{\begin{array}{c}\mathsf{fun}\,f : t_1 \rightsquigarrow t_2\ [sta]\ [dyn] \in pol \\ \Phi;\Gamma;\mathbf{P}_{curr} \vdash e : t_1 \overset{\mathbf{exp}}{\Longrightarrow} e' \qquad \vdash t_1 \overset{\mathbf{typ}}{\Longrightarrow} \tau_1 \qquad \vdash t_2 \overset{\mathbf{typ}}{\Longrightarrow} \tau_2 \qquad \mathcal{STA}(sta) \\ \mathcal{SPLIT}(t_1) = (argvars, SOME\ argvar) \qquad \mathcal{SPLIT}(t_2) = (\_,NONE)\end{array}}{\begin{array}{l}\Phi;\Gamma;\mathbf{P}_{curr} \vdash f\ e : t_2 \overset{\mathbf{exp}}{\Longrightarrow} \\ \quad \mathbf{let}\ argvars = e'\ \mathbf{in} \\ \quad \mathbf{open}\ (\rho,temp) = arg\ \mathbf{in} \\ \quad \mathbf{let}\ (x_{io},arg) = temp\ \mathbf{in} \\ \quad \mathcal{DYN}(dyn, f\ argvars, \mathbf{abort}_{t_2'})\end{array}}\ \mathsf{pet{:}arg}$$

$$\frac{\begin{array}{c}\mathsf{fun}\,f : t_1 \rightsquigarrow t_2\ [sta]\ [dyn] \in pol \\ \Phi;\Gamma;\mathbf{P}_{curr} \vdash e : t_1 \overset{\mathbf{exp}}{\Longrightarrow} e' \qquad \vdash t_1 \overset{\mathbf{typ}}{\Longrightarrow} \tau_1 \qquad \vdash t_2 \overset{\mathbf{typ}}{\Longrightarrow} \tau_2 \qquad \mathcal{STA}(sta) \\ \mathcal{SPLIT}(t_1) = (\_,NONE) \qquad \mathcal{SPLIT}(t_2) = (argvars, SOME\ argvar)\end{array}}{\begin{array}{l}\Phi;\Gamma;\mathbf{P}_{curr} \vdash f\ e : t_2 \overset{\mathbf{exp}}{\Longrightarrow} \\ \quad \mathbf{let}\ argvars = (f\ e')\ \mathbf{in} \\ \quad \mathbf{open}\ (\rho,temp) = arg\ \mathbf{in} \\ \quad \mathbf{let}\ (x_{io},arg) = temp\ \mathbf{in} \\ \quad \mathcal{DYN}(dyn, argvars, \mathbf{abort}_{t_2'})\end{array}}\ \mathsf{pet{:}res}$$

Figure 4.12: Algorithm from Interference Policy to Translation Rules

pair with a run-time protection domains $\mathbb{P}$, representing the result of a resource policy query on the IO store, and the actual core calculus address $addrval$. This existentially-wrapped value will be unwrapped at run time when the address is to be used by a descriptor creation operation. The run-time protection domain will be then be used in any dynamic checks required by the integrity policy.

Next, we translate library function calls in the last three rules. We will first discuss dynamic checks. There are three cases, determined by the result of the $\mathcal{SPLIT}$ function in Figure 4.13. The $\mathcal{SPLIT}$ function determines if a type is a resource type, or, if that type is a tuple, whether the resource type is included in that tuple. We can imagine a more complex analysis to find resource types within non-tuple complex types, but no libraries that we have examined required such complexity. Note that $\mathcal{SPLIT}'$ function is a helper function for the $\mathcal{SPLIT}$ function. In Rule pet:none, no resource types are in the argument type t_1 nor the result type t_2 of the library function; therefore, no dynamic checks are necessary.

In Rule pet:arg, there is a resource type in the argument type of the library function, while in Rule pet:res, there is a resource type in the result type of the library function. As described previously, resource address values are translated into a packed pair contained the run-time protection domain (symbolized the integrity level of the address as specified by the resource policy) and the core calculus address. In these two translation rules, the existential is first opened. Then then any dynamic checks specified by the interference policy are created by the $\mathcal{DYN}$ function. The $\mathcal{DYN}$ function, specified in Figure 4.13, uses **ifpd** expressions on the run-time protection domain of the address to enforce the required dynamic checks.

We have not generalized the algorithm for a case when a resource type appears in both the argument and result of a library function. We have not found any such

$\boxed{\mathcal{STA}(\overline{dir})}$

$$
\begin{array}{rcl}
\mathcal{STA}(\overline{dir}, \texttt{static n}) & = & \mathcal{STA}(\overline{dir}) \\
\mathcal{STA}(\overline{dir}, \texttt{static r } \mathbf{P}_{desc}) & = & \mathcal{STA}(\overline{dir}); \mathcal{L} \vdash \mathbf{P}_{curr} \leq \mathbf{P}_{desc} \\
\mathcal{STA}(\overline{dir}, \texttt{static w } \mathbf{P}_{desc}) & = & \mathcal{STA}(\overline{dir}); \mathcal{L} \vdash \mathbf{P}_{curr} \leq \mathbf{P}_{desc} \\
\mathcal{STA}(\overline{dir}, \texttt{static rw } \mathbf{P}_{desc}) & = & \mathcal{STA}(\overline{dir}); \mathcal{L} \vdash \mathbf{P}_{curr} \leq \mathbf{P}_{desc}; \mathcal{L} \vdash \mathbf{P}_{curr} \leq \mathbf{P}_{desc}
\end{array}
$$

$\boxed{\mathcal{DYN}(dir, e_1, e_2)}$

$$
\begin{array}{rcl}
\mathcal{DYN}(\texttt{dynamic n}, e_1, \_) & = & e_1 \\
\mathcal{DYN}(\texttt{dynamic r } \mathbf{P}_{desc}, e_1, e_2) & = & \mathbf{ifpd}(\mathbb{P}_{desc} \leq x_{io})\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2 \\
\mathcal{DYN}(\texttt{dynamic w } \mathbf{P}_{desc}, e_1, e_2) & = & \mathbf{ifpd}(x_{io} \leq \mathbb{P}_{desc})\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2 \\
\mathcal{DYN}(\texttt{dynamic rw } \mathbf{P}_{desc}, e_1, e_2) & = & \mathbf{ifpd}(\mathbb{P}_{desc} \leq x_{io})\,\mathbf{then} \\
 & & \quad (\mathbf{ifpd}(x_{io} \leq \mathbb{P}_{desc})\,\mathbf{then}\,e_1\,\mathbf{else}\,e_2)\,\mathbf{else}\,e_2
\end{array}
$$

$\boxed{\mathcal{SPLIT}'(t)}$

$$
\begin{array}{rcl}
\mathcal{SPLIT}'(SOME\ arg :: \overline{argopt}) & = & SOME\ arg \\
\mathcal{SPLIT}'(NONE :: \overline{argopt}) & = & \mathcal{SPLIT}'(\overline{argopt}) \\
\mathcal{SPLIT}'(.) & = & NONE
\end{array}
$$

$\boxed{\mathcal{SPLIT}(t)}$

$$
\begin{array}{rcl}
\mathcal{SPLIT}(t_1 \times ... \times t_n) & = & \mathbf{let}\ \overline{(argvars, argopt)} = \overline{\mathcal{SPLIT}(t_\rangle)} \\
 & & \mathbf{in}\ (\overline{argvars}, \mathcal{SPLIT}'(\overline{argopt})) \\
\mathcal{SPLIT}(t) & = & (arg, SOMEarg)\ if\ \mathsf{iotype}\ t \in pol \\
\mathcal{SPLIT}(t) & = & (arg, NONE)\ if\ \mathsf{iotype}\ t \notin pol
\end{array}
$$

Figure 4.13: Helper Functions for Algorithm from Interference Policy to Translation Rules

"Baked-in" Translation Rules from Previous Chapter

$$\frac{\Phi;\Gamma;p \vdash e_1 : \tau \overset{\mathbf{exp}}{\Longrightarrow} e_1' \qquad P;\Gamma, x:\tau\ \mathsf{ref}_p; p \vdash as; asps; e_2 : \tau' \overset{\mathbf{dec}}{\Longrightarrow} e_2'}{\Phi;\Gamma;p \vdash \texttt{ref}\ x\ \texttt{=}\ e_1\ as; asps; e_2 : \tau' \overset{\mathbf{dec}}{\Longrightarrow} \mathbf{let}\ x = \mathbf{ref}_p\ e_1'\ \mathbf{in}\ e_2'}\ \mathsf{sdt{:}ref}$$

$$\frac{\Phi;\Gamma;p \vdash e : \tau\ \mathsf{ref}_{p'} \overset{\mathbf{exp}}{\Longrightarrow} e' \qquad \vdash p \leq p'}{\Phi;\Gamma;p \vdash\ !e : \tau \overset{\mathbf{exp}}{\Longrightarrow} !e'}\ \mathsf{set{:}deref}$$

$$\frac{\Phi;\Gamma;p \vdash e_1 : \tau\ \mathsf{ref}_{p'} \overset{\mathbf{exp}}{\Longrightarrow} e_1' \qquad \Phi;\Gamma;p \vdash e_2 : \tau \overset{\mathbf{exp}}{\Longrightarrow} e_2' \qquad \vdash p' \leq p}{\Phi;\Gamma;p \vdash e_1 \texttt{:=} e_2 : 1 \overset{\mathbf{exp}}{\Longrightarrow} e_1'\ :=\ e_2'}\ \mathsf{set{:}asgn}$$

Expression Translation Generated by Algorithm

$$\frac{\Phi;\Gamma;\mathbf{P_{curr}} \vdash e : \tau \overset{\mathbf{exp}}{\Longrightarrow} e' \qquad \vdash \mathbf{P_{desc}} \leq \mathbf{P_{curr}}}{\Phi;\Gamma;\mathbf{P_{curr}} \vdash \texttt{ref}\ \mathbf{P_{desc}}\ e : \tau\ \mathsf{ref}_{\mathbf{P_{desc}}} \overset{\mathbf{exp}}{\Longrightarrow} \mathbf{ref}_{\mathbf{P_{desc}}}\ e'}\ \mathsf{pet{:}ref}$$

$$\frac{\Phi;\Gamma;\mathbf{P_{curr}} \vdash e : \tau\ \mathsf{ref}_{\mathbf{P_{desc}}} \overset{\mathbf{exp}}{\Longrightarrow} e' \qquad \vdash \mathbf{P_{curr}} \leq \mathbf{P_{desc}}}{\Phi;\Gamma;\mathbf{P_{curr}} \vdash\ !e : t \overset{\mathbf{exp}}{\Longrightarrow} !e'}\ \mathsf{pet{:}deref}$$

$$\frac{\Phi;\Gamma;\mathbf{P_{curr}} \vdash e : \tau\ \mathsf{ref}_{\mathbf{P_{desc}}} \times \tau \overset{\mathbf{exp}}{\Longrightarrow} e' \qquad \vdash \mathbf{P_{desc}} \leq \mathbf{P_{curr}}}{\Phi;\Gamma;\mathbf{P_{curr}} \vdash\ \texttt{:=}\ e : 1 \overset{\mathbf{exp}}{\Longrightarrow} :=\ e'}\ \mathsf{pet{:}asgn}$$

Figure 4.14: Translation Rules Generated by Interference Policy for Reference Library

functions in the example libraries we have studied, and believe it would needlessly complicate our algorithm to include this case.

Finally, any static checks specified by the interference policy are created by the $\mathcal{STA}$ function. The $\mathcal{STA}$ function, specified in Figure 4.13, enforces the specified *none* (`n`), *read-up* (`r`), *write-down* (`w`), and *read/write* (`rw`) policies with the relevant protection domain lattice order judgments. Notice that several static checks can be enforced on a library function by a policy.

An example of using the algorithm in Figures 4.12 and 4.13 to create translation rules from an interference policy is shown in Figure 4.4. The top half of the figure displays the "baked-in" static checks for reference operations from the previous chapter. The bottom half of the figure contains the translation rules for a mutable reference library that were generated by running the algorithm on the example interference policy for mutable references in Figure 4.4. The static checks generated by the algorithm in Rules pet:deref and pet:asgn correspond to the "baked-in" static checks in Rules set:deref and set:asgn from the previous chapter. That is because the interference policy matches the "baked-in" harmless operations for reference assignment and dereference.

However, notice that the static checks generated by the algorithm in Rule pet:ref does not match the "baked-in" static check in Rule sdt:ref from the previous chapter. Recall that in the previous chapter, to preserve syntactic separation between the main program and aspects, created references were given the protection domain that they were created in. Mutable references created by the main program were placed in the **MAIN** protection domain, while references created by an aspect $ASPECT$ were placed in the **ASPECT** protection domain. In contrast, in the example interference policy in Figure 4.4, we decided to allow a less restrictive policy that should still preserve harmlessness. The result of this less restrictive interference policy is that mainline code can use this mutable reference library to create references in the protection domain of an aspect. However, some of the separation between the main program and aspects has been lost, making the program more confusing to understand. Allowing the programmer to specify these trade-offs is an example of the flexibility allowed by the interference policy system.

The mutable reference library does not require a policy with resource types, resource policies, or dynamic checks. As such, it uses only the simplest features of the translation generation algorithm. In the next section, we shall more fully explore a case study of the file I/O library to demonstrate these advanced policy features.

4.5 Case Study of File I/O

We now provide a case study to demonstrate the usefulness of our interference policy system. We first add an idealized file I/O system library to the source language, and we design a corresponding interference policy for the library and resource policy for the underlying file system. We then show that the translation created by the interference policy is type-safe with reference to hand-coded, non-interfering core calculus file I/O operations (with baked-in protection domain checks). We will prove that a well-typed source program (with its general interference policy) translates to a well-typed core calculus expression (with its baked-in policy). The interference policy thus enforces "harmlessness" on advice that uses our source language file I/O library.

In this manner, we show that, if the user so desires, interference policies can be useful to enforce our previous definition of harmlessness on system libraries. However, this should not be understood as providing a general result that all interference policies enforce the harmlessness of advice. Recall that in the general case, the purpose of interference policies is to allow the user to enforce the levels of "harmlessness" or "harmfulness" on advice that they desire, not just to enforce the particular harmlessness result that we defined in the previous chapter.

(*types*)	t	::=	... | `infile` **P** | `outfile` **P** | `fname`
(*vals*)	v	::=	... | *'f'*
(*exprs*)	e	::=	... | `openr` **P** e | `openw` **P** e
		|	`read` e | `write` e
(*main decl*)	d	::=	... | `infile` x `=` e | `outfile` x `=` e

Figure 4.15: Idealized File I/O Library in HarmlessAML

```
fname : {   (* argtype ~> restype        [statchecks]  [dynchecks]*)
  fun read : infile P ~> string          [static r P]  [dynamic n]
  fun write:(outfile P * string) ~> unit [static rw P] [dynamic n]
  fun openr P : fname ~> infile P        [static w P]  [dynamic r P]
  fun openw P : fname ~> outfile P       [static w P]  [dynamic w P]
}
```

Figure 4.16: Example Interference Policy for Idealized File I/O Library

4.5.1 Idealized File I/O Library in HarmlessAML

We first reiterate our idealized file I/O library, previously used as an example in Section 4.2.1 The syntax, displayed here in Figure 4.15, is based on a simplified version of the file I/O in Standard ML. Our library follows the assumptions required of system libraries described in Section 4.4. As before, filenames, *'f'* are given the resource type `fname` type. The files that these filenames describe can be opened for reading and writing with the `openr` **P** e and `openw` **P** e commands respectively. The resulting input and output file descriptors are given descriptor types `infile` **P** and `outfile` **P**. The `read` e expression reads from an input file descriptor, while the `write` e expression receives a tuple containing an output file descriptor and a string and writes the string to that output file descriptor

$$
\begin{array}{llll}
 & i, o \in \text{File Descriptors} & & f \in \text{Strings} \\
(types) & \tau & ::= & \ldots \mid \mathsf{infile}_p \mid \mathsf{outfile}_p \mid \mathsf{fname}_p \\
(values) & v & ::= & \ldots \mid i \mid o \mid f \\
(exprs) & e & ::= & \ldots \mid \mathbf{openr}_p\, e \mid \mathbf{read}\, e \\
 & & \mid & \mathbf{openw}_p\, e \mid \mathbf{write}\, e \\
(store) & \Sigma & ::= & \ldots \mid \Sigma, i \to (f, \mathbf{P}) \mid \Sigma, o \to (f, \mathbf{P}) \\
(file\ sys) & F & ::= & . \mid F, f \to (s, \mathbf{P}) \\
(mach\ sta) & M & ::= & (\Sigma; A; F; \mathbf{P}; e) \\
(var\ env) & \Gamma & ::= & \ldots \mid \Gamma, i : \mathsf{infile}_p \mid \Gamma, o : \mathsf{outfile}_p \\
 & & \mid & \Gamma, f : \mathsf{fname}_p
\end{array}
$$

Figure 4.17: Noninterfering Idealized File I/O Extension to $\mathbb{F}_{HRM}$ Syntax

4.5.2 File I/O Interference Policy

We describe the interference policy for our file I/O library, first used as an example policy in Section 4.2.2, and reiterated here as Figure 4.16. As before, we use static checks to describe how each file system library operation will use its file descriptors. The `read` and `write` operations do not require dynamic checks (`dynamic n`) as they do not manipulate filenames. As such, static checks will be enough to ensure that their use of file descriptors does not cause aspects to interfere with the main program. However, we have inserted dynamic checks on the `openr` and `openw` operations to ensure that the files they open are not in a region of the file system that could cause interference.

4.5.3 Noninterfering Idealized File I/O Library in $\mathbb{F}_{HRM}$

To show that the interference policy for our file I/O library is versatile enough to preserve a harmlessness property on the source language, we present hand-coded, noninterfering core calculus file I/O operations. The syntax of this core calculus library is presented in Figure 4.17. We represent the underlying file system resource

$\boxed{\beta\text{-reduction } (\Sigma; A; F; \mathbf{P}, e) \longmapsto_\beta (\Sigma'; A'; F'; \mathbf{P}; e')}$

$$\frac{F(f) = (s, P'') \qquad \mathcal{L} \vdash \mathbf{P}' \leq \mathbf{P} \qquad \mathcal{L} \vdash \mathbf{P}' \leq \mathbf{P}''}{\begin{array}{c}(\Sigma; A; F; \mathbf{P}; \mathbf{openr}_{\mathbf{P}'}\, f) \longmapsto_\beta \\ (\Sigma, i \rightarrow (f, \mathbf{P}'); A; F; \mathbf{P}; i)\end{array}}\ \textsf{ceb:openr}$$

$$\frac{\Sigma(i) = (f, \mathbf{P}') \qquad F(f) = (s, \mathbf{P}'') \qquad \Pi \vdash \mathbf{P} \leq \mathbf{P}' \leq \mathbf{P}''}{(\Sigma; A; F; \mathbf{P}; \mathbf{read}\, i) \longmapsto_\beta (\Sigma; A; F; \mathbf{P}; s)}\ \textsf{ceb:read}$$

$$\frac{F(f) = (s, \mathbf{P}'') \qquad \mathcal{L} \vdash \mathbf{P}'' \leq \mathbf{P}' \leq \mathbf{P}}{\begin{array}{c}(\Sigma; A; F; \mathbf{P}; \mathbf{openw}_{\mathbf{P}'}\, f) \longmapsto_\beta \\ (\Sigma, o \rightarrow (f, \mathbf{P}'); A; F; \mathbf{P}; o)\end{array}}\ \textsf{ceb:openw}$$

$$\frac{\Sigma(o) = (f, P) \qquad F(f) = (s', \mathbf{P}') \qquad \mathcal{L} \vdash \mathbf{P}' \leq \mathbf{P}}{\begin{array}{c}(\Sigma; A; F; \mathbf{P}; \mathbf{write}\, (o, s)) \longmapsto_\beta \\ (\Sigma; A; F, f \rightarrow (s, \mathbf{P}'); \mathbf{P}; ())\end{array}}\ \textsf{ceb:write}$$

Figure 4.18: Noninterfering Idealized File I/O Extensions to $\mathbb{F}_{HRM}$ β-redex Operational Semantics

in the core calculus as a file store F. An element of the file store, $f \rightarrow (s, \mathbf{P})$, maps a filename f to a string s that represents the contents of the file and the protection domain $\mathbf{P}$ in which the resource policy places the files. As a side note, we duplicate the resource policy (in addition to the run-time protection domains created during translating the source language to the core calculus) here in order to later verify that the run-time protection domains are in fact enforcing the required noninterference policy. We assume that all files that can and will be manipulated are represented in the file store F.

As in the source language, the core calculus **openr** (and **openw**) expressions create a input (or output) file descriptor that can be used to read a file. This file descriptor is then placed in the store Σ. When reading from a input file descriptor, the input file descriptor is used to look up the file contents in the file store F. When

Well-formed Values $\Delta; \Pi; \Gamma \vdash v : \tau$

$$\frac{\Gamma(i) = \mathsf{infile}_p}{\Delta; \Pi; \Gamma \vdash i : \mathsf{infile}_p}\ \mathsf{cvt{:}infd} \qquad \frac{\Gamma(i) = \mathsf{outfile}_p}{\Delta; \Pi; \Gamma \vdash o : \mathsf{outfile}_p}\ \mathsf{cvt{:}outfd}$$

$$\frac{\Gamma(f) = \mathsf{fname}_p}{\Delta; \Pi; \Gamma \vdash f : \mathsf{fname}_p}\ \mathsf{cvt{:}fname}$$

Well-formed Expressions $\Delta; \Pi; \Gamma; p \vdash e : \tau$

$$\frac{\Delta \vdash p' \qquad \Pi \vdash p' \leq p \qquad \Pi \vdash p' \leq p'' \qquad \Delta; \Pi; \Gamma; p \vdash e : \mathsf{fname}_{p''}}{\Delta; \Pi; \Gamma; p \vdash \mathbf{openr}_{p'}\, e : \mathsf{infile}_{p'}}\ \mathsf{cet{:}openr}$$

$$\frac{\Pi \vdash p \leq p' \qquad \Delta; \Pi; \Gamma; p \vdash e : \mathsf{infile}_{p'}}{\Delta; \Pi; \Gamma; p \vdash \mathbf{read}\, e : \mathsf{string}}\ \mathsf{cet{:}read}$$

$$\frac{\Delta \vdash p' \qquad \Delta \vdash p'' \leq p' \leq p \qquad \Delta; \Pi; \Gamma; p \vdash e : \mathsf{fname}_{p''}}{\Delta; \Pi; \Gamma; p \vdash \mathbf{openw}_{p'}\, e : \mathsf{outfile}_{p'}}\ \mathsf{cet{:}openw}$$

$$\frac{\Delta; \Pi; \Gamma; p \vdash e : \mathsf{outfile}_p \times \mathsf{string}}{\Delta; \Pi; \Gamma; p \vdash \mathbf{write}\, e : 1}\ \mathsf{cet{:}write}$$

Figure 4.19: Noninterfering Idealized File I/O Extensions to $\mathbb{F}_{HRM}$ Value and Expression Typing Rules

writing a file, the output file descriptor is used to index the correct file in the store—the contents of that file are then replaced by the new string. These operational semantics are formalized in Rules ceb:openread and ceb:readfile (or ceb:openwrite and ceb:writefile) in Figure 4.18.

We must hand-code the required protection domain checks into the core calculus typing rules of the file I/O expressions. These typing rules are presented in Figure 4.19. The first three rules describe the typing of input file descriptors, output file descriptors, and filenames. The protection domain given to a fname type is implicitly set by the resource policy when the initial environment Γ is

Well-formed Contexts $\Delta \vdash \Gamma$

$$\frac{\forall x, r, l, i, o, f \in dom(\Gamma).\Delta \vdash \Gamma(x, r, l, i, o, f)}{\Delta \vdash \Gamma}\ \textsf{cg:g}$$

$F \vdash \Sigma : \Gamma$

$$\frac{\begin{array}{c} dom(\Gamma) = dom(\Sigma) \cup dom(F) \\ \forall r \in dom(\Sigma) \quad \Sigma(r) = (v, \mathbf{P}) \quad \Gamma(r) = \tau\ \mathsf{ref}_{\mathbf{P}} \quad \Gamma \vdash v : \tau \\ \forall l \in dom(\Sigma) \quad \Sigma(l) = (t, \mathbf{P}) \quad \Gamma(r) = \tau\ \mathsf{label}_{\mathbf{P}} \quad . \vdash \tau \\ \forall i \in dom(\Sigma) \quad \Sigma(i) = (f, \mathbf{P}) \quad \Gamma(i) = \mathsf{infile}_{\mathbf{P}} \\ F(f) = (s, \mathbf{P}') \quad \mathcal{L} \vdash P \leq P' \\ \forall o \in dom(\Sigma) \quad \Sigma(o) = (f, \mathbf{P}) \quad \Gamma(o) = \mathsf{outfile}_{\mathbf{P}} \\ F(f) = (s, \mathbf{P}') \quad \mathcal{L} \vdash P' \leq P \\ \forall f \in dom(F) \quad F(f) = (s, P) \quad \Gamma(f) = \mathsf{fname}_{\mathbf{P}} \end{array}}{F \vdash \Sigma : \Gamma}\ \textsf{camt:amt}$$

Well-formed Machine States $\vdash (\Sigma; A; F; \mathbf{P}; e)$ **ok**

$$\frac{F \vdash \Sigma : \Gamma \qquad \Gamma \vdash A\ \mathbf{ok} \qquad .; \mathcal{L}; \Gamma; \mathbf{P} \vdash e : \tau\ \mathbf{for\ some}\ \tau}{\vdash (\Sigma; A; F; \mathbf{P}; e)\ \mathbf{ok}}\ \textsf{camt:nms}$$

Figure 4.20: Noninterfering Idealized File I/O Extensions to $\mathbb{F}_{HRM}$ Machine State Well-formedness Rules

$$
\begin{array}{llcl}
(\textit{simul. fds}) & f^2 & ::= & f \mid \texttt{<}f \texttt{|} \textit{void}\texttt{>} \mid \texttt{<}\textit{void} \texttt{|} f\texttt{>} \\
(\textit{simul. files}) & s^2 & ::= & s \mid \texttt{<}s \texttt{|} s\texttt{>} \\
(\textit{store}) & S & ::= & \ldots \mid S, i \rightarrow (f^2, \mathbf{P}) \mid S, o \rightarrow (f^2, \mathbf{P}) \\
(\textit{file store}) & F & ::= & \cdot \mid F, f \rightarrow (s^2, \mathbf{P})
\end{array}
$$

Figure 4.21: Noninterfering Idealized File I/O Extensions to $\mathbb{F}_{HRM2}$ Grammar

created in Rule camt:amt. Rules cet:openread and cet:readfile use protection domain checks to enforce a "read-up" noninterference property. Similarly, cet:openwrite and cet:writefile) enforce a "write-down" noninterference property. Rule camt:amt in Figure 4.20 places the types of input file descriptors and output file descriptor from store Σ into the initial environment Γ. As described earlier, the rule also implicitly uses the resource policy to set the protection domain that annotated the fname types for filenames in the file store F.

4.5.4 Proving $\mathbb{F}_{HRM}$ File I/O Noninterference Theorem

We now prove a noninterference property for our core calculus file I/O library. We use the now-familiar proof technique of defining a new calculus $\mathbb{F}_{HRM2}$ that simulates the simultaneous execution of two $\mathbb{F}_{HRM}$. The $\mathbb{F}_{HRM2}$ grammar, described in Figure 4.21, has two main additions. First, file descriptors i and o can now be created in one of the simultaneously executing expressions. As such, file descriptors point in the updated store Σ to a simultaneous filename f^2. If both simultaneously executing expressions created the file descriptor, then f^2 will be a single filename f. If the left (or right) expression created the file descriptor, then the descriptor will point to <f | *void*> (or <*void* | f>) to indicate that situation. For example, if an input file descriptor was defined only in the first of the two simultaneously executing $\mathbb{F}_{HRM}$ programs, **LOW<openr$_{\mathbf{LOW}}$** '*temp.log*' | ()>, then the resulting store would

contain $o \rightarrow \langle \text{`}temp.log\text{'} | void \rangle$. Note that we require that the same file descriptor cannot be independently created in the left and right expressions. Therefore, we can omit a potential $\texttt{<}f_1 \texttt{|} f_2\texttt{>}$ case for simultaneous filenames.

Similarly, the contents of the file store F are now represented by simultaneous strings s^2. If only the left (or right) expression changes the file store, then the divided nature of the file is represented by $\texttt{<}s_1 \texttt{|} s_2\texttt{>}$. Note that since all files that can be used by a program are located in the file store F, we do not need to consider the case where a file only exists in the left (or right) of the simultaneously executing program. Therefore, we can omit the potential $\texttt{<}s \texttt{|} void\texttt{>}$ and $\texttt{<}void \texttt{|} s\texttt{>}$ cases.

Figures 4.5.4 and 4.5.4 describe how to construct an initial environment Γ from a store Σ and a file store F. Notice that if the file descriptor (or file contents) have been created (or changed) by a low-integrity expression, the protection domain that annotates the file descriptor (or filename) type must be a low-protection domain. The typing rules for the rest of the new $\mathbb{F}_{HRM2}$ features are described in Figure 4.5.4.

Figure 4.5.4 describes the operational semantics of the new $\mathbb{F}_{HRM2}$ features. Mainly, creating file descriptors in a low-protection expression only changes the relevant side of the store Σ. Similarly, reading and writing to a file in a low-protection expression only reads or writes to the relevant side of the file store F.

We have defined the $\mathbb{F}_{HRM2}$ calculus—we now wish to prove that it correctly simulates the simultaneous execution of two $\mathbb{F}_{HRM}$ programs. Since the $\mathbb{F}_{HRM2}$ calculus of this chapter is an expansion of the calculus of the previous chapter, the following is an expansion of the proofs in the previous chapter.

Well-formed Input File Descriptions $\vdash_2 (f, \mathbf{P}) \stackrel{\mathbf{i}}{\Longrightarrow} \mathsf{infile}_P$

$$\frac{F(f) = (s^2, \mathbf{P}') \qquad \mathcal{L} \vdash \mathbf{P} \leq \mathbf{P}'}{\vdash_2 (f, \mathbf{P}) \stackrel{\mathbf{i}}{\Longrightarrow} \mathsf{infile}_{\mathbf{P}}} \text{ c2st:infile}$$

$$\frac{F(f) = (s^2, \mathbf{P}') \qquad \mathcal{L} \vdash \mathbf{P} \leq \mathbf{P}' \qquad \mathbf{P} \in L}{\vdash_2 (\mathsf{<}f \,|\, void\mathsf{>}, \mathbf{P}) \stackrel{\mathbf{i}}{\Longrightarrow} \mathsf{infile}_{\mathbf{P}}} \text{ c2st:infile1}$$

$$\frac{F(f) = (s^2, \mathbf{P}') \qquad \mathcal{L} \vdash \mathbf{P} \leq \mathbf{P}' \qquad \mathbf{P} \in L}{\vdash_2 (\mathsf{<}void \,|\, f\mathsf{>}, \mathbf{P}) \stackrel{\mathbf{i}}{\Longrightarrow} \mathsf{infile}_{\mathbf{P}}} \text{ c2st:infile2}$$

Well-formed Output File Descriptors $\vdash_2 (f, \mathbf{P}) \stackrel{\mathbf{o}}{\Longrightarrow} \mathsf{outfile}_P$

$$\frac{F(f) = (s^2, \mathbf{P}') \qquad \mathcal{L} \vdash \mathbf{P}' \leq \mathbf{P}}{\vdash_2 (f, \mathbf{P}) \stackrel{\mathbf{o}}{\Longrightarrow} \mathsf{outfile}_{\mathbf{P}}} \text{ c2st:outfile}$$

$$\frac{F(f) = (s^2, \mathbf{P}') \qquad \mathcal{L} \vdash \mathbf{P}' \leq \mathbf{P} \qquad \mathbf{P} \in L}{\vdash_2 (\mathsf{<}f \,|\, void\mathsf{>}, \mathbf{P}) \stackrel{\mathbf{i}}{\Longrightarrow} \mathsf{outfile}_{\mathbf{P}}} \text{ c2st:outfile1}$$

$$\frac{F(f) = (s^2, \mathbf{P}') \qquad \mathcal{L} \vdash \mathbf{P}' \leq \mathbf{P} \qquad P \in L}{\vdash_2 (\mathsf{<}void \,|\, f\mathsf{>}, \mathbf{P}) \stackrel{\mathbf{i}}{\Longrightarrow} \mathsf{outfile}_{\mathbf{P}}} \text{ c2st:outfile2}$$

Well-formed Filenames $\vdash_2 (s, \mathbf{P}) \stackrel{\mathbf{f}}{\Longrightarrow} \mathsf{fname}_P$

$$\frac{}{\vdash_2 (s, \mathbf{P}) \stackrel{\mathbf{f}}{\Longrightarrow} \mathsf{fname}_{\mathbf{P}}} \text{ c2ft:fname} \qquad \frac{P \in L}{\vdash_2 (\mathsf{<}s_1 \,|\, s_2\mathsf{>}, \mathbf{P}) \stackrel{\mathbf{f}}{\Longrightarrow} \mathsf{fname}_{\mathbf{P}}} \text{ c2ft:fname12}$$

Figure 4.22: $\mathbb{F}_{HRM2}$ Extensions to Store Typing Rules

Well-formed Stores $F \vdash_2 \Sigma : \Gamma$

$$\frac{\begin{array}{c} dom(\Gamma) = dom(\Sigma) \cup dom(F) \\ \forall r \in dom(\Sigma).\ \Gamma \vdash_2 \Sigma(r) \stackrel{\mathbf{r}}{\Longrightarrow} \Gamma(r) \qquad \forall l \in dom(\Sigma).\ \Gamma \vdash_2 \Sigma(l) \stackrel{\mathbf{l}}{\Longrightarrow} \Gamma(l) \\ \forall i \in dom(\Sigma).\ \Gamma \vdash_2 \Sigma(i) \stackrel{\mathbf{i}}{\Longrightarrow} \Gamma(i) \qquad \forall o \in dom(\Sigma).\ \Gamma \vdash_2 \Sigma(o) \stackrel{\mathbf{o}}{\Longrightarrow} \Gamma(o) \\ \forall f \in dom(F).\ \Gamma \vdash_2 F(f) \stackrel{\mathbf{f}}{\Longrightarrow} \Gamma(f) \end{array}}{F \vdash_2 \Sigma : \Gamma}\ \textsf{c2sft}$$

Figure 4.23: $\mathbb{F}_{HRM2}$ Extensions to Store Typing Rules

Well-formed Expressions $\Pi; \Gamma; p \vdash_2 e : \tau$

$$\frac{\Gamma; p' \vdash_2 e_1 : 1 \qquad \Gamma; p' \vdash_2 e_2 : 1 \qquad p \in H \qquad p' \in L \qquad \vdash_2 p' \leq p}{\Gamma; p \vdash_2 p'\texttt{<}e_1 \mid e_2\texttt{>} : 1}\ \textsf{c2et:highlow}$$

Well-formed Machine States $\vdash_2 (\Sigma; A; F; \mathbf{P}; e)_i$ **ok**

$$\frac{\begin{array}{c} F \vdash_2 \Sigma : \Gamma \qquad \Gamma \vdash_2 A\ \mathbf{ok} \qquad .; \mathcal{L}; \Gamma; \mathbf{P} \vdash_2 e : \tau\ \mathbf{for\ some}\ \tau \\ i \in \{0\} \Rightarrow \mathbf{P} \in H \qquad i \in \{1, 2\} \Rightarrow \mathbf{P} \in L \end{array}}{\vdash_2 (\Sigma; A; F; \mathbf{P}; e)_i\ \mathbf{ok}}\ \textsf{c2amt:nms}$$

Figure 4.24: Idealized File I/O Extensions to $\mathbb{F}_{HRM2}$ Typing Rules

$\boxed{\beta\text{-reduction } (\Sigma; A; F; \mathbf{P}, e) \longmapsto_{2,\beta} (\Sigma'; A'; F'; \mathbf{P}; e')}$

$$\frac{F(f) = (s, P'') \qquad \mathcal{L} \vdash_2 \mathbf{P}' \leq \mathbf{P} \qquad \mathcal{L} \vdash_2 \mathbf{P}' \leq \mathbf{P}''}{\begin{array}{c}(\Sigma; A; F; \mathbf{P}; \mathbf{openr}_{\mathbf{P}'}\ f)_i \longmapsto_{2,\beta} \\ (\Sigma, i \to (new_i\ f, \mathbf{P}'); A; F; \mathbf{P}; i)_i\end{array}}\ \textsf{c2eb:openr}$$

$$\frac{\begin{array}{c}\Sigma(i) = (f^2, \mathbf{P}') \\ f = read_i\ f^2 \qquad F(f) = (s^2, \mathbf{P}'') \qquad \mathcal{L} \vdash_2 \mathbf{P} \leq \mathbf{P}' \leq \mathbf{P}''\end{array}}{(\Sigma; A; F; \mathbf{P}; \mathbf{read}\ i)_i \longrightarrow_{2,\beta} (\Sigma; A; F; \mathbf{P}; read_i\ s^2)}\ \textsf{c2eb:read}$$

$$\frac{F(f) = (s^2, \mathbf{P}'') \qquad \mathcal{L} \vdash_2 \mathbf{P}'' \leq \mathbf{P}' \leq \mathbf{P}}{\begin{array}{c}(\Sigma; A; F; \mathbf{P}; \mathbf{openw}_{\mathbf{P}'}\ f)_i \longmapsto_{2,\beta} \\ (\Sigma, o \to (new_i\ f, \mathbf{P}'); A; F; \mathbf{P}; o)_i\end{array}}\ \textsf{c2eb:openw}$$

$$\frac{\Sigma(o) = (f^2, \mathbf{P}) \qquad f = read_i\ f^2 \qquad F(f) = (s^2, \mathbf{P}') \qquad \mathcal{L} \vdash_2 \mathbf{P}' \leq \mathbf{P}}{\begin{array}{c}(\Sigma; A; F; \mathbf{P}; \mathbf{write}\ (o, s))_i \longmapsto_{2,\beta} \\ (\Sigma; A; F, f \to (upd_i\ s^2\ s, \mathbf{P}'); \mathbf{P}; ())_i\end{array}}\ \textsf{c2eb:write}$$

Figure 4.25: Idealized File I/O Extensions to $\mathbb{F}_{HRM2}$ β-redex Operational Semantics

Soundness. The Soundness Theorem 4.5.3 states that if a $\mathbb{F}_{HRM2}$ expression takes a step, then the two corresponding $\mathbb{F}_{HRM}$ programs (the projections of the $\mathbb{F}_{HRM2}$ expression) must each take the same respective steps.

Lemma 4.5.1 (Expression Soundness Lemma) *For* $i \in \{1, 2\}$,
if $(\Sigma, A, F, \mathbf{P}, e)_i \longmapsto_{2,\beta} (\Sigma', A', F, \mathbf{P}, e')_i$,
then $(|\Sigma|_i, |A|_i, |F|_i, \mathbf{P}, e) \longmapsto_{\beta} (|\Sigma'|_i, |A'|_i|F'|_i, \mathbf{P}, e')$.

Proof: By induction on the structure of $(\Sigma, A, F, \mathbf{P}, e)_i \longmapsto_{2,\beta} (\Sigma', A', F', \mathbf{P}, e')_i$.

□

Lemma 4.5.2 (Beta-redex Soundness Lemma) *For* $i \in \{1, 2\}$,
if $(\Sigma; A; F; \mathbf{P}; e) \longmapsto^*_{2,\beta} (\Sigma'; A'; F'; \mathbf{P}; e')$
then $|(\Sigma; A; F; \mathbf{P}; e)|_i \longmapsto^*_{\beta} |(\Sigma'; A'; F'; \mathbf{P}; e')|_i$

Proof: By induction on the structure of the operational judgment $(\Sigma, A, F, \mathbf{P}, e) \longmapsto^*_{2,\beta} (\Sigma', A', F', \mathbf{P}, e')$, with use of the Expression Soundness Lemma 4.5.1. □

Theorem 4.5.3 (Soundness) *For* $i \in \{1, 2\}$,
if $(\Sigma; A; F; \mathbf{P}; e) \longmapsto^*_{2,top} (\Sigma'; A'; F'; \mathbf{P}; e')$
then $|(\Sigma; A; F; \mathbf{P}; e)|_i \longmapsto^*_{top} |(\Sigma'; A'; F'; \mathbf{P}; e')|_i$

Proof: By induction on the structure of the operational judgment $(\Sigma, A, F, \mathbf{P}, e) \longmapsto^*_{2,top} (\Sigma', A', F', \mathbf{P}, e')$.

- Case ce:beta uses Lemma 4.5.2.

□

Completeness. The Completeness Theorem 4.5.5 states that if two $\mathbb{F}_{HRM}$ programs step to values, then the representation in $\mathbb{F}_{HRM2}$ that simulates them simultaneously must step to a value. The completeness theorem requires an auxiliary lemma stating that a $\mathbb{F}_{HRM2}$ program is only stuck when one of its corresponding $\mathbb{F}_{HRM}$ programs are stuck.

Lemma 4.5.4 (Completeness Stuck Lemma) *Assume* $(\Sigma, A, F, \mathbf{P}, e)$ *is stuck. Then* $|(\Sigma, A, F, \mathbf{P}, e)|_i$ *is stuck for some* $i \in \{1, 2\}$.

Proof: Proof by induction on the structure of e. □

Theorem 4.5.5 (Completeness) *Assume*
$|(\Sigma; A; F; \mathbf{P}; e)|_i \longmapsto^*_{top} (\Sigma'_i; A'_i; F'_i; \mathbf{P}; v_i)$ *for all* $i \in 1, 2$ *then there exists*
$(\Sigma'; A'; F'_i; \mathbf{P}; v)$ *such that* $(\Sigma; A; F; \mathbf{P}; e) \longmapsto^*_{2,top} (\Sigma'; A'; F'; \mathbf{P}; v)$

Proof: If $(\Sigma, A, F, \mathbf{P}, e)$ yields an infinite reduction sequence, then $|(\Sigma, A, F, \mathbf{P}, e)|_i$ yields a infinite reduction sequence by Theorem 4.5.3.
If $(\Sigma, A, F, \mathbf{P}, e)$ is stuck, then $|(\Sigma, A, F, \mathbf{p}, e)|_i$ is stuck by Lemma 4.5.4.
Therefore, $(\Sigma, A, F, \mathbf{P}, e)$ reduces to a successful configuration. □

Progress. In this section, we present the lemmas used to prove Progress Theorem 4.5.16.

Lemma 4.5.6 (Canonical Forms) *Suppose* $\cdot; \mathcal{L}; \cdot \vdash_2 \quad v : \tau$ *is a closed, well-formed value.*

- *If* $\tau = \mathsf{S}_{\mathbf{P}}$, *then* $v = \mathbb{P}$.
- *If* $\tau = \mathsf{infile}_{\mathbf{P}}$, *then* $v = i$.

- *If* $\tau = \mathsf{outfile}_{\mathbf{P}}$, *then* $v = o$.
- *If* $\tau = \exists\rho.\tau$, *then* $v = \mathbf{pack}(\mathbf{P}, v)$ **as** $\exists\rho.\tau$.
- *If* $\tau = \mathsf{fname}_p$, *then* $v = f$.

Proof: By induction on the structure of $\Delta; \Pi; \Gamma \vdash v : \tau$, using the fact that v is a value. □

Lemma 4.5.7 (Σ to Well-formed Γ) *If* $F \vdash_2 \Sigma : \Gamma$, *then* $\vdash_2 \cdot : \Gamma$.

Proof: Trivial using Rule c2sft. □

Lemma 4.5.8 (Reverse Evaluation Contexts) *If* $\cdot; \mathcal{L}; \Gamma; \mathbf{P} \vdash_2 E[e] : \tau$, *then* $\cdot; \mathcal{L}; \Gamma; \mathbf{P} \vdash_2 e : \tau'$ *for some* τ'.

Proof: By induction on the structure of evaluation contexts E. □

Lemma 4.5.9 (Empty Δ) *If* $\cdot \vdash_2 p$, *then* p *is some* $\mathbf{P}$.

Proof: By induction on the structure of the typing judgment $\Delta \vdash_2 p$. □

Lemma 4.5.10 (Γ to Σ)

- *If* $F \vdash_2 \Sigma : \Gamma$ *and* $\Gamma(i) = \mathsf{infile}_{\mathbf{P}}$, *then* $\Sigma(i) = f^2$ *for some* f^2.
- *If* $F \vdash_2 \Sigma : \Gamma$ *and* $\Gamma(o) = \mathsf{outfile}_{\mathbf{P}}$, *then* $\Sigma(o) = f^2$ *for some* f^2.

Proof: Trivial using Rule c2sft. □

Lemma 4.5.11 (Protection Domain Weakening) *If* $\Delta \vdash_2 p$, *then* $\Delta, \rho \vdash_2 p$.

Proof: By induction on the structure of the typing rule $\Delta \vdash_2 p$. □

Lemma 4.5.12 (Protection Domain Reverse Substitution) *If* $\Delta \vdash_2 p[p'/\rho]$ *and* $\Delta \vdash_2 p'$, *then* $\Delta, \rho \vdash_2 p$.

Proof: By induction on the structure of the typing rule $\Delta \vdash_2 p$ with the Protection Domain Weakening Lemma 4.5.11. □

Lemma 4.5.13 (Type Reverse Substitution) *If* $\Delta \vdash_2 \tau[p/\rho]$ *and* $\Delta \vdash_2 p$, *then* $\Delta, \rho \vdash_2 \tau$.

Proof: By induction on the structure of the typing rule $\Delta \vdash_2 \tau$ with the Protection Domain Reverse Substitution Lemma 4.5.12. □

Lemma 4.5.14 (Well-formed types) *If* $\Delta \vdash_2 \Gamma$ *and* $\Delta; \Pi; \Gamma; p \vdash_2 e : \tau$, *then* $\Delta \vdash_2 \tau$.

Proof: By induction on the structure of the typing rule $\Delta; \Pi; \Gamma; p \vdash_2 e : \tau$.

- Case cvt:pack uses Lemma 4.5.13.

□

Lemma 4.5.15 (Progress Lemma) *If* $F \vdash_2 \Sigma : \Gamma$ *and* $\Gamma \vdash_2 A$ *ok and* $\cdot; \mathcal{L}; \Gamma; \mathbf{P} \vdash_2 e : \tau$ *and*

- *If* $i \in \{o\}$, *then* $\mathbf{P} \in \mathcal{HIGH}$.
- *If* $i \in \{1, 2\}$, *then* $\mathbf{P} \in \mathcal{LOW}$.

then either e *is a value or* $(\Sigma'; A'; F'; P; e')$ *such that* $(\Sigma; A; F; P; e) \longmapsto_{2,top} (\Sigma'; A'; F'; P; e')$.

Proof: By induction on the structure of the typing judgment $\cdot; \mathcal{L}; \Gamma; \mathbf{P} \vdash_2 e : \tau$ with the Canonical Forms Lemma 4.5.6, the Σ to Γ Lemma 4.5.7, the Reverse Evaluation Context Lemma 4.5.8, the Empty Δ Lemma 4.5.9, the Γ to Σ Lemma 4.5.10, and the Well-formed Type Lemma 4.5.14. □

Theorem 4.5.16 (Progress) *If* $\vdash_2 (\Sigma; A; F; P; e)$ ok *then either* e *is a value, or there exists* $(\Sigma'; A'; F'; P; e')$ *such that* $(\Sigma; A; F; P; e) \longmapsto_{2,top} (\Sigma'; A'; F'; P; e')$.

Proof: Straightforward use of the Progress Lemma 4.5.15. □

Preservation. In this section, we present the lemmas used to prove Preservation Theorem 4.5.31.

Lemma 4.5.17 (Protection Domain Substitution) *If* $\Delta, \rho \vdash_2 p$ *and* $\Delta \vdash_2 p'$, *then* $\Delta \vdash_2 p[p'/\rho]$.

Proof: By induction on the structure of the typing rule $\Delta \vdash_2 p$. □

Lemma 4.5.18 (Lattice Protection Domain Substitution) *If* $\Pi \vdash_2 p \leq p'$ *and* $\Delta \vdash_2 p''$,
then $\Pi[p''/\rho]| - p[p''/\rho] \leq p'[p''/r]$

Proof: By induction on the structure of the typing rule $\Pi \vdash_2 p \leq p'$. □

Lemma 4.5.19 (Type Protection Domain Substitution) *If* $\Delta, \rho \vdash_2 \tau$ *and* $\Delta \vdash_2 p$, *then* $\Delta \vdash_2 \tau[p/\rho]$.

Proof: By induction on the structure of the typing rule $\Delta \vdash_2 \tau$. □

Lemma 4.5.20 (Value Protection Domain Substitution) *If* $\Delta, \rho; \Pi; \Gamma \vdash_2 v : \tau$ *and* $\Delta \vdash_2 p$,
then $\Delta; \Pi[p/\rho]; \Gamma[p/\rho] \vdash_2 v[p/\rho] : \tau[p/\rho]$.

Proof: By induction on the structure of typing rule $\Delta; \Pi; \Gamma \vdash_2 v : \tau$ with the Protection Domain Substitution Lemma 4.5.17, the Lattice Protection Domain Substitution Lemma 4.5.18, the Type Protection Domain Substitution Lemma 4.5.19, and the Expression Protection Domain Substitution Lemma 4.5.21. □

Lemma 4.5.21 (Expression Protection Domain Substitution)
If $\Delta, \rho; \Pi; \Gamma; \mathbf{P} \vdash_2 e : \tau$ *and* $\Delta \vdash_2 p$,
then $\Delta; \Pi[p/\rho]; \Gamma[p/\rho]; \mathbf{P} \vdash_2 e[p/\rho] : \tau[p/\rho]$.

Proof: By induction on the structure of typing rule $\Delta; \Pi; \Gamma; \mathbf{P} \vdash_2 e : \tau$ with the Protection Domain Substitution Lemma 4.5.17, the Lattice Protection Domain Substitution Lemma 4.5.18, the Type Protection Domain Substitution Lemma 4.5.19, and the Value Protection Domain Substitution Lemma 4.5.20. □

Lemma 4.5.22 (Value Substitution) *If* $\Delta; \Pi; \Gamma, x : \tau_2 \vdash v_1 : \tau_1$ *and* $\Delta; \Pi; \Gamma \vdash v_2 : \tau_2$,
then $\Delta; \Pi; \Gamma \vdash v_1[v_2/x] : \tau_1$.

Proof: By induction on the structure of typing rule $\Delta; \Pi; \Gamma \vdash v : \tau$ with the Expression Substitution Lemma 4.5.23. □

Lemma 4.5.23 (Expression Substitution) *If* $\Delta; \Pi; \Gamma, x : \tau_2; \mathbf{P} \vdash e : \tau_1$ *and* $\Delta; \Pi; \Gamma \vdash v : \tau_2$,
then $\Delta; \Pi; \Gamma; \mathbf{P} \vdash e[v/x] : \tau_1$.

Proof: By induction on the structure of typing rule $\Delta; \Pi; \Gamma; \mathbf{P} \vdash e : \tau$ with the Value Substitution Lemma 4.5.22. □

Lemma 4.5.24 (Aspect Store Weakening) *If* $\mathcal{L}; \Gamma \vdash A\ ok$,
then

- $\mathcal{L}; \Gamma, i : \mathsf{infile}_{\mathbf{P}} \vdash A\ ok$ *for some* $i, \mathbf{P}$
- $\mathcal{L}; \Gamma, o : \mathsf{outfile}_{\mathbf{P}} \vdash A\ ok$ *for some* $o, \mathbf{P}$

Proof: By induction on the structure of typing rule $\mathcal{L}; \Gamma \vdash Aok$. □

Lemma 4.5.25 (Lattice Stripping Expressions) *If* $\mathcal{L} \vdash_2 \mathbf{P}' \leq \mathbf{P}''$ *and* $\cdot; \mathcal{L}, \mathbf{P}' \leq \mathbf{P}''; \Gamma; \mathbf{P} \vdash_2 e : \tau$,
then $\cdot; \mathcal{L}, \mathbf{P}' \leq \mathbf{P}''; \Gamma; \mathbf{P} \vdash_2 e : \tau$.

Proof: By induction on the structure of the typing rule $\Delta; \Pi; \Gamma; \mathbf{P} \vdash_2 e : \tau$. □

Lemma 4.5.26 (β-redex Preservation Lemma) *If* $F \vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A\ ok$ *and* $\cdot; \mathcal{L}; \Gamma; \mathbf{P} \vdash e : \tau$ *and* $(\Sigma; A; F; \mathbf{P}; e) \longmapsto_{2,\beta} (\Sigma'; A'; F'; \mathbf{P}; e')$,
then

- $F' \vdash \Sigma' : \Gamma'$
- $\Gamma' \vdash A'ok$

- $\cdot; \mathcal{L}; \Gamma'; \mathbf{P} \vdash e' : \tau$.

Proof: By induction on the structure of the operational rules $(\Sigma; A; F; \mathbf{P}; e) \longmapsto_{2,\beta} (\Sigma'; A'; F'; \mathbf{P}; e')$ with the Σ to Γ Lemma 4.5.7, the Well-formed Type Lemma 4.5.14, and the Empty Δ Lemma 4.5.9.

- Case cev:o uses the Expression Protection Domain Substitution Lemma 4.5.21 and the the Expression Substitution Lemma 4.5.23.
- Cases c2eb:openr and c2eb:openw use the A Weakening Lemma 4.5.24.
- Case ceb:ifpdthen uses the Lattice Stripping Lemma 4.5.25.

□

Lemma 4.5.27 (Evaluation Context Preservation) *If* $\cdot; \mathcal{L}; \Gamma; \mathbf{P} \vdash E[e] : \tau$ *and* $\cdot; \mathcal{L}; \Gamma; \mathbf{P} \vdash e : \tau'$ *and* $\cdot; \mathcal{L}; \Gamma; \mathbf{P} \vdash e' : \tau'$,
then $cdot; \mathcal{L}; \Gamma; \mathbf{P} \vdash E[e'] : \tau$

Proof: By induction on the structure of evaluation contexts E. □

Lemma 4.5.28 (Projection Typing) *If* $\Delta; \Pi; \Gamma; \mathbf{P} \vdash e : \tau$,
then $\Delta; \Pi; \Gamma; \mathbf{P} \vdash |e|_i : \tau$ *for all* $i \in \{1, 2\}$

Proof: By induction on the structure of the typing rules $\Delta; \Pi; \Gamma; \mathbf{P} \vdash e : \tau$.
□

Lemma 4.5.29 (Preservation Lemma) *If* $F \vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A$ *ok and* $\cdot; \mathcal{L}; \Gamma; \mathbf{P} \vdash e : \tau$ *and* $(\Sigma; A; F; \mathbf{P}; e) \longmapsto_2 (\Sigma'; A'; F'; \mathbf{P}; e')$, *then*

- $F' \vdash \Sigma' : \Gamma'$
- $\Gamma' \vdash A' ok$
- $\cdot; \mathcal{L}; \Gamma'; \mathbf{P} \vdash e' : \tau$.

Proof: By induction on the structure of the operational rules $(\Sigma; A; F; \mathbf{P}; e) \longmapsto_2 (\Sigma'; A'; F'; \mathbf{P}; e')$.

- Case ce:beta uses the β-redex Preservation Lemma 4.5.26.
- Case ce:eval uses the Reverse Evaluation Context Lemma 4.5.8 and the Evaluation Context Preservation Lemma 4.5.27.
- Case c2e:highlow uses the Projection Typing Lemma 4.5.28.

□

Lemma 4.5.30 (Preservation Top Lemma) *If* $F \vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A$ *ok and* $\cdot; \mathcal{L}; \Gamma; \mathbf{P} \vdash e : \tau$ *and* $(\Sigma; A; F; \mathbf{P}; e) \longmapsto_{2,top} (\Sigma'; A'; F'; \mathbf{P}; e')$, *then*

- $F' \vdash \Sigma' : \Gamma'$
- $\Gamma' \vdash A' ok$
- $\cdot; \mathcal{L}; \Gamma'; \mathbf{P} \vdash e' : \tau$.

Proof: By induction on the structure of the operational rule $(\Sigma; A; F; \mathbf{P}; e) \longmapsto_{2,top} (\Sigma'; A'; F'; \mathbf{P}; e')$.

- Case cevt:ce uses the Preservation Lemma 4.5.29.

□

Theorem 4.5.31 (Preservation) *If* $\vdash_2 (\Sigma; A; F; \mathbf{P}; e)$ ok *and* $(\Sigma; A; F; \mathbf{P}; e) \longmapsto_{2,top} (\Sigma'; A'; F'; \mathbf{P}; e')$ *then* $\vdash_2 (\Sigma'; A'; F'; \mathbf{P}; e')$ ok.

Proof: Straightforward use of the Top Preservation Lemma 4.5.30. □

Equivalent Execution.

Lemma 4.5.32 (Equivalent Execution in $\mathbb{F}_{HRM2}$)
If $\mathbf{HIGH} \in H$ *and* $F \vdash_2 \cdot : \Gamma$ *and* $\cdot; \mathcal{L}; \Gamma; \mathbf{HIGH} \vdash_2 e :$ bool *and*
$(\cdot; \cdot; F; \mathbf{HIGH}; e) \longmapsto^*_{2,top} (\Sigma, A, F'; \mathbf{HIGH}; v)$ *then* $|v|_1 = |v|_2$.

Proof: By the Preservation Theorem 4.5.31, $F' \vdash_2 \Sigma : \Gamma'$ and $\cdot; \mathcal{L}; \Gamma' \vdash_2 v :$ bool.
By the Canonical Forms Lemma 4.5.6, v is either **true** or **false**.
$|\mathbf{true}|_1 = |\mathbf{true}|_2$ and $|\mathbf{false}|_1 = |\mathbf{false}|_2$. □

Putting it all together: Noninterference.

Theorem 4.5.33 (Noninterference) *If* $F \vdash_2 \cdot : \Gamma$ *and* $\mathbf{HIGH} \in H$ *and* $\mathbf{LOW} \in L$ *and* $\mathcal{L} \vdash \mathbf{LOW} \leq \mathbf{HIGH}$ *and* e *is a core language expression where* $\cdot; \mathcal{L}; \Gamma, x : 1; \mathbf{HIGH} \vdash e :$ bool *and* $\cdot; \mathcal{L}; \Gamma; \mathbf{LOW} \vdash e' : 1$ *and*
$(\cdot; \cdot; F; \mathbf{HIGH}; e[\mathbf{LOW}\texttt{<}e'\texttt{>}/x]) \longmapsto^*_{top} (\Sigma_1; A_1; F_1; \mathbf{HIGH}; v_1)$
and
$(\cdot; \cdot; F; \mathbf{HIGH}; e[\mathbf{LOW}\texttt{<}()\texttt{>}/x]) \longmapsto^*_{top} (\Sigma_2; A_2; F_2; \mathbf{HIGH}; v_2)$
then $v_1 = v_2$.

Proof: Construct the $\mathbb{F}_{HRM2}$ expression $\mathbf{LOW}\texttt{<}e' \texttt{|} ()\texttt{>}; e$, such that
$|\mathbf{LOW}\texttt{<}e' \texttt{|} ()\texttt{>}; e|_1 = \mathbf{LOW}\texttt{<}e'\texttt{>}; e$ and $|\mathbf{LOW}\texttt{<}e' \texttt{|} ()\texttt{>}; e|_2 = \mathbf{LOW}\texttt{<}()\texttt{>}; e$.

Therefore, $|(\cdot;\cdot;F;\mathbf{HIGH};\mathbf{LOW}\texttt{<}e'\,|\,()\texttt{>};e)|_1 \longmapsto^*_{2,top} (\Sigma_1;A_1;F_1;\mathbf{HIGH},v_1)$ and $|(\cdot;\cdot;F;\mathbf{HIGH};\mathbf{LOW}\texttt{<}e'\,|\,()\texttt{>};e)|_2 \longmapsto^*_{2,top} (\Sigma_2;A_2;F_2;\mathbf{HIGH};v_2)$.

By the Completeness Theorem 4.5.5,

$(\cdot;\cdot;F;\mathbf{HIGH};\mathbf{LOW}\texttt{<}e'\,|\,()\texttt{>};e) \longmapsto^*_{2,top} (\Sigma;A;F';\mathbf{HIGH};v)$ for some Σ, A, F', and v.

By the Soundness Theorem 4.5.3, for $i \in \{1,2\}$, $|(\cdot;\cdot;F;\mathbf{HIGH};\mathbf{LOW}\texttt{<}e'\,|\,()\texttt{>};e)|_i \longmapsto^*_{2,top} |(\Sigma;A;F';\mathbf{HIGH};v)|_i$.

Therefore, $|v|_1 = v_1$ and $|v|_2 = v_2$.

By the Equivalent Execution Lemma 3.2.30, $|v|_1 = |v|_2$.

Therefore, $v_1 = v_2$. □

Our noninterference theorem proves that low-protection core calculus expression that uses the file I/O operations cannot change the behavior of the high-protection code.

4.5.5 File I/O Translation Generated by Interference Policy

The file I/O interference policy defined in Figure 4.16 describes a translation from source to core for file I/O operations. The general algorithm of the translation was described in Section 4.4 – using that algorithm, the translation constructed for our file I/O library is shown in Figure 4.26.

First, the static checks required by the interference policy on `read` and `write` operations are enforced in Rules set:readfile and set:writefile by checking the current protection domain and the protection domain of the file descriptor for their proper order in the protection domain lattice $\mathcal{L}$. Recall that the `read` operation requires a read ($\mathcal{L} \vdash \mathbf{P} \leq \mathbf{P}'$) static check, while the `write` operation requires both read ($\mathcal{L} \vdash \mathbf{P} \leq \mathbf{P}'$) and write ($\mathcal{L} \vdash \mathbf{P}' \leq \mathbf{P}$) static checks.

Type Translation $\vdash t \overset{\mathbf{typ}}{\Longrightarrow} t$

$$\frac{}{\vdash \texttt{fname} \overset{\mathbf{typ}}{\Longrightarrow} \exists\rho.(\mathsf{S}_\rho \times \mathsf{fname}_\rho)}\ \text{stt:fname}$$

Value Translation $\Phi;\Gamma \vdash v : t \overset{\mathbf{val}}{\Longrightarrow} v'$

$$\frac{F(f) = (s, \mathbf{P})}{\Phi;\Gamma \vdash f : \texttt{fname} \overset{\mathbf{val}}{\Longrightarrow} \mathbf{pack}(\mathbf{P}, (\mathbb{P}, f))\ \mathbf{as}\ \exists\rho.(\mathsf{S}_\rho \times \mathsf{fname}_\rho)}\ \text{svt:f}$$

Expression Translation $\Phi;\Gamma;P \vdash e : t \overset{\mathbf{exp}}{\Longrightarrow} e'$

$$\frac{\Phi;\Gamma;\mathbf{P} \vdash e : \texttt{infile}\ \mathbf{P}' \overset{\mathbf{exp}}{\Longrightarrow} e' \qquad \mathcal{L} \vdash \mathbf{P} \leq \mathbf{P}'}{\Phi;\Gamma;\mathbf{P} \vdash \texttt{read}\, e : \texttt{String} \overset{\mathbf{exp}}{\Longrightarrow} \mathbf{read}\, e'}\ \text{set:readfile}$$

$$\frac{\begin{array}{c}\Phi;\Gamma;\mathbf{P} \vdash e : \texttt{outfile}\ \mathbf{P}' \times \texttt{String} \overset{\mathbf{exp}}{\Longrightarrow} e' \\ \mathcal{L} \vdash \mathbf{P}' \leq \mathbf{P} \qquad \mathcal{L} \vdash \mathbf{P} \leq \mathbf{P}'\end{array}}{\Phi;\Gamma;\mathbf{P} \vdash \texttt{write}\, e : \texttt{String} \overset{\mathbf{exp}}{\Longrightarrow} \mathbf{write}\, e'}\ \text{set:writefile}$$

$$\frac{\Phi;\Gamma;\mathbf{P} \vdash e : \texttt{fname} \overset{\mathbf{exp}}{\Longrightarrow} e' \qquad \mathcal{L} \vdash \mathbf{P}' \leq \mathbf{P}}{\begin{array}{l}\Phi;\Gamma;\mathbf{P} \vdash \texttt{openr}\ \mathbf{P}'\, e : t \overset{\mathbf{exp}}{\Longrightarrow} \\ \quad \mathbf{open}(r, x) = e'\ \mathbf{in}\ \mathbf{split}\ (y_{pd}, y_{fn}) = x\ \mathbf{in} \\ \quad \mathbf{ifpd}\ (\mathbb{P}' \leq y_{pd})\ \mathbf{then}\ \mathbf{openr}_{\mathbf{P}'}\, y_{fn}\ \mathbf{else}\ \mathbf{abort}_{\mathsf{infile}_{\mathbf{P}'}}\end{array}}\ \text{set:openr}$$

$$\frac{\Phi;\Gamma;\mathbf{P} \vdash e : \texttt{fname} \overset{\mathbf{exp}}{\Longrightarrow} e' \qquad \mathcal{L} \vdash \mathbf{P}' \leq \mathbf{P}}{\begin{array}{l}\Phi;\Gamma;\mathbf{P} \vdash \texttt{openr}\ \mathbf{P}'\, e : t \overset{\mathbf{exp}}{\Longrightarrow} \\ \quad \mathbf{open}(r, x) = e'\ \mathbf{in}\ \mathbf{split}\ (y_{pd}, y_{fn}) = x\ \mathbf{in} \\ \quad \mathbf{ifpd}\ (y_{pd} \leq \mathbb{P}')\ \mathbf{then}\ \mathbf{openw}_{\mathbf{P}'}\, y_{fn}\ \mathbf{else}\ \mathbf{abort}_{\mathsf{outfile}_{\mathbf{P}'}}\end{array}}\ \text{set:openw}$$

Figure 4.26: Translation based on File I/O Interference Policy

Next, filenames, with their `fname` resource type, are translated in Rule svt:f to a packed tuple with a run-time protection domain representing the result of a resource policy query and the actual core calculus filename. The existentially-wrapped value will be unwrapped at run time in Rules set:openr and set:openw when the filename is to be used by an *openr* or *openw* operation. The run-time protection domains will be used in any dynamic checks required by the interference policy. Recall that the *openr* operation required a read (**ifpd** ($\mathbb{P} \leq y_{pd}$)) dynamic check, while the *openw* operation required a write (**ifpd** ($y_{pd} \leq \mathbb{P}$)) dynamic check.

As an aside, for the translation type-safety proof, we will extend the pointcut environment translation function $\mathcal{T}$ to map a file store F into a variable environment Γ. The created environment will map filenames to filename resource types.

$$\mathcal{T}(F, f \rightarrow (\_, \mathbf{P})) = \mathcal{T}, f : \mathsf{fname}_{\mathbf{P}} \qquad \mathcal{T}(\Gamma, x : t) = \mathcal{T}(\Gamma), x : \tau \ \textit{where} \ \vdash_2 t \Rightarrow \tau$$

$$\mathcal{T}(\Phi) = \ \textit{as defined previously}$$

Finally, we show that translation generated from the file I/O interference policy is type-safe.

Lemma 4.5.34 (Type Translation) *If* $\vdash_2 t \Rightarrow \tau$, *then* $\cdot \vdash_2 \tau$.

Proof: By induction on the structure of the translation rule $\vdash_2 t \Rightarrow \tau$. □

Lemma 4.5.35 (Environment Translation) *If* $\mathcal{T}(F) = \Gamma$, *then* $F \vdash_2 \cdot : \Gamma$.

Proof: Straightforward. □

Lemma 4.5.36 (Translation Lemma) *If* $\vdash_2 t \Rightarrow \tau$

- *and* $\Phi; \Gamma \vdash v : t \stackrel{\mathbf{val}}{\Longrightarrow} v'$, *then* $\cdot; \mathcal{L}; \mathcal{T}(F), \mathcal{T}(\Phi), \mathcal{T}(\Gamma) \vdash_2 v : \tau$.
- *and* $\Phi; \Gamma; p \vdash e : t \stackrel{\mathbf{exp}}{\Longrightarrow} e'$, *then* $\cdot; \mathcal{L}; \mathcal{T}(F), \mathcal{T}(\Phi), \mathcal{T}(\Gamma); \mathbf{P} \vdash_2 e : \tau$.
- *and* $\Phi; \Gamma; p \vdash as; asps; e : \tau \stackrel{\mathbf{dec}}{\Longrightarrow} e'$, *then* $\cdot; \mathcal{L}; \mathcal{T}(F), \mathcal{T}(\Phi), \mathcal{T}(\Gamma); \mathbf{P} \vdash_2 e : \tau$.

Proof: By mutual induction on the translation judgments with the Type Translation Lemma 4.5.34 and the Environment Translation Lemma 4.5.35. □

Theorem 4.5.37 (Translation Type Safety)
If $\vdash$ *prog* : `Bool` $\stackrel{\mathbf{prog}}{\Longrightarrow} e$ *then* $\cdot; \mathcal{L}; \Gamma; \mathbf{MAIN} \vdash_2 e$: `bool`.

Proof: Straightforward use of the Translation Lemma 4.5.36. □

4.5.6 Harmlessness of File I/O using Interference Policy

We now want to use our core calculus noninterference theorem (Theorem 4.5.33) to verify the harmlessness of the source language. Recall that "harmlessness" means that code defined in the aspect declarations, $P : \{as\}$, does not affect the behavior of the mainline program, $ds + e$. Again, when the source language is translated into the core calculus, code defined by the mainline program is placed in the **MAIN** protection domain. Code defined in an aspect $ASPECT$ is placed in an **ASPECT** protection domain that is lower than the **MAIN** protection domain.

Source language file system I/O is translated into the core calculus with the static and dynamic checks required by the interference policy. An aspect declaration, $P' : \{as\}$, is translated into the $\mathbb{F}_{HRM2}$ expression, $P'\texttt{<}()\texttt{|}e'\texttt{>}$, where e' is the $\mathbb{F}_{HRM}$ expression generating by translating as. In other words, aspect declarations

are translated into the simultaneous execution of the main program without the aspect and with the aspect. Because this resulting translation is type-safe by Theorem 4.5.37, the noninterference theorem of the core calculus is combined with this translation of aspect declarations to prove that the low-protection code, the aspect declaration, will not interfere with the high-protection code, the mainline program. Therefore, aspects in the source language that use file I/O are harmless.

Theorem 4.5.38 (Source Language Aspects are Harmless)
If $\vdash prog : \mathsf{bool} \stackrel{\mathbf{prog}}{\Longrightarrow} e$
and $(\cdot;\cdot;F;\mathbf{MAIN};|e|_1) \longmapsto^*_{top} (\Sigma_1;A_1;F_1;\mathbf{MAIN};v_1)$
and $(\cdot;\cdot;F;\mathbf{MAIN};|e|_2) \longmapsto^*_{top} (\Sigma_2;A_2;F_2;\mathbf{MAIN};v_2)$
then $v_1 = v_2$.

Proof: Straightforward use of the Translation Type Safety Theorem 4.5.37 and the proof technique and lemmas of the Noninterference Theorem 4.5.33. □

We have now shown that our interference policy mechanism can be used to enforce our previous definition of harmlessness on a file I/O library.

Chapter 5

Related Work and Conclusions

5.1 Related Work

In this chapter, we describe related work to the thesis. In Section 5.1.1, we describe several functional aspect-oriented programming language designs. In Section 5.1.2, we describe research into polymorphic aspect-oriented programming languages. In Section 5.1.3, we describe previous attempts to enforce security policies using aspect-like execution monitoring systems. In Section 5.1.4, we describe attempts by researchers to classify advice into "harmless" and "harmful" categories. In Section 5.1.5, we describe information flow systems related to our protection domain mechanism. Finally, in Section 5.1.6, we examine projects that combine aspect-oriented programming and module systems.

5.1.1 Aspect-oriented programming languages

In this section, we describe related research into functional aspect-oriented programming languages.

MinAML. As mentioned earlier, our work builds upon the framework proposed by Walker, Zdancewic, and Ligatti [37]. That work distilled aspect-oriented programming into its fundamental components, clearly separating the semantics of aspects (join points and advice) from that of the underlying programming language. They used the simply-typed λ-calculus as the basic model of computation. The reason that they could develop such a minimal model is that their calculus reduced the aspect language to the most critical components, relying on translation from their MinAML source language to generate the full power of modern AOPLs.

There are four fundamental differences between MinAML and AspectML.

- Most fundamentally, MinAML was a monomorphically-typed aspect-oriented programming language. As such, functions with diverse argument and result types could not be advised by a single piece of advice. AspectML allows us to write polymorphic advice that can advise a diverse collection of functions.

- The semantics for "around" advice, specifically the order in which nested around advice was triggered, in MinAML was unconventional compared to around advice in the literature. Around advice in AspectML behaves in the conventional fashion.

- AspectML utilizes a novel type inference algorithm that is conservative over Hindley-Milner inference. This algorithm was not described in this thesis. MinAML lacked a type inference algorithm.

- Unlike in MinAML, where labels were monomorphic, polymorphism allows us to structure the labels in a tree-shaped hierarchy in AspectML. Labels lower in the hierarchy will trigger pointcuts that contain labels above them in the hierarchy. This allows us to create `any` pointcuts in AspectML.

There are two fundamental differences between MinAML and HarmlessAML.

- Advice in MinAML could be triggered before, after, or in place of (around) a function. Harmless advice in HarmlessAML can only be triggered before or after a function – using advice in place of a function would not be harmless.
- Advice in MinAML could modify the behavior of the main program– there is no mechanism like that in HarmlessAML to enforce the harmlessness of advice. Advice in MinAML could not be added to a program without fear that important program invariants would be disrupted. Programmers who develop, debug, or enhance mainline code must examine all present MinAML advice to determine the true behavior of the program.

Untyped and Typed Aspect-oriented Calculi. Jagadeesan, Jeffrey, and Riely [27] developed a class-based aspect-oriented programming language. The language was untyped – they provided a dynamic semantics but not a static semantics for their language. Unlike in AspectML or HarmlessAML, the aspect-oriented features were not orthogonal to other language features. Advice was triggered implicitly during function application.

The main contribution was an analysis of "weaving." To contrast different methods of implementing aspect-oriented programming language, they provided a dynamic semantics directly for the aspect-oriented language, and then provided a separate "weaving" algorithm that translated the aspect-oriented language to the underlying class-based, non-aspect-oriented language. This is different from AspectML and HarmlessAML, which translate the source language to a simpler core language, not to a non-aspect-oriented version of the source language. AspectJ is also implemented with a "weaving" algorithm – AspectJ advice is translated into

ordinary Java bytecode. They then showed that such weaving is correct with regard to the direct aspect-oriented dynamic semantics, with the exception of dynamic loading of advice that affects existing classes.

In a separate paper [26], the authors expanded their work with a static semantics for their class-based, aspect-oriented language. Using the static and dynamic semantics, they proved a traditional type safety theorem for their aspect-oriented language. They also reexamined weaving, proving that well-typed aspect-oriented programs weave to well-typed non-aspect-oriented programs. Finally, they pointed out that polymorphic advice, not examined in their paper, would be required for many practical uses of advice.

AJD. Masuhara, Kiczales, and Dutchyn developed AJD [38], an aspect-oriented Scheme-based programming language. Partial evaluation was used to optimize away as much as aspect-oriented features as possible by weaving applicable static advice, advice that will always be triggered, into the main function code. Dynamic advice, advice that is triggered depending on the runtime state, was woven in with a guard that checked at runtime whether the required condition is met. Like Scheme and unlike AspectML and HarmlessAML, their language was dynamically typed.

Advice and Dynamic Join Points. Wand, Kiczales, and Dutchyn [52] provide a denotational, monadic semantics for an aspect-oriented language. As in other calculi, they provide a weaving algorithm that wraps function with the advice that they trigger. Their main results were their handling of dynamic join points, which refer to execution events, and advice written over recursive functions. Their language, unlike AspectML and HarmlessAML, is untyped.

μABC. Bruns, Jagadeesan, Jeffrey, and Riely developed the μABC aspect-oriented calculus [6]. Like our core calculus, they added advice and joinpoints as orthogonal constructs to function abstraction and application in their calculus. Their concept of a hierarchical set of "names," taken from concurrency theory, was similar to our core calculus labels. To show that their calculus was as powerful as the core calculus of MinAML, they provided a translation from MinAML to their core calculus. Also, to show what features are required to implement the orthogonal aspect-oriented features in their calculus, they provided a translation from μABC to the polyadic π-calculus. Unlike $\mathbb{F}_A$ and $\mathbb{F}_{HRM}$, μABC was untyped.

Aspect-oriented Scheme. Tucker and Krishnamurthi [51] developed a variant of Scheme with aspect-oriented features. As in $\mathbb{F}_A$ and $\mathbb{F}_{HRM}$, pointcuts and advice were first-class objects in their language. However, unlike $\mathbb{F}_A$ and $\mathbb{F}_{HRM}$, Scheme is dynamically typed.

5.1.2 Polymorphic Aspect-oriented Programming Languages

In this section, we describe related research into aspect-oriented programming languages with polymorphic types.

Aspectual Caml. Masuhara, Tatsuzawa, and Yonezawa implemented an aspect-oriented version of core O'Caml they call Aspectual Caml [39]. Their implementation effort was impressive and dealt with several features we have not considered here including curried functions and datatypes. Although there are similarities between AspectML and Aspectual Caml, there are also many differences:

- Pointcut designators in AspectML can only reference names that are in scope. AspectML names are indivisible and α-vary as usual. In Aspectual Caml, programmers used regular expressions to refer to all names that matched the regular expression in any scope. For instance, `get*` referenced all objects with a name beginning with `get` in all scopes.

- Aspectual Caml did not check pointcut designators for well-formedness. When a programmer wrote the pointcut designator `call f (x:int)`, the variable `f` was assumed to be a function and the argument `x` was assumed to have type `int`. There was some run-time checking to ensure safety, but it was not clear what happens in the presence of polymorphism or type definitions. Aspectual Caml does not appear to have run-time type analysis.

- Aspectual Caml pointcuts were second-class citizens. It was not possible to write down the type of a pointcut in Aspectual Caml, or pass a pointcut to a function, store it in a tuple, etc.

- The previous two limitations made it possible to develop a two-phase type inference algorithm for Aspectual Caml (ordinary O'Caml type inference occurred first and inference for pointcuts and advice occurred second), which bears little resemblance to the type inference algorithm for AspectML.

- There was no formal description of the Aspectual Caml type system, type inference algorithm, or operational semantics. We have a formal description of both the static semantics and the dynamic semantics of AspectML. AspectML's type system has been proven sound with respect to its operational semantics.

Staticness and Coherence. Meng Wang, Kung Chen, and Siau-Cheng Khoo [53] examined language design problems in combining aspects with a polymorphic functional language. Their design made fundamentally different assumptions about aspects that led to significant differences in expressiveness:

- Their advice was scoped such that it was not possible to install advice that would affect functions that had already been defined. We feel that this had both positive and negative consequences for the language. It had positive, because they used a type-directed weaving algorithm (not unlike the way type classes are compiled to dictionary passing in Haskell) to completely eliminate the need to dynamically calculate advice composition, as our operational semantics does. However, we feel that this design decision did not adequately take into account the needs of separate compilation: A programmer could not compile a program separately from its advice. Furthermore, some of our most interesting uses of advice so far have involved advising an already defined function.

- Their advice was named. Not only was this useful as mnemonic for the programmer, but it allowed them to advise advice. We do not presently name our advice, but there is no fundamental reason that we cannot, and likewise support advice advisement.

- Like Aspectual Caml, their pointcuts were second-class. We believe that first-class pointcuts are an important step toward allowing programmers to develop reusable libraries of advice.

- Their design did not provide a mechanism for examining the call-stack or obtaining information about the specific function being advised. But we do

not see any technical challenges that would prevent them from adding such features.

Aspectual Collaborations. Another study of the interaction between polymorphism and aspect-oriented programming features occurred in the context of Lieberherr, Lorenz and Ovlinger's Aspectual Collaborations [36]. They extended a variant of AspectJ with a form of module that allowed programmers to choose the join points (i.e., control-flow points) that were exposed to external aspects. Aspectual Collaborations had parameterized aspects that resembled the parameterized classes of Generic Java. When a parameterized aspect was linked into a module, concrete class names replaced the parameters. Since types were merely names, the sort of polymorphism necessary was much simpler (at least in certain ways) than required by a functional programming language. For instance, there was no need to develop a generalization relation and type analysis could be replaced by conventional object-oriented down-casts. Overall, the differences between functional and object-oriented language structure have caused our two groups to find quite different solutions to the problem of constructing generic advice.

Aspect Featherweight Generic Java. Jagadeesan, Jeffrey, and Riely [28] developed an aspect-oriented version of Featherweight Generic Java, that allowed polymorphic aspects to be written on generic methods. They provided two static semantics, the first where type information was carried at runtime and could be used during advice lookup. By placing restrictions on pointcut typing, they provided a second static semantics where types were not needed for advice lookup and could be erased. Well-typed program in the second static semantics were proven to be well-typed in the first static semantics. Their pointcut logic was relatively simple,

disallowing context-sensitive advice. Unlike AspectML and HarmlessAML, they did not translate their language to a simpler core calculus, and the dynamic semantics of their language were quite complex as a result.

5.1.3 Advice for Security

In this section, we describe related research into aspect-oriented implementations of security features.

Naccio. Evans and Twyman developed Naccio, a platform-independent execution monitoring system [16, 17]. Naccio enforced safety policies that restrict how programs can use abstract system resources, such as file and network I/O. The Naccio system was given a policy to be enforced, a list of the abstract system resources that the policy references, the code of the system library that manipulates system resources, and a mapping of system library operations to the abstract resources each manipulates. Naccio then returned a new version of the library where the library operations are protected by the resource checks required by the policy. Finally, the main program was modified to call the new resource library rather than the old one.

At the time, Evans and Twyman thought of Naccio as a domain specific language for implementing security policies, and they argued effectively that their language, which completely separated security code from mainline program, was an effective means of developing reliable security policies. The new, policy-enforcing version of the library could also be visualized as an aspect-oriented program where before advice that enforced the safety policy was triggered by each of the library operations. However, in such an aspect-oriented version of Naccio policy enforcement, the last step, rewriting the main program to point to the new library, would not be necessary.

As described in Section 3.3.3, we implemented the suite of example Naccio security policies as harmless advice to study the usefulness of harmless advice somewhat more broadly in the security domain. We found that many access control tasks could be performed by harmless advice.

Security Automata SFI Implementation. Erlingsson and Schneider created SASI [14], a reference monitor that enforced security policies by modifying object code. They described a security policy in terms of a security automaton whose transition relations were operations on system resources. During execution, if there was no transition relation out of the current state marked with the resource operation to be performed, then the operation was blocked.

SASI was implemented for x86 assembly and Java Virtual Machine Language code by modifying the object code of the program to include the checks required by the security automaton. SASI could also be visualized as an aspect that simulates a security automation by updating and checking the automaton state with before advice on resource operations.

Monitoring and Checking. Lee, Kannan, Kim, Sokolsky, and Viswanathan developed the MaC framework [32, 35] for execution monitoring. Like Naccio, security policies were specified in terms of high-level, abstract system resources. The MaC framework consisted of a filter, event recognizer, and run-time checker. The filter monitored program execution for low-level resource operations, the event recognizer mapped those low-level operations to the high-level, abstract resource that they manipulated, and the run-time checker verified that the resulting manipulation of the abstract resources was allowed by the security policy. As in the previous security

mechanisms, the MaC framework could also be thought of as a security aspect that enforces policies on the low-level resource operations.

PoET/PSLang. Erlingsson and Schneider described PoET/PSLang [15], the successor to their SASI system. Like our aspect-oriented Java security case study in Section 2.3, they examined an alternate implementation of the Java security mechanism using in-lined reference monitors. They discovered that this aspect-like reference monitor implementation allowed a flexibility in enforcing security policies that was lacking in the existing Java virtual machine implementation.

Polymer. Bauer, Ligatti and Walker [4] introduced a calculus that included several different kinds of aspect combinators (parallel conjunction and disjunction; sequenced conjunction and disjunction) and used a type and effect system to prevent interference between them. The technical machinery they used was extremely complicated and quite different from the current work. In contrast to harmless advice, they did not concern themselves with the effects these aspects would have on the mainline computation.

5.1.4 Classifying Advice

In this section, we describe attempts to classify aspect-oriented programs into "harmless" and "harmful" advice categories.

Observers and Assistants. Clifton and Leavens [7] proposed techniques for Hoare-style reasoning using JML specifications on AspectJ programs. They classified advice into *assistants* and *observers*. Our notion of harmless advice is similar to their notion of observers — an observer does not change the pre- and post- conditions

of the specification (ie. the behavior) of the mainline code. They postulated a static analysis to classify observer advice, but did not formalize it due to difficulty in dealing with aliasing in AspectJ.

The details of the type and effect system of $\mathbb{F}_{HRM}$ are entirely different from their Hoare logic. One point of interest is that Clifton and Leavens mentioned that it is not clear whether their model could "accommodate dynamic context join points like CFlow." Our analysis of our stack operations, which are sufficient for coding up CFlow-like primitives, indicates that harmless advice can indeed safely use these primitives and avoid interfering with the mainline computation or each other.

Orthogonal, Independent, and Observation Aspects. Rinard, Salcianu, and Bugrara [46] developed a system that classified the interaction between aspects and main program code into five categories: orthogonal (aspects and main program code have no fields in common), independent (aspects and main program code cannot write to fields that the other can access), observation (aspects can read fields that the main program code writes), actuation (aspects can write to fields that the main program code reads), and interference (aspects and main program code can write to the same fields) interactions. Our definition of harmlessness would include Rinard, et al.'s orthogonal, independent, and observation interactions. Their tool used a context-sensitive, flow-sensitive pointer, escape, and effect analysis and operated within Java's nominal type structure, so the details of their system are quite different from our own. In addition, their system was described informally in English; they have not proven any properties of their analyses.

Almost Spectative Advice. Katz [29] also described several categories of advice, including "almost spectative" advice. "Almost spectative" advice was similar

to our notion of harmless advice in that it could halt execution but did not otherwise influence the result of the program. He then speculated on several potential methods to categorize advice, including regression testing, proofs by induction, and dataflow analysis.

Strongly Independent Advice. Douence, Fradet and Südholt [13] analyzed aspects defined by recursion in an untyped calculus together with parallel and sequencing combinators. They developed a number of formal laws for reasoning about their combinators and an algorithm that was able to detect *strong independence*. Two pieces of advice were strongly independent when they did not interfere with each other regardless of the contents of the advice bodies or the contents of the programs they are applied to. In other words, strong independence was determined exclusively by analysis of the pointcut designators of the two pieces of advice and consequently it is orthogonal to our analysis which (mostly) ignores the pointcuts and examines the advice bodies instead.

Model Checking of Advice. Krishnamurthi, Fisler, and Greenberg [33] tackled the more general problem of verifying aspect-oriented programs. Given a set of properties a program must satisfy, specified in a temporal logic, and a set of pointcut designators, they verified programs using model checking. Their approach to verification was partly modular since as long as the set of pointcuts did not change and the underlying mainline code remained fixed, it was not necessary to reanalyze the mainline code as advice definitions were edited. However, if the pointcuts or mainline code did change, the whole program must be rechecked. Aside from the fact that we are both interested in modular checking of aspect-oriented programs, there is not too much similarity between the techniques. In terms of trade-offs, our approach

is lighter weight (temporal specifications of properties are unnecessary) and more modular (changing pointcut designators used by advice does not necessitate re-type checking the mainline program), but checks a much coarser-grained property (we guarantee that *all* functional properties of advice are preserved).

5.1.5 Protection Domains and Information Flow

In this section, we describe similar research to our protection domain mechanism and information flow analysis.

Region-based Memory Management. Region-based memory management [23] was a technique to ensure that, when a program was evaluated, no accesses to unallocated or freed memory could occur at runtime. This was enabled by allocating each piece of data on the program heap into a particular *region*, or area of memory. There were typically three primitives that operated on regions: region creation, allocation or "tagging" of a value into a region, and deallocation or "freeing" of a region. The required property that no program could access unused or freed memory than became that a value tagged with a region that had been deallocated could not be used. This property was enforced by an effect type system that not only tracked the region of a value but also tracked the *effect* of an expression. The effect of an expression was a list of regions the expression would need to access during evaluation. A well-typed expression could not access a deallocated region.

There are several similarities and differences between regions and protection domains.

- Both systems use an effect type system. The type system of region-based memory management tracked what regions are accessed when evaluating an

expression. The type system of $\mathbb{F}_{HRM}$ places an expression into a protection domain and ensures that the expression cannot use values from a higher protection domain. The type system tracks information flow to preserve an noninterference property between protection domains.

- The set of regions grew and shrunk during evaluation, as there were explicit primitives for region creation and deallocation. In that way, an evaluating expression could not use a value from a deallocated region. In HarmlessAML, the set of protection domains is static–the main program and each aspect is allocated a protection domain.

- In region-based memory-management, type systems have been developed that allow effect polymorphism. This enables the effect of a function (the list of regions accessed by the function) to be polymorphic. We have not developed a similar system for protection domain polymorphism; therefore, a function can only be applied in a particular protection domain.

- Neither system require the programmer to explicitly manage regions or protection domains. Region inference mechanisms have be created that insert region annotations into programs. In harmless advice, when HarmlessAML programs are translated into $\mathbb{F}_{HRM2}$ expressions, main program code is implicitly placed in the **MAIN** protection domain and code in aspect ASP is implicitly placed in the **ASP** protection domain.

- The goals of the type systems are different. The goal of region-based memory management is to prevent memory accesses to unallocated or freed memory. The goal of HarmlessAML is to prevent aspects from altering the behavior of mainline program code.

JFlow and Jif. Myers developed JFlow [41], an extension of Java with information flow annotations to protect the privacy and integrity of data. The JFlow compiler statically checked the information flow annotations in a manner similar to type checking. The compiler then transformed the verified JFlow program into a standard Java program.

Like HarmlessAML, JFlow ignored some kinds of interference such as changes to the timing and termination behavior of programs, arguing that these kinds of interference would have a minimal impact on security. However, their information flow annotation system was complex, requiring annotations on many of the types in the language. By contrast, due to the syntactic separation between aspects and mainline code in HarmlessAML, the protection domain annotations in $\mathbb{F}_{HRM}$ are simpler and required on less types than general information flow annotations. The resulting $\mathbb{F}_{HRM}$ calculus is simpler to understand and still enforces the required properties

Information Flow for CoreML. Pottier and Simonet developed an information flow analysis [45] for a simplified ML-like calculus. Our noninterference proof in Section 3.2.5 that creates a new core calculus $\mathbb{F}_{HRM2}$ to simulate the simultaneous execution of two $\mathbb{F}_{HRM}$ expressions is directly based on their work. However, as in the JFlow system, the syntactic separation between aspects and mainline code in HarmlessAML allows us to utilize a simpler type system then their work. Their calculus included exceptions and let-polymorphism (with an included type inference system), which $\mathbb{F}_{HRM}$ does not contain. However, their calculus did not examine any aspect-oriented features, while our $\mathbb{F}_{HRM}$ examines how primitive advice, pointcuts, and joinpoints interact with our protection domain system.

5.1.6 Aspects and Module Systems

Another interesting line of current research involves finding ways to add aspect-oriented programming features to languages with module systems, or vice-versa.

Aspectual Collaborations. One of the first systems to combine aspects and modules effectively was Lieberherr, Lorenz and Ovlinger's *Aspectual Collaborations* [36, 42]. Their proposal allowed module programmers to choose the join points (i.e., control-flow points) that they will expose to external advice. External advice could not intercept control-flow points that had not been exposed. In this sense, Aspectual Collaborations were not completely oblivious – programmers must make choices about which control-flow points to expose up-front, during program design. Aspectual Collaborations did enjoy a number of important properties including strong encapsulation, type safety and the possibility of separately compiling and checking module definitions.

Open Modules. Aldrich proposed another model for combining aspects and modules called *Open Modules* [2]. Open Modules have a special module sealing operator that hides internal control-flow points from external advice. Aldrich used logical relations to show that sealed modules had a powerful implementation-independence property. In essence, the behavior of advice did not depend on the behavior of the mainline program. HarmlessAML differs from this previous research as we do not suggest that visibility of the interception points be limited; instead, we suggest limiting the capabilities of advice.

Information Hiding. Sullivan, et al. [49] presented a case study transforming crosscutting concerns in an overlay network application into aspects. Based on

their experience, they proposed a software engineering technique where the mainline code programmer is aware of locations in their code that might be requested to trigger future common crosscutting concerns and takes care to allow access for aspect programmers to the corresponding pointcuts. As in Aldrich's work, this allowed the mainline module code to change without disrupting the aspects.

Aspect-aware Modules. Kiczales and Mezini [31], rather than restricting aspects' influence on a module, proposed a new method of modular reasoning in the presence of aspects. To determine the interface to a module in their system, the entire program must be examined—an aspect-aware module interface also lists all aspects that are triggered by the module.

5.2 Concluding Remarks

In this thesis, we extended functional programming languages with aspect-oriented features, primarily to explore aspect-oriented enforcement of security policies.

AspectML In Chapter 2, we examined an aspect-oriented implementation of the Java security mechanism, which required the security advice to be triggered by functions with diverse argument and return types. We presented a new language, AspectML, that allowed a programmer to define type-safe polymorphic advice using pointcuts constructed from a collection of polymorphic join points. In particular, we focused on the synergy between polymorphism and aspect-oriented programming – the combination was clearly more expressive than the sum of its parts. At the simplest level, AspectML allowed programmers to reference control-flow points that appeared in polymorphic code. However, we also demonstrated that

polymorphic pointcuts were necessary even when the underlying code base was completely monomorphic. Otherwise, there was no way to assemble a collection of join points that appeared in code with different types. In addition, run-time type analysis allowed programmers to define polymorphic advice that behaved differently depending upon the type of its argument.

From a technical standpoint, we gave AspectML a semantics by compiling it into a typed intermediate calculus, $\mathbb{F}_A$. We proved the intermediate calculus was type-safe. The reason for giving AspectML a semantics this way was to first decompose complex source-level syntax into a series of simple and orthogonal constructs. Giving a semantics to the simple constructs of $\mathbb{F}_A$ and proving $\mathbb{F}_A$ sound was quite straightforward.

The definition of the intermediate calculus, $\mathbb{F}_A$, was also an important contribution of this work. An interesting element of $\mathbb{F}_A$ was the definition of our label hierarchy, which allowed us to form groups of related control flow points. Here, polymorphism was again essential: it was not possible to define these groups in a monomorphic language. The second interesting element of $\mathbb{F}_A$ was our support for reification of the current call stack. In addition to being polymorphic, our treatment of static analysis was more flexible, simpler semantically, and easier for programmers to use than the initial proposition in the core calculus of MinAML. Moreover, it was a good fit with standard data-driven functional programming idioms.

Finally, we compared our AspectML implementation of the Java security mechanism against the existing Java implementation using polymorphic advice. Through this case study, AspectML seemed useful to implement security advice.

Harmless Advice In Chapter 3, we examined how, in ordinary aspect-oriented programming, security and other advice added after the fact to an existing codebase could disrupt important data invariants and prevent local reasoning. Instead, we showed that many common aspects, including security advice, could be implemented as harmless advice. Harmless advice had the advantage that it could be added to a program after the fact, in the typical aspect-oriented style, without corrupting important mainline data invariants. As a result, programmers using harmless advice retained the ability to perform local reasoning about partial correctness of their programs.

Harmless advice used a novel type and effect system related to information-flow type systems to ensure that harmless advice could not modify the behavior of mainline code. We proved a noninterference property using a proof technique based on simultaneous execution of programs by Simonet and Pottier [45]. We then used this noninterference property, combined with the type safety of the translation from the source language to the core language, to show that aspects in our source language, HarmlessAML, were harmless.

Finally, to demonstrate the usefulness of harmless advice for security, we implemented in HarmlessAML many of the security examples used by the Naccio execution monitoring system by Evans and Twyman as harmless advice. We found that many access control tasks could be performed by harmless advice. As such, aspects to implement security tasks could be added after-the fact without modifying the underlying behavior of the main program.

Interference Policies Finally, in Chapter 4, we expanded HarmlessAML to allow programmers to create interference policies for system libraries to define how those

libraries can be used by aspects. These policies used a selection of compile-time type checking and run-time monitoring to enforce the desired degree of harmlessness on aspect-oriented programs.

To test our interference policy mechanism on a system library, we performed a file I/O case study. We demonstrated that, in addition to many other possibilities, interference policies could certainly be used to enforce the previous chapter's definition of "harmlessness" on a file I/O library. We formalized an underlying noninterfering file I/O library in the core calculus, extending the $\mathbb{F}_{HRM}$ and $\mathbb{F}_{HRM2}$ syntax, operational semantics, and type system to include a idealized file system. We proved that the new core file system was type-safe and preserved strong noninterference properties. We then demonstrated that we could define an interference and resource policy for the source-level file I/O library that defined a type-safe translation to the baked-in noninterfering core-level file I/O library. This allowed us to prove that aspects that use the source-level file system were harmless. Therefore, we showed that an interference policy on a system library specified by our policy language could continue to enforce our original view of harmlessness for advice that used that system library.

Bibliography

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Symposium on Network and Distributed System Security*, pages 107–121, Reston, VA, Feb. 2003. Internet Society.

[2] J. Aldrich. Open modules: Modular reasoning about advice. In *European Conference on Object-Oriented Programming*, pages 144–168, July 2005.

[3] P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere. Semantics of static pointcuts in AspectJ. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 11–23, New York, NY, USA, January 2007. ACM Press.

[4] L. Bauer, J. Ligatti, and D. Walker. Composing security policies in polymer. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–314, New York, NY, USA, June 2005. ACM Press.

[5] M. Blume and A. W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4):813–847, 1999.

[6] G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. muABC: A minimal aspect calculus. In P. Gardner and N. Yoshida, editors, *Proceedings of the 15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224, Berlin, Germany, Sept. 2004. Springer.

[7] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Foundations of Aspect-oriented Languages*, Apr. 2002.

[8] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 56–65. ACM Press, 2004.

[9] D. S. Dantas and D. Walker. Harmless advice. In *International Workshop on Foundations of Object-oriented Languages*, Jan. 2005.

[10] D. S. Dantas and D. Walker. Harmless advice. In *Symposium on Principles of Programming Languages*, pages 383–396, Jan. 2006.

[11] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. PolyAML: A polymorphic aspect-oriented functional programming language. In *International Conference on Functional Programming*, pages 306–319, Sept. 2005.

[12] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, To appear, 2007.

[13] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *International Conference on Aspect-oriented Software Development*, pages 141–150, Mar. 2004.

[14] Úlfar. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Canada, Sept. 1999.

[15] Úlfar. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000.

[16] D. Evans. *Policy-Directed Code Safety*. PhD thesis, MIT, 1999.

[17] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, Oakland, CA, May 1999.

[18] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[19] R. E. Filman and D. P. Friedman. *Aspect-Oriented Software Development*, chapter Aspect-Oriented Programming is Quantification and Obliviousness, pages 21–35. Addison-Wesley, Boston, MA, 2005.

[20] M. Fiuczynski, Y. Cody, R. Grimm, and D. Walker. Patch(1) considered harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pages 91–96. USENIX, July 2005.

[21] A. Gordon and C. Fournet. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.

[22] R. Harper and C. Stone. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press, 1998.

[23] F. Henglein, H. Makholm, and H. Niss. *Advanced Topics in Types and Programming Languages*, chapter 3, pages 87–135. MIT Press, Cambridge, MA, 2005.

[24] List of main users. AspectJ Users List: aspectj-users@eclipse.org, June 2004. Requires subscription to access archives.

[25] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java. In *ACM conference on Object-Oriented Programming, Systems, Languages and Applications*, volume 34 of *ACM Sigplan Notices*, pages 132–146, Denver, CO, Aug. 1999. ACM Press.

[26] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of typed aspect-oriented programs. Unpublished manuscript., 2003.

[27] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[28] R. Jagadeesan, A. Jeffrey, and J. Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 2006.

[29] S. Katz. Diagnosis of harmful aspects using regression verification. In *Foundations of Aspect-Oriented Languages*, Mar. 2004.

[30] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *European Conference on Object-oriented Programming*. Springer-Verlag, 2001.

[31] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *International Conference on Software Engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[32] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *European Conference on Real-time Systems*, York, UK, June 1999.

[33] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *Foundations of Software Engineering*, Oct.-Nov. 2004.

[34] D. K. Lee, K. Crary, and R. Harper. Towards a mechanized metatheory of Standard ML. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–184, New York, NY, USA, January 2007. ACM Press.

[35] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, June 1999.

[36] K. J. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual collaborations – combining modules and aspects. *The Computer Journal*, 46(5):542–565, September 2003.

[37] J. Ligatti, D. Walker, and S. Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming: Special Issue on Foundations of Aspect-Oriented Programming*, 63(3):240–266, Dec. 2006.

[38] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In G. T. Leavens and R. Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop*, pages 17–25, Apr. 2002.

[39] H. Masuhara, H. Tatsuzawa, and A. Yonezawa. Aspectual caml: an aspect-oriented functional language. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, pages 320–330, New York, NY, USA, Sept. 2005. ACM Press.

[40] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[41] A. C. Myers. Jflow: Practical mostly-static information flow control. In *ACM Symposium on Principals of Programming Languages*, pages 226–241, Jan. 1999.

[42] J. Ovlinger. *Combining Aspects and Modules*. PhD thesis, Northeastern University, Apr. 2004.

[43] G. D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[44] G. D. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6, pages 101–124. Edinburgh University Press, 1971.

[45] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003.

[46] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *International Symposium on Foundations of Software Engineering*, pages 147–158, 2004.

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

[48] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, Feb. 2000.

[49] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *European Software Engineering Conference/International Symposium on Foundations of Software Engineering*, pages 166–175, 2005.

[50] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. *Transactions on Programming Languages and Systems*, 2006. To appear.

[51] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *International Conference on Aspect-Oriented Software Development*, pages 158–167, Mar. 2003.

[52] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *Transactions on Programming Languages and Systems*, 26(5):890–910, 2004.

[53] M. Wang, K. Chen, and S.-C. Khoo. On the pursuit of staticness and coherence. In *Proceedings of the 5th Workshop on Foundations of Aspect-Oriented Languages*, Mar. 2006.